Qualcomm Technologies, Inc.

Telematics SDK

API Reference
v1.58.0

80-PF458-2 Rev. AJ

June 1, 2023

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Revision History

Revision	Date	Description
AJ	May 2023	Updated documentation for SDK release V1.58.0 changes
AH	Apr 2023	Updated documentation for SDK release V1.57.0 changes
AG	Mar 2023	Added documentation for thermal notifications
AF	Dec 2022	Added documentation for crypto accelerator
	Nov 2022	Added support for Third Party Service (TPS) eCall over IMS
AE	Nov 2022	Updated documentation for SDK release V1.54.0 changes
AD	Sep 2022	Updated documentation for SDK release V1.53.0 changes
AC	Jul 2022	Updated documentation
AB	Apr 2022	Added documentation for API access control
AA	Mar 2022	Addition of documentation for Call flow of control filesystem for ECALL operation
	Apr 2022	Added crypto APIs call flows
	Jan 2022	Updated C-V2X radio call flows
	Dec 2021	Addition of documentation for sensor self test APIs
	Dec 2021	Addition of documentation for ECALL and OTA operation APIs
	Nov 2021	Support for Xtra feature in location
	Sep 2021	Addition of new sub system Readiness APIs for Modem Config
	Aug 2021	Addition of Subsystem Restart handling in Telsdk
	Aug 2021	Added further EFS API related information for platform sub-system
	Jul 2021	Updated thermal subsystem readiness flow for new methodology
	Jun 2021	Addition of Remote SIM provisioning APIs and related call flows
	Jun 2021	Addition of documentation for platform subsystem
	Apr 2021	Addition of documentation for sensor subsystem
	Apr 2021	Addition of Serving System Manager APIs for data
	Mar 2021	Addition of calibration init status API for audio
	Feb 2021	Addition of new sub system Readiness APIs for Audio
	Jan 2021	Updated Data Subsystem Readiness flow for new methodology
	Feb 2021	Addition of configure NMEA types in location SDK
	Jan 2021	Addition of Audio Subsystem Restart Support
	Jan 2021	Addition of terrestrial positioning APIs and callflows in location SDK
	Nov 2020	Addition of Year of hardware and configure engine state APIs in location and callflows
	Nov 2020	Addition of cv2x SE-linux interface documentation
	Aug 2020	Addition of Location Manager APIs, Location Configurator APIs and callflows
	May 2020	Addition of Public Logging API
	May 2020	Addition of L2TP to Data Networking APIs and call flows
	Apr 2020	Addition of Security section with SE-linux interface documentation
	Mar 2020	Addition of Robust Location API in Location Configurator and call flows
	Feb 2020	Addition of Location Configurator APIs and call flows
	Jan 2020	Addition of Socks Proxy to Data Networking APIs and call flows
	Jan 2020	Addition of Data software bridge management APIs and call flows
	Nov 2019	Addition of Compressed audio format playback on voice paths APIs and call flows
	Nov 2019	Addition of Data Networking APIs and call flows
	Nov 2019	Addition of Audio Format Transcoding APIs and call flows

Table 1 Revision history (cont.)

Revision	Date	Description
	Oct 2019	Addition of Location constraint time uncertainty APIs and call flows
	Oct 2019	Addition of Location concurrent report APIs and call flows
	Oct 2019	Addition of Data Filter APIs and call flows
	Sep 2019	Addition of modem config APIs and related call flows
	Sep 2019	Addition of Audio APIs for compressed audio format playback and related call flows
	Sep 2019	Addition of Audio APIs for Loopback, Tone Generator and related call flows
	Sep 2019	Addition of Remote SIM APIs and call flows
	Jul 2019	Addition of Thermal shutdown manager APIs and call flows
	Jul 2019	Updated the Location APIs and call flows
	Jun 2019	Addition of Audio APIs for play, capture, DTMF and related call flows
	May 2019	Addition of TCU Activity Management APIs and call flows
	Mar 2019	Addition of Thermal manager APIs and call flows
	Jan 2019	Addition of Audio APIs and call flows
	Nov 2018	Addition of C-V2X APIs
	Oct 2018	Addition of Network Selection and Serving System Management APIs and call flows
	Jun 2018	Added Cellular Connection Management APIs
	Dec 2017	Added subscription APIs and section on Versioning and API Status
	Sep 2017	Initial release
	Nov 2022	Addition of WLAN management support and call flow

Contents

List of Figures

1 Introduction

1.1 Purpose

The Telematics software development kit (TelSDK) is a set of application programming interfaces (APIs) that provide access to the QTI-specific hardware and software capabilities.

This document is intended to act as a reference guide for developers by providing details of the TelSDK APIs, function call flows and overview of TelSDK architecture.

1.2 Scope

This document focusses on providing details of the TelSDK APIs. It assumes that the developer is familiar with Linux and C++11 programming.

1.3 Conventions

Function declarations, function names, type declarations, and code samples appear in a different font. For example,

```
#include
```

Parameter directions are indicated as follows:

Parameters

`in`	*paramname*	indicates an input parameter.
`out`	*paramname*	indicates an output parameter.
`in,out`	*paramname*	indicates a parameter used for both input and output.

Most APIs are asynchronous as underlying sub-systems such as telephony are asynchronous. API names follow the convention of using prefix " get " for synchronous calls and " request " for asynchronous calls. Asynchronous responses such as listener callbacks come on a different thread than the application thread.

1.4 SDK Versioning

The following convention is used for versioning the SDK releases

SDK version (major.minor.patch)

SDK_VERSION = 1.0.0

Major version: This number will be incremented whenever significant changes or features are introduced.

Minor version: This number will be incremented when smaller features with some new APIs are introduced.

Patch version: If the release only contains bug fixes, but no API change then the patch version would be incremented, No change in the actual API interface.

NOTE: Use telux::common::Version::getSdkVersion() API to query the current version of SDK or refer the VERSION file in the repository

1.5 Public API Status

Public APIs are introduced and removed (if necessary) in phases. This allows users of an existing API that is being Deprecated to migrate. APIs will be marked with note indicating whether the API is subject to change or if it is not recommended to use the API.

as follows:

- **Eval**: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.
- **Obsolete**: API is not preferred and a better alternative is available.
- **Deprecated**: API is not going to be supported in future releases. Clients should stop using this API. Once an API has been marked as Deprecated, the API could be removed in future releases.

2 Functional Overview

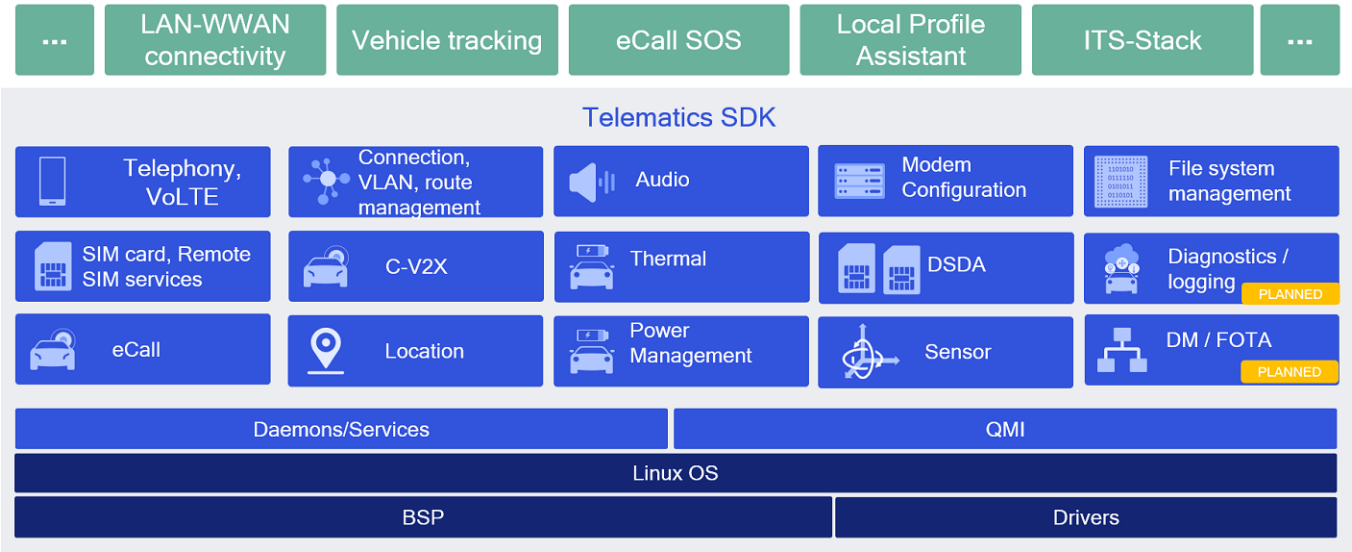

Figure 2-1 SDK-Overview

2.1 Overview

The Telematics library runs in the user space of the Linux system. It interacts with Telephony services and other sub-systems to provide various services like phone calls, SMS etc. These services are exposed by the SDK through fixed public APIs that are available on all Telematics platforms that support SDK. The Telematics APIs are grouped into the following functional modules:

Telephony

Telephony sub-system consists of APIs for functions related to Phone, Call, SMS and Signal Strength, Network Selection and Serving System Management.

SIM Card Services

SIM Card services sub-system consists of APIs to perform SIM card operations such as Send APDU messages to SIM card applications, SIM Access Profile(SAP) operations etc.

Location Services

Location Services sub-system consists of APIs to receive location details such as GNSS Positions, Satellite Vehicle information, Jammer Data signals, nmea and measurements information. The location manager sub-system also consists of APIs to get location system info, request energy consumed, get year of hardware information, get terrestrial position information and cancel terrestrial position information.

LocationConfigurator allows general engine configurations (example: TUNC, PACE etc),configuration of specific engines like SPE (example: minSVElevation, minGPSWeek etc) or DRE, deletion of warm and cold aiding data, NMEA configuration and support for XTRA feature.

Connection Management

Connection Management sub-system consists of APIs for establishing Cellular WAN/ Backhaul connection sessions and for Connection Profile Management etc.

Audio Management

Audio Management sub-system consists of APIs for Audio management such as setting up audio streams, switching devices on streams, apply volume/mute etc

Thermal Management

Thermal Management sub-system consists of APIs to get list of thermal zones, cooling devices and binding information. The sub-system also provides notifications about certain thermal related events such as when trip event occur for any thermal zone or cooling device changes its level.

Thermal Shutdown Mangement

Thermal shutdown management sub-system consists of APIs to get/set the thermal auto-shutdown mode and listen to its updates.

TCU Activity Management

TCU Activity Management sub-system consists of APIs to get TCU-activity state updates, set the TCU-activity state, etc.

Remote SIM Services

Remote SIM sub-system consists of APIs that allow a device that does not have a SIM physically connected to it to access a SIM remotely (e.g. over BT, Ethernet, etc.) and perform card operations on that SIM, such as requesting reset, transmitting APDU messages, etc.

Modem Config Services

Modem Config sub-system consists of APIs that allow to request modem config files, load/delete a modem config file from modem's storage, activate/deactivate a modem config file, get/set auto selection mode for config files.

Data Network Management

Data Network Management sub-system consists of APIs to setup VLAN, static NAT, Firewall, Socks, etc.

Sensor services

The sensor sub-system provides API to configure and acquire data from underlying hardware sensors like accelerometers, gyroscopes among others.

Platform services

The platform sub-system provides APIs to configure and control platform functionalities, like starting an EFS backup, control filesystem for ECALL and OTA operations. This sub-system also provides notifications about certain system related events, for instance filesystem events such as EFS restore and backup events.

Remote SIM Provisioning

Remote SIM provisioning provides API to add profile, delete profile, activate/deactivate profile on the embedded SIMs (eUICC) , get list of profiles, get server address like SMDP+ and SMDS and update SMDP+ address, update nick name of profile and retrieve Embedded Identity Document(EID) of the SIM.

Debug Logger

Logger consists of API that can be utilized to log messages from SDK Applications. **WLAN Management**

The WLAN management subsystem consists of APIs to configure, enable, and set access point and station configurations.

Telematics SDK classes can be broadly divided into the following types:

- Factory - Factory classes are central classes such as PhoneFactory which can be used to create Manager classes corresponding to their sub-systems such as PhoneManager.
- Manager - Manager classes such as PhoneManager to manage multiple Phone instances, CardManager to manage multiple SIM Card instances etc.
- Observer/ Listener - Listener for unsolicited responses.
- Command Callback - Single-shot response callback for asynchronous API requests.
- Logger - APIs to log messages, control the log levels.

2.2 Features

Telematics SDK provides APIs for the following features:

2.2.1 Call Management

CallManager, Phone and PhoneManager APIs of Telematics SDK provides call related control operations such as

- Initiate a voice call
- Answer the incoming call
- Hold the call
- Hangup waiting, held or active call

CallManager and PhoneManager also provides additional functionality such as

- Allowing conference, and switch between waiting or holding call and active call
- Emergency Call (dial 112)
- Third Party Service (TPS) Emergency Call (dial custom number)
- Notifications about call state change

2.2.2 SMS

SMS Manager APIs of Telematics SDK provides SMS related functionality such as

- Sends and receives SMS messages of type GSM7, GSM8 and UCS2

2.2.3 SIM Card Services

The SIM Card operations are performed by CardManager and SapCardManager.

CardManager APIs of Telematics SDK perform operations on UICC card such as

- Open or close logical/basic channel to ICC card
- Transmit Application Protocol Data Unit (APDU) to the ICC Card over logical/basic channel
- Receive response APDU from the ICC Card with the status
- Notify about ICC card information change

SapCardManager APIs provides SIM Access Profile(SAP) related functionality such as

- Open or close SIM Access Profile(SAP) connection
- Transmit Application Protocol Data Unit (APDU) over SAP connection
- Receive response APDU over SAP connection
- Perform SAP operations such as Answer to Reset(ATR), SIM Power off, SIM Power On, SIM Reset and fetch Card Reader status.

2.2.4 Phone Information

Phone APIs of Telematics SDK provides phone related information such as

- Get Service state of phone i.e. EMERGENCY_ONLY, IN_SERVICE and OUT_OF_SERVICE
- Get Radio state of device i.e RADIO_STATE_OFF, RADIO_STATE_ON and RADIO_STATE_UNAVAILABLE
- Retrieve the signal strength corresponding to the technology supported by SIM
- Device Identity
- Set or Request Operating Mode
- Subscription Information

2.2.5 Location Services

Location Services APIs of Telematics SDK provide the mechanism to register listener and to receive location updates, satellite vehicle information, jammer signals, nmea and measurement updates. The location manager sub-system also consists of APIs to get location system info, request energy consumed, get year of hardware information, get terrestrial position information and cancel terrestrial position information. Following parameters are configurable through the APIs.

- Minimum time interval between two consecutive location reports.
- Minimum distance travelled after which the period between two consecutive location reports depends on the interval specified.

LocationConfigurator allows general engine configurations (example: TUNC, PACE etc),configuration of specific engines like SPE (example: minSVElevation, minGPSWeek etc) or DRE, deletion of warm and cold aiding data, NMEA configuration and support for XTRA feature.

2.2.6 Data Services

Data Services APIs in the Telematics SDK used for cellular connectivity, modem profile management, filters management, and networking.

Data Connection Manager APIs provide functionality such as

- start / stop data call
- listen for data call state changes

Data Profile Manager APIs provide functionality such as

- List available profiles in the modem
- Create / modify / delete / modify modem profiles
- Query for the selected profile

Data Serving System Manager APIs provide functionality such as

- Get dedicated radio bearer status
- Request Modem Service Status
- Request Modem Roaming Status

Data Filter Manager APIs provide functionality such as

- get / set data filter mode per data call
- get / set data filter mode for all data call in up state
- add / remove data filter per data call
- add / remove data filter for all data call in up state

Data VLAN Manager APIs provide functionality such as

- Create / remove VLAN
- Query VLAN info
- Bind / unbind VLAN to PDN
- Query current VLAN to PDN mapping

Data Static LAN Manager APIs provide functionality such as

- Add / remove static LAN entry
- Request current static NAT entries

Data Firewall Manager APIs provide functionality such as

- Add / remove DMZ entry
- Query current DMZ entries
- Set Firewall configuration to enable / disable Firewall
- Query current status of Firewall
- Add / remove Firewall entry

- Query Firewall entry rules

Data Socks Manager APIs provide functionality such as

- Enable/Disable Socks feature

Data L2TP Manager APIs provides functionality such as

- Set L2TP configuration to enable/disable L2TP, TCP Mss and MTU size
- Add / remove L2TP tunnel
- Query active L2TP configuration

Data Software Bridge Manager provides interface to enable packet acceleration for non-standard WLAN and Ethernet physical interfaces. It facilitates to configure a software bridge between the interface and Hardware accelerator. Its APIs provide functionality such as

- Add / remove a software bridge
- Query the software bridges configured in the system
- Enable / Disable the software bridge management

Data Serving System Manager provides the interface to access network and modem low level services. The API includes method for:

- Request Modem Service Status
- Request Modem Roaming Status
- Register to get notifications when Service Status and Roaming status Change

2.2.7 Network Selection and Serving System Management

Network Selection and Service System Management APIs in the Telematics SDK used for configuring the networks and preferences

Network Selection Manager APIs provide functionality such as

- request or set network selection mode (Manual or Automatic)
- scan for available networks
- request or set preferred networks list

Serving System Manager APIs provide functionality such as

- request and set service domain preference and radio access technology mode preference for searching and registering (CS/PS domain, RAT and operation mode)

2.2.8 C-V2X

The C-V2X sub-system contains APIs that support Cellular-V2X operation.

Cellular-V2X APIs in the Telematics SDK include Cv2xRadioManager and Cv2xRadio interfaces.

Cv2xRadioManager provides an interface to a C-V2X capable modem. The API includes methods for

- Enabling C-V2X mode

- Disabling C-V2X mode
- Querying the status of C-V2X
- Updating the C-V2X configuration via a config XML file

Cv2xRadio abstracts a C-V2X radio channel. The API includes methods for

- Obtaining the current capabilities of the radio
- Listen for radio state changes
- Creating and Closing an RX subscription
- Creating and Closing a TX event-driven flow
- Creating and Closing a TX semi-persistent-scheduling (SPS) flow
- Updating TX SPS flow parameters
- Update Source L2 Info

2.2.9 Audio

The Audio subSystem contains of APIs that support Audio operation.

Audio APIs in the Telematics SDK include AudioManager, AudioStream, AudioVoiceStream, AudioPlayStream, AudioCaptureStream, AudioLoopbackStream, AudioToneGenerator, Transcoder interfaces.

AudioManager provides an interface for creation/deletion of audio stream. The API includes methods for

- Query readiness of subSystem
- Query supported "Device Types"
- Query supported "Stream Types"
- Creating Audio Stream
- Deleting Audio Stream

AudioStream abstracts the properties common to different stream types. The API includes methods for

- Query stream type
- Query routed device
- Set device
- Query volume details
- Set volume
- Query mute details
- Set mute

AudioVoiceStream along with inheriting AudioStream, provides additional APIs to manage voice call session as stated below.

- Start Voice Audio Operation
- Stop Voice Audio Operation

- Play DTMF tone
- Detect DTMF tone

AudioPlayStream along with inheriting AudioStream, provides additional APIs to manage audio play session as stated below.

- Write audio samples
- Write audio samples for compressed audio format
- Stop Audio for compressed audio format
- Play compressed format audio on voice paths

AudioCaptureStream along with inheriting AudioStream, provides additional APIs to manage audio capture session as stated below.

- Read audio samples

AudioLoopbackStream along with inheriting AudioStream, provides additional APIs to manage audio loopback session as stated below.

- Start loopback
- Stop loopback

AudioToneGeneratorStream along with inheriting AudioStream, provides additional APIs to manage audio tone generator session as stated below.

- Play tone
- Stop tone

Transcoder provides APIs to manage audio transcoder which is able to perform below operations.

- Convert one audio format to another

Audio SDK provides details of supported "Device Types" and "Stream Types" in the audio subsystem of Reference Telematics platform.

“Device Type” encapsulates the type of device supported in Reference Telematics platform. The representation of these devices would be made available via public header file <usr/include/telux/audio/AudioDefines.hpp>.

Example: DEVICE_TYPE_XXXX

Internally SDK DeviceTypes are mapped to Audio HAL devices as per <usr/include/system/audio.h>.

In current release it is mapped per below table.

Current Device Mapping Table:

SDK Audio Device Representation	Audio HAL Representation
DEVICE_TYPE_SPEAKER	AUDIO_DEVICE_OUT_SPEAKER
DEVICE_TYPE_MIC	AUDIO_DEVICE_IN_BACK_MIC

However Device Mapping is configurable as stated below. This configurability provides flexibility to map different Audio HAL devices to SDK representation.

Update tel.conf file with below details before boot of system.

NUM_DEVICES specifies the number of device types supported

DEVICE_TYPE specifies the SDK type of each device (in comma separated values)

DEVICE_DIR specifies the device direction for each device in order above (in comma separated values)

AHAL_DEVICE_TYPE specifies the mapped Audio HAL type of each device (in comma separated values)

Example:

Note: The default values provided here are based on QTI's reference design.

NUM_DEVICES=6

DEVICE_TYPE=1,2,3,257,258,259

DEVICE_DIR=1,1,1,2,2,2

AHAL_DEVICE_TYPE=2,1,4,2147483776,2147483652,2147483664

For any stream types, maximum device supported would be one. Single stream per multiple devices not supported. For voice stream Rx Device would decide corresponding Tx Device pair as decided by Audio HAL.

NOTE FOR SYSTEM INTEGRATORS:

The mapping of Audio devices to Audio HAL devices is static currently based on QTI's Reference Telematics platform. For custom platforms this mapping need to be updated.

“Stream Type” encapsulates the type of stream supported in Reference Telematics Platform. The representation of these stream types made available via public header file <usr/include/telux/audio/AudioDefines.hpp>.

Example: VOICE_CALL, PLAY, CAPTURE, LOOPBACK, TONE_GENERATOR etc

Volume Support Table:

This table captures scenarios where the volume could be modified.

Stream Type	Stream Direction (RX)	Stream Direction (Tx)
VOICE_CALL	Applicable	Not Applicable
PLAY	Applicable	Not Applicable
CAPTURE	Not Applicable	Applicable
LOOPBACK	Not Applicable	Not Applicable
TONE_GENERATOR	Not Applicable	Not Applicable

In case QTI's reference design does not support volume for specific stream category, API responds with error.

Mute Support Table:

This table captures when stream could be muted and in which direction.

Stream Type	Stream Direction (RX)	Stream Direction (Tx)
VOICE_CALL	Applicable	Applicable
PLAY	Applicable	Not Applicable

Stream Type	Stream Direction (RX)	Stream Direction (Tx)
CAPTURE	Not Applicable	Applicable
LOOPBACK	Not Applicable	Not Applicable
TONE_GENERATOR	Not Applicable	Not Applicable

In case QTI's reference design does not support mute for specific stream category, API responds with error.
Note: If mute operations is performed for play or capture stream direction(Tx or RX), the stream will get muted irrespective of the direction provided.

2.2.10 Thermal Management

Thermal Management APIs in the Telematics SDK are used for reading thermal zone, cooling device and binding information.

Thermal Management APIs provide functionality such as

- get thermal zones with thermal zone description, current temperature, trip points and binding info
- get cooling devices with cooling device type, maximum and current cooling level
- get thermal zone by Id
- get cooling device by Id

2.2.11 Thermal Shutdown Management

Thermal Shutdown Management APIs provide funtionality such as

- Query auto-shutdown mode.
- Set auto-shutdown mode.
- Get notifications on auto-shutdown mode updates.

2.2.12 TCU Activity Management

TCU-activity Manager APIs in Telematics SDK provides TCU-activity state related operations such as

- Set the activity state of the machines in TCU
- Get notifications about the imminent activity state changes on a machine in TCU
- Set the modem activity state
- Get all machine names
- Query the current TCU-activity state

2.2.13 Remote SIM

Remote SIM APIs in the Telematics SDK allow a device to use the WWAN capabilities of a SIM on another device.

Remote SIM APIs provide functionality such as

- Sending card events (reset, power up, errors) to the modem

- Sending/receiving APDU messages from/to the modem and remote SIM.
- Receiving operations from the modem (disconnect, power up, reset) to the remote SIM.

2.2.14 Modem Config Management

Modem Config APIs in the Telematics SDK provides modem config related functionalities such as

- Request modem config files from modem's storage.
- Load a modem config file to modem's storage.
- Activate/Deactivate a modem config file from modem's storage.
- Get Active config info details.
- Get/Set config auto selection mode.
- Delete a modem config file from modem's storage.
- Ability to get notified whenever a SW config file is activated.

2.2.15 Sensors

The sensor sub-system provides APIs to

- Configure and acquire continuous stream of data from an underlying sensor
- Create multiple clients for a given sensor, each of which can have their own configuration (sampling rate, batch count) for data acquisition.
- Query and control sensor features available on the hardware or those offerred by the software framework. Availability of sensor features depend on the sensor hardware being used and the capabilities it offers.

In addition to the sensor sub-system APIs, configuration items relevant to the underlying sensors are also available in /etc/sensors.conf on the device filesystem. This includes the range for the sensors, the limits on sampling frequency and batch count among other parameters.

2.2.16 Platform

The platform sub-system provides APIs to

- Register and listen to filesystem events such as EFS backup and restore notifications
- Request EFS backup

2.2.17 Remote SIM Provisioning

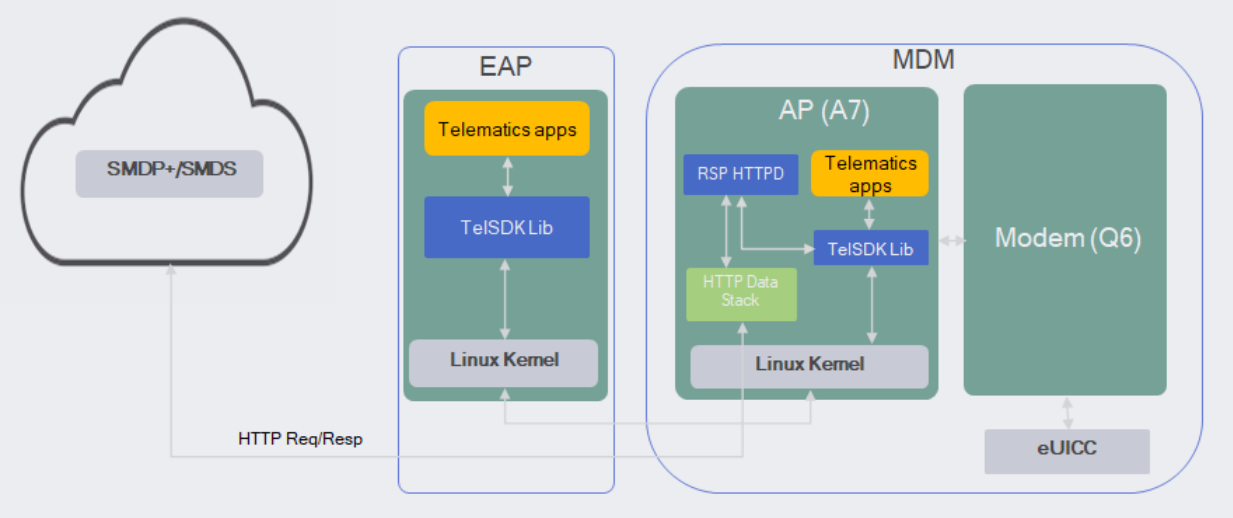

Figure 2-2 Remote SIM Provisioning

The Telematics Application can leverage Remote SIM Provisioning (RSP) APIs to perform eUICC profile management operations.

Remote SIM Provisioning APIs in Telematics SDK provides operations such as

- Download a profile on the eUICC. Allow downloading of profile based on activation code and confirmation code. Also provide user consent for downloading of profile.
- Enable or disable a profile to activate/deactivate subscription corresponding to profile.
- Delete a profile from an eUICC.
- Query list of profile on the eUICC.
- Get and update the server address(SMDP+ and SMDS)
- Get EID of the eUICC.
- Update nickname of the profile.
- Perform memory reset which allows to delete test and operational profiles or reset to default SMDP+ address.

When modem LPA/eUICC needs to reach SMDP+/SMDS server on the cloud for HTTP transaction, the HTTP request is sent to RSP service i.e RSP HTTP daemon. The RSP HTTP Daemon performs these HTTP transactions on behalf of modem with SMDP+/SMDS server running on the cloud. The HTTP response from cloud is sent back to modem LPA/eUICC to take appropriate action.

2.2.18 Debug Logger

Logging APIs in the Telematics SDK provides logging related functionalities such as

- Runtime configurable logging to console, diag and file.
- Possible LOG_LEVEL values are NONE, PERF, ERROR, WARNING, INFO, DEBUG.

2.2.19 WLAN Management

WLAN management APIs in the Telematics SDK provide services related to the following Wi-Fi functionality.

- Enable/disable WLAN.
- Set/request WLAN mode: number of access points and number of stations to be enabled.
- Request current WLAN status.
- Set/request an access points configuration.
- Request access point status
- Set/request a station's configuration.
- Request station status
- Request list of devices connected to any access point.
- Restart hostapd and wpa_supplicant daemons

2.3 Subsystem Restart

Subsystem restart events occur when device operating system or services crashes due to any reason and then reboots to operational state. This section explains notifications that are sent to application when such an event occurs, the impact on application, notifications that are sent to application when device recovers to operating state, and suggested action application should take after recovery.
Examples of Subsystem Restart events:

- External application processor crash
- Modem application processor crash
- Modem processor crash

If application is running on either application processors when it crashes, application is expected to be restarted to initial state. For other scenarios, details are explained below for each subsystem.

2.3.1 Data Services

Data services behavior when Subsystem Restart event occur is shown in table below:

Telsdk Notifications	Impact	Recovery	Suggested Action
“onServiceStatusChange” notification with SERVICE_UNAVAILABLE “onDataCallInfoChanged” Notification with IDataCall object with DataCallStatus “NET_NO_NET”.	Data calls impacted by SSR will be torn down and “NET_NO_NET” notifications will be delivered for those calls. Based on SSR type, it is possible that certain data calls will remain active. No “NET_NO_NET” notification will be delivered for active calls.	Application receives “onServiceStatusChange” notification with “SERVICE_AVAILABLE”	Re-trigger wwan data call for which “NET_NO_NET” notification was received.

Figure 2-3 Data SSR and Recovery

2.4 Security

2.4.1 SELinux

SELinux is an access control framework provided by the Linux kernel. It provides a mechanism to restrict/control access to system resources such as file nodes and sockets. SELinux framework expects any process running in userspace to declare all its interactions with the system resources in the form of SELinux policies. On platforms enabled with SELinux, an app that uses an SDK API would also need to declare its usage through SELinux policies to ensure that it has all the required permissions.

Listed below are the SELinux Interfaces which are generic for any API in particular namespace that app needs to declare in its policies.

Note For the below list, let us consider the application's security context to be `app_t` (also called domain context).

Namespace	SELinux interface	Arguments	Usage
tel	telux_allow_tel()	domain context	telux_allow_tel(app_t)
data	telux_allow_data()	domain context	telux_allow_data(app_t)
audio	telux_allow_audio()	domain context	telux_allow_audio(app_t)
loc	telux_allow_loc()	domain context	telux_allow_loc(app_t)
thermal	telux_allow_thermalmanager()	domain context	telux_allow_thermalmanager(app_t)
power	telux_allow_power()	domain context	telux_allow_power(app_t)
config	telux_allow_modemconfig()	domain context	telux_allow_modemconfig(app_t)
cv2x	telux_allow_v2x()	domain context	telux_allow_v2x(app_t)
sensor	telux_allow_sensor()	domain context	telux_allow_sensor(app_t)
platform	telux_allow_platform()	domain context	telux_allow_platform(app_t)

The following example illustrates how an application can incorporate the SELinux interfaces exposed by SDK in its SELinux policies. Below code snippet is part of a Type Enforcement (TE) file of the application which grants required permissions to perform SDK data operations.

```
policy_module(application, 1.0)
type app_t;

#Granting SDK data client permissions to the application

telux_allow_data(app_t);
```

In addition to the above SELinux interfaces list, below are the SELinux interfaces specific to a usecase. When an app needs to use any API, it should identify which SELinux interface to be used corresponding to the permission type from the below table and add it to the policy file.

To determine which permission type to be used for a API, please refer to the documentation for the API in the API Reference or in the API header file

Note The system integrator has the option to turn on/off this feature, where APIs related to a particular use case require certain permissions. If this feature is turned off, the use case specific permissions listed below are not required by the caller.

Tech Area	Permission Type	SELinux Interface
Audio	TELUX_AUDIO_VOICE	telux_allow_audio_voice
	TELUX_AUDIO_PLAY	telux_allow_audio_play
	TELUX_AUDIO_FACTORY_TEST	telux_allow_audio_factory_test
	TELUX_AUDIO_CAPTURE	telux_allow_audio_capture
	TELUX_AUDIO_TRANSCODE	telux_allow_audio_transcode
Data	TELUX_DATA_SETTING	telux_allow_data_setting
	TELUX_DATA_CALL_OPS	telux_allow_data_call_ops
	TELUX_DATA_CALL_PROPS	telux_allow_data_call_props
	TELUX_DATA_PROFILE_OPS	telux_allow_data_profile_ops
	TELUX_DATA_FILTER_OPS	telux_allow_data_filter_ops
	TELUX_DATA_NETWORK_CONFIG	telux_data_network_config
ModemConfig	TELUX_CONFIG_MODEM_CONFIG	telux_allow_config_modem_config
Power	TELUX_POWER_CONTROL_STATE	telux_allow_power_control_state_t
Sensor	TELUX_SENSOR_DATA_READ	telux_allow_sensor_data_read
	TELUX_SENSOR_PRIVILEGED_OPS	telux_allow_sensor_privileged_ops
	TELUX_SENSOR_FEATURE_CONTROL	telux_allow_sensor_feature_control
Telephony	TELUX_TEL_CARD_POWER	telux_allow_tel_card_power
	TELUX_TEL_CARD_OPS	telux_allow_tel_card_ops
	TELUX_TEL_PRIVATE_INFO_READ	telux_allow_tel_private_info_read
	TELUX_TEL_CARD_PRIVILEGED_OPS	telux_allow_tel_card_privileged_ops
	TELUX_TEL_SAP	telux_allow_tel_sap
	TELUX_TEL_CELL_BROADCAST_CO↩ NFIG	telux_allow_tel_cell_broadcast_config
	TELUX_TEL_CELL_BROADCAST_LIS↩ TEN	telux_allow_tel_cell_broadcast_listen
	TELUX_TEL_SIM_PROFILE_OPS	telux_allow_tel_sim_profile_ops
	TELUX_TEL_SIM_PROFILE_USER_CO↩ NSENT	telux_allow_tel_sim_profile_user_↩ consent
	TELUX_TEL_SIM_PROFILE_CONFIG	telux_allow_tel_sim_profile_config
	TELUX_TEL_SIM_PROFILE_READ	telux_allow_tel_sim_profile_read
	TELUX_TEL_SIM_PROFILE_HTTP_PR↩ OXY	telux_allow_tel_sim_profile_http_proxy
	TELUX_TEL_REMOTE_SIM	telux_allow_tel_remote_sim
	TELUX_TEL_SMS_OPS	telux_allow_tel_sms_ops
	TELUX_TEL_SMS_LISTEN	telux_access_tel_sms_listen_t
	TELUX_TEL_SMS_CONFIG	telux_access_tel_sms_config
	TELUX_TEL_IMS_SETTINGS	telux_allow_tel_ims_settings
	TELUX_TEL_SRV_SYSTEM_CONFIG	telux_allow_tel_srv_system_config
	TELUX_TEL_SRV_SYSTEM_READ	telux_allow_tel_srv_system_read
	TELUX_TEL_NETWORK_SELECTION_↩ OPS	telux_allow_tel_network_selection_ops
	TELUX_TEL_NETWORK_SELECTION_↩ READ	telux_allow_tel_network_selection_read

Tech Area	Permmission Type	SELinux Interface
	TELUX_TEL_MULTISIM_MGMT	telux_allow_tel_multisim_mgmt
	TELUX_TEL_PRIVATE_INFO_READ	telux_allow_tel_private_info_read
	TELUX_TEL_SUB_PRIVATE_INFO	telux_allow_tel_sub_private_info_read
	TELUX_TEL_SUBSCRIPTION_READ	telux_allow_tel_subscription_read
	TELUX_TEL_CALL_INFO_READ	telux_allow_tel_call_info_read
	TELUX_TEL_CALL_MGMT	telux_allow_tel_call_mgmt
	TELUX_TEL_CALL_PRIVATE_INFO	telux_allow_tel_call_private_info
	TELUX_TEL_EMERGENCY_OPS	telux_allow_tel_emergency_ops
	TELUX_TEL_ECALL_MGMT	telux_allow_tel_ecall_mgmt
	TELUX_TEL_PHONE_MGMT	telux_allow_tel_phone_mgmt
	TELUX_TEL_PRIVATE_INFO_READ	telux_access_tel_private_info_read
	TELUX_TEL_PHONE_CONFIG	telux_allow_tel_phone_config
	TELUX_TEL_ECALL_CONFIG	telux_allow_tel_ecall_config
	TELUX_TEL_SUPP_SERVICES	telux_allow_tel_supp_services

The following example illustrates how to declare permissions for an application that wants to use telux::data::IDataConnectionManager::startDataCall() to setup a cellular data connection.The documentation of this API indicates that the caller needs to have TELUX_DATA_CALL_OPS permission. From the above table, the permission maps to **telux_allow_data_call_ops** SELinux interface:

In order for the app to use the API the below code snippet needs to be entered in the Type Enforcement (TE) file of the application.

```
policy_module(application, 1.0)
type app_t;

#Allow data call operations

telux_allow_data_call_ops(app_t)
```

3 Call Flow Diagrams

3.1 Application initialization call flow

TelSDK initializes various sub-systems during start-up. It marks each sub-system as ready once the initialization procedures are completed for that sub-system. The application has to wait until the corresponding sub-system is ready on which it needs to make API requests. TelSDK provides APIs to check whether sub-system is ready or not.

Example:

3.1.1 Phone manager initialization

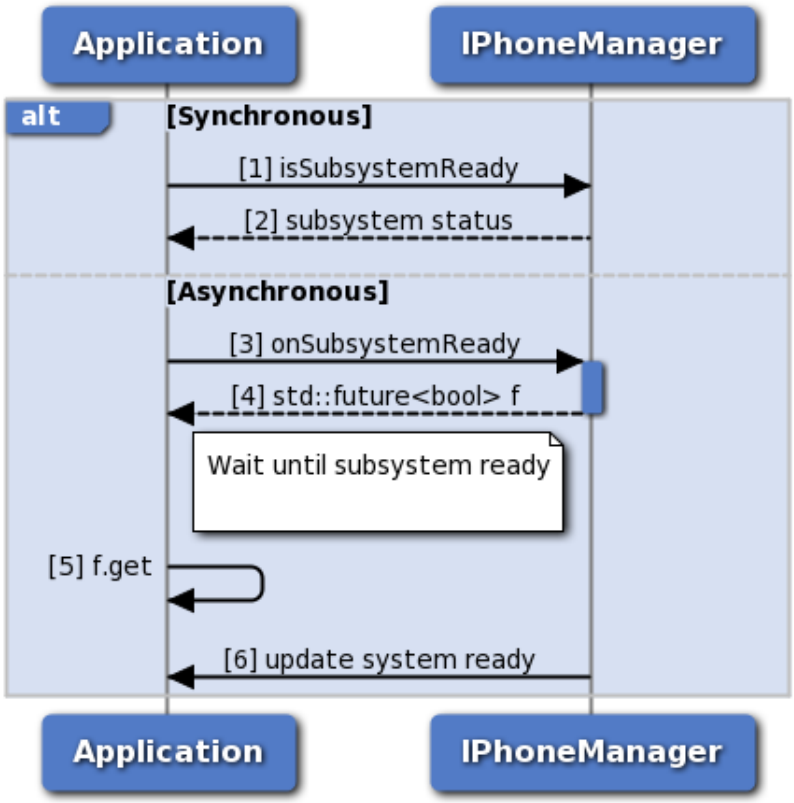

Figure 3-1 Phone manager initialization

1. Application can use IPhoneManager::isSubsystemReady to determine if the system is ready.
2. The application receives the status i.e. either true or false whether sub-system is ready or not.

3. If it is not ready, then the application could use onSubsystemReady which returns std::future.
4. PhoneManager notifies the application when the subsystem is ready through the std::future object.
5. The application waits until the asynchronous operation i.e onSubsystemReady completes.
6. PhoneManager updates the application once sub-system initialization completes.

3.2 Telephony

3.2.1 Dial call flow

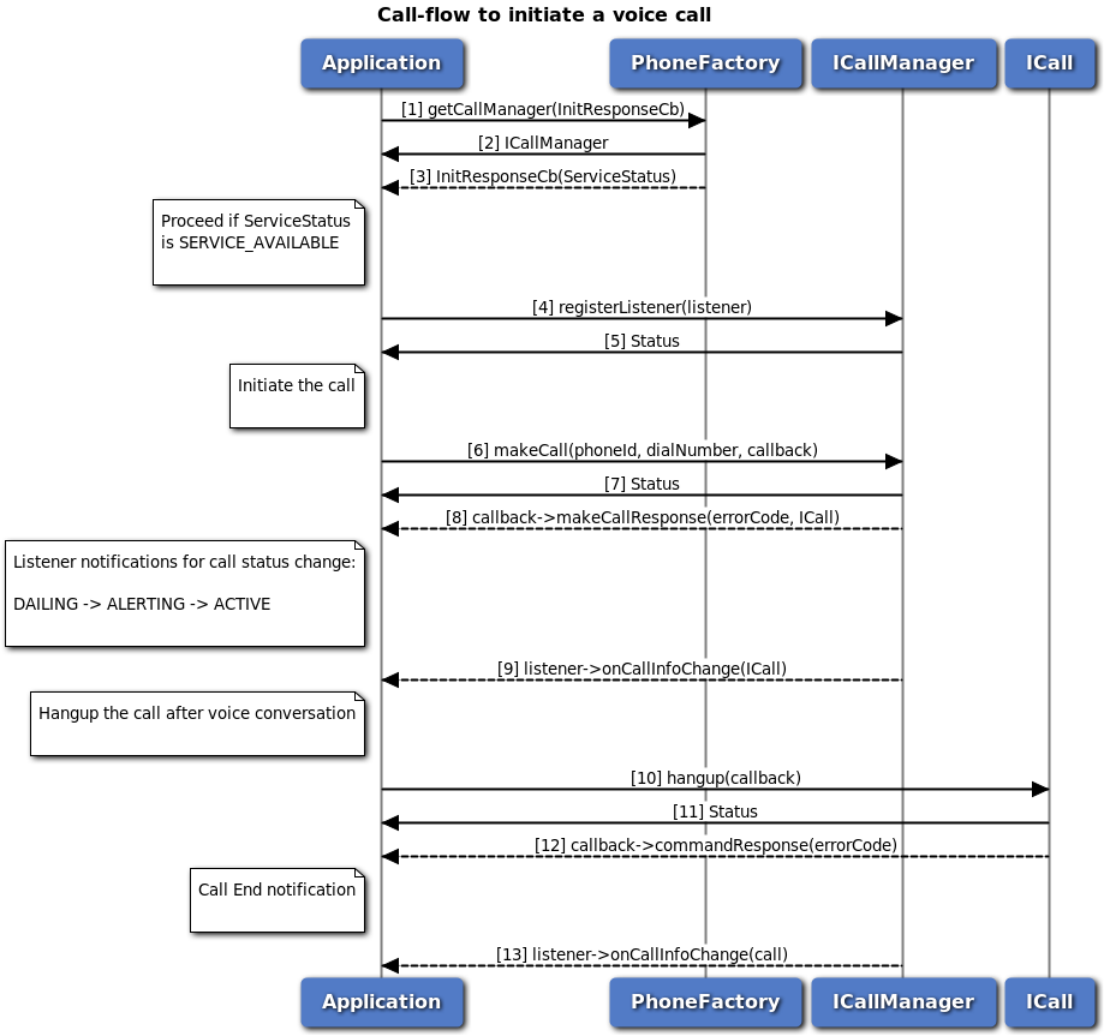

Figure 3-2 Dial call flow

1. Application requests an instance of Call Manager object using PhoneFactory, providing the initialization callback.
2. Application receives a Call Manager instance.
3. Application waits for the subsystem initialization callback, which notifies the subsystem initialization status.

4. If the subsystem is initialized successfully, application registers a listener for call info change notifications like DIALING, ALERTING, ACTIVE and ENDED.
5. Application receives the status(SUCCESS or suitable failure) based on registration of listener to CallManager.
6. Application dials a number by using makeCall API, optionally specifying callback to get asynchronous response.
7. Application receives the status(SUCCESS or suitable failure) based on the execution of makeCall API.
8. Optionally, the application gets asynchronous response for makeCall operation status, through makeCallResponse callback.
9. Application receives the listener notifications on call status change like DIALING/ALERTING/ACTIVE when other party accepts the call.
10. Application sends hangup command to hangup the call, optionally specifying callback to get asynchronous response.
11. Application receives the status(SUCCESS or suitable failure) based on the execution of hangup API.
12. Optionally, the application gets asynchronous response for hangup using CommandResponseCallback.
13. Application receives the listener notification for the call end notification.

3.2.2 ECall call flow

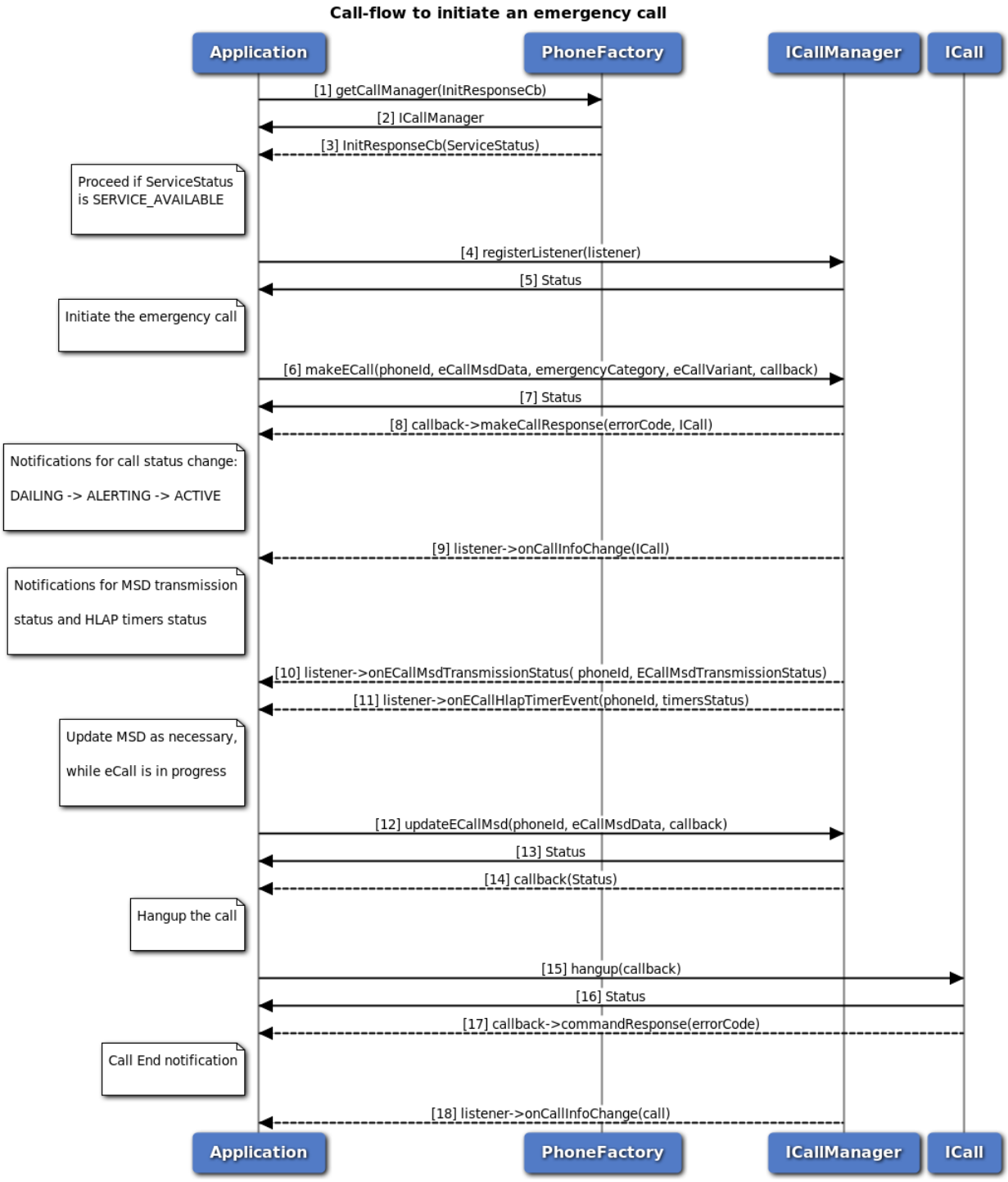

Figure 3-3 ECall call flow

1. Application requests an instance of Call Manager object using PhoneFactory, providing the initialization callback.
2. Application receives a Call Manager instance.

3. Application waits for the subsystem initialization callback, which notifies the subsystem initialization status.
4. If the subsystem is initialized successfully, application registers a listener for call info change notifications like DIALING, ALERTING, ACTIVE and ENDED.
5. Application receives the status(SUCCESS or suitable failure) based on registration of listener to CallManager.
6. Application dials an emergency call by using makeECall API, optionally specifying callback to get asynchronous response.
7. Application receives the status(SUCCESS or suitable failure) based on the execution of makeECall API.
8. Optionally, the application gets asynchronous response for makeECall operation status, through makeCallResponse callback.
9. Application receives the listener notifications on call status change like DIALING/ALERTING/ACTIVE when other party accepts the call.
10. Application receives the listener notifications related to MSD transmission status updates.
11. Application receives the listener notifications related to eCall HLAP timer status updates.
12. Optionally, the application can update the MSD using updateECallMsd API, whenever there is an update to MSD.
13. Application receives the status(SUCCESS or suitable failure) based on the execution of updateECallMsd API.
14. Optionally, the application gets asynchronous response for updateECallMsd operation status.
15. Application sends hangup command to hangup the call, optionally specifying callback to get asynchronous response.
16. Application receives the status(SUCCESS or suitable failure) based on the execution of hangup API.
17. Optionally, the application gets asynchronous response for hangup using CommandResponseCallback.
18. Application receives the listener notification for the call end notification.

3.2.3 TPS eCall over IMS call flow

Private eCall:

1. It is a normal VOLTE call(to a custom number) with MSD information.
2. Application processor (AP) sends the MSD data at call connect and later AP gets an explicit indication, upon which it can provide updated MSD(unlike standard eCall where AP updates MSD constantly)
3. Device doesn't support fallback to CS. It is the AP's responsibility to retry over CS.
4. AP can send the MSD information data in non standard format and PSAP should be able to recognize it.

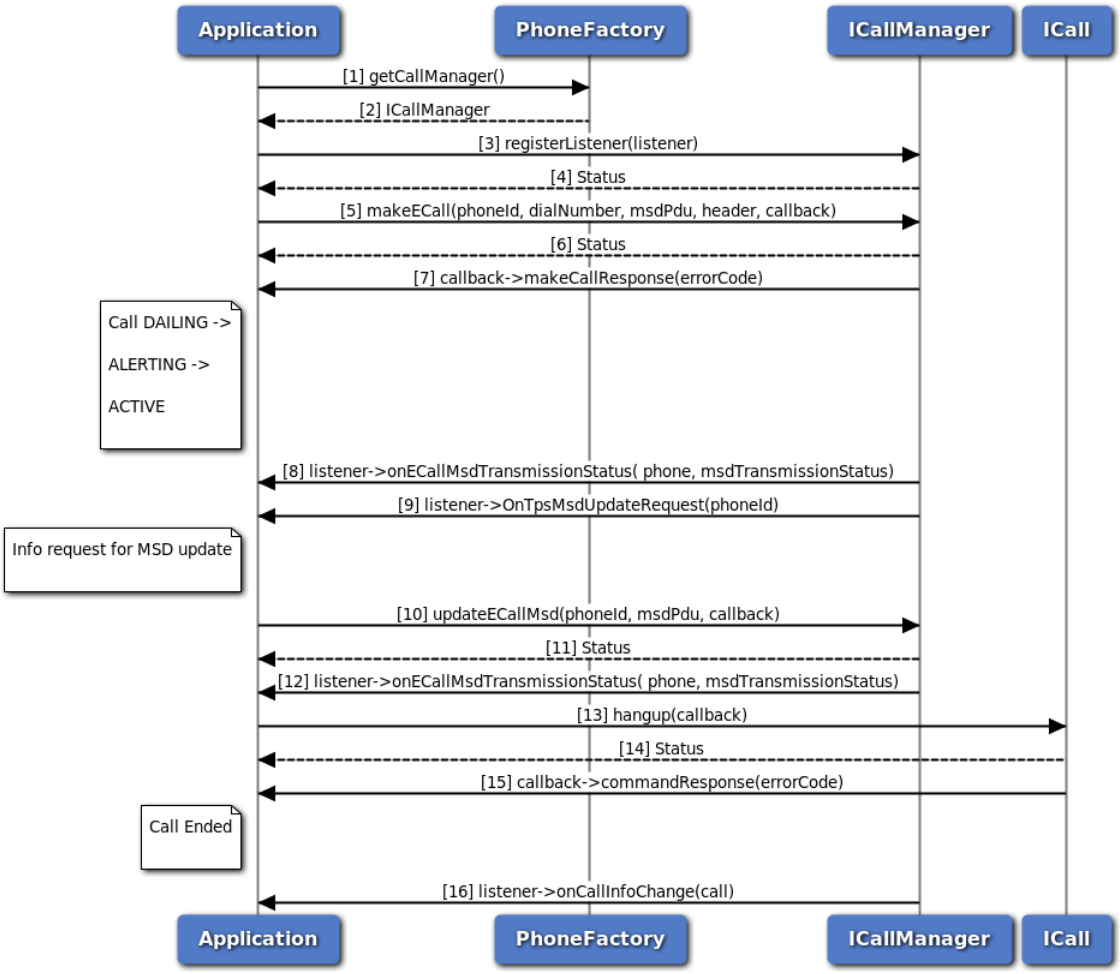

Figure 3-4 ECall call flow

1. The application gets the CallManager object using PhoneFactory.
2. The application receives the CallManager object in order to make eCall.
3. The application registers a listener with CallManager to listen to the call info change notifications like DIALING, ALERTING, ACTIVE etc.
4. The application receives the status like SUCCESS, FAILED etc based on registration of listener from

CallManager.

5. The application dials a TPS emergency call over IMS(i.e normal VOLTE call) by using makeECall API, optionally specifying header and callback to get asynchronous response.
6. The application receives the status like SUCCESS, FAILED etc based on the execution of makeECall API.
7. Optionally, the application gets asynchronous response for makeECall using makeCallResponseCallback.
8. CallManager sends eCall MSD transmission status to the application by using onECallMsdTransmissionStatus API.
9. CallManager sends MSD update request recieved from network to the application by using OnTpsMsdUpdateRequest API.
10. The application sends the MSD data to network using updateECallMsd API, optionally specifying callback to get asynchronous response.
11. The application receives the status like SUCCESS, FAILED etc based on the execution of updateECallMsd API.
12. CallManager sends eCall MSD transmission status to the application by using onECallMsdTransmissionStatus API for MSD sent at step 10.
13. The application sends hangup command to hangup the call, optionally gets asynchronous response using callback.
14. The application receives the status like SUCCESS, FAILED etc based on the execution of hangup API.
15. Optionally, the application gets asynchronous response for hangup using CommandResponseCallback.
16. CallManager sends call info change i.e Call Ended to the application by using onCallInfoChange callback function.

3.2.4 Signal strength call flow

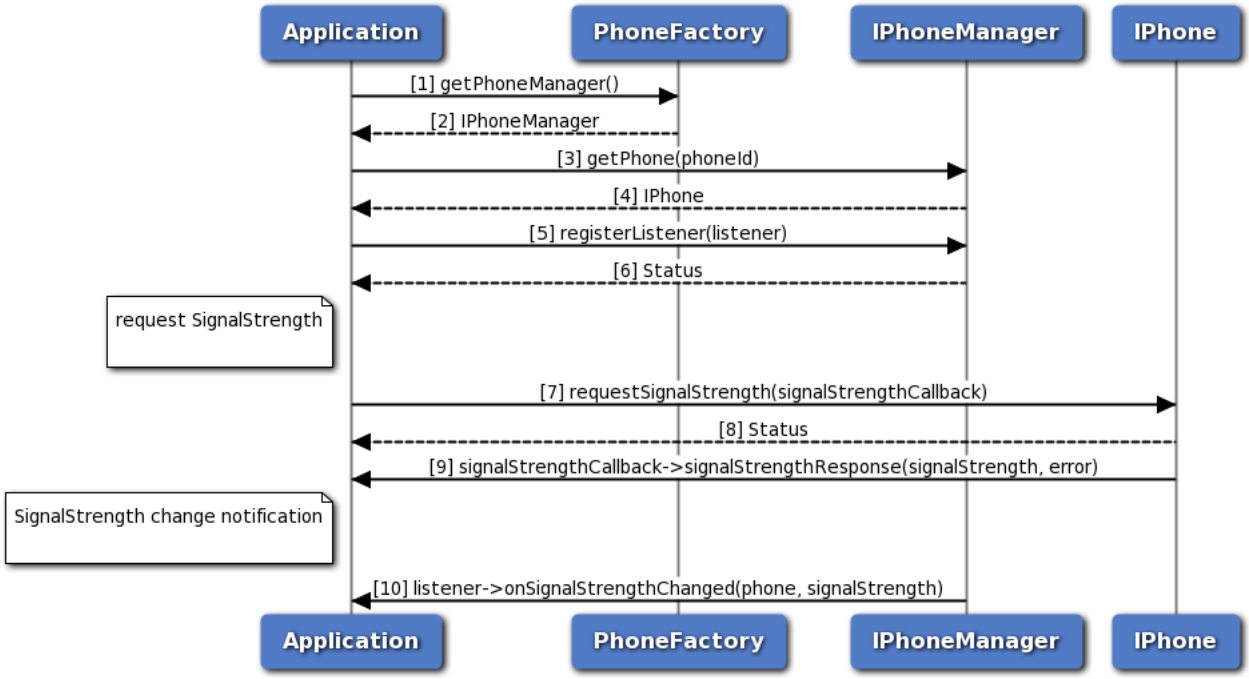

Figure 3-5 Signal strength call flow

1. The application gets PhoneManager object using PhoneFactory.
2. The application receives the PhoneManager object in order to get Phone instance.
3. The application gets phone instance for a given phone identifier using PhoneManager object.
4. PhoneManager returns IPhone object to the application.
5. Application registers the listener to get notification for signal strength change.
6. The application receives the status i.e. either SUCCESS or FAILED based on the registration of the listener.
7. The application requests for signal strength and optionally, gets asynchronous response using ISignalStrengthCallback.
8. The application receives the status i.e. either SUCCESS or FAILED based on execution of requestSignalStrength API in SapCardManager.
9. Optionally, the response for signal strength request is received by the application.
10. Application receives a notification when there is a change in signal strength.

3.2.5 Answer, Reject, RejectWithSMS call flow

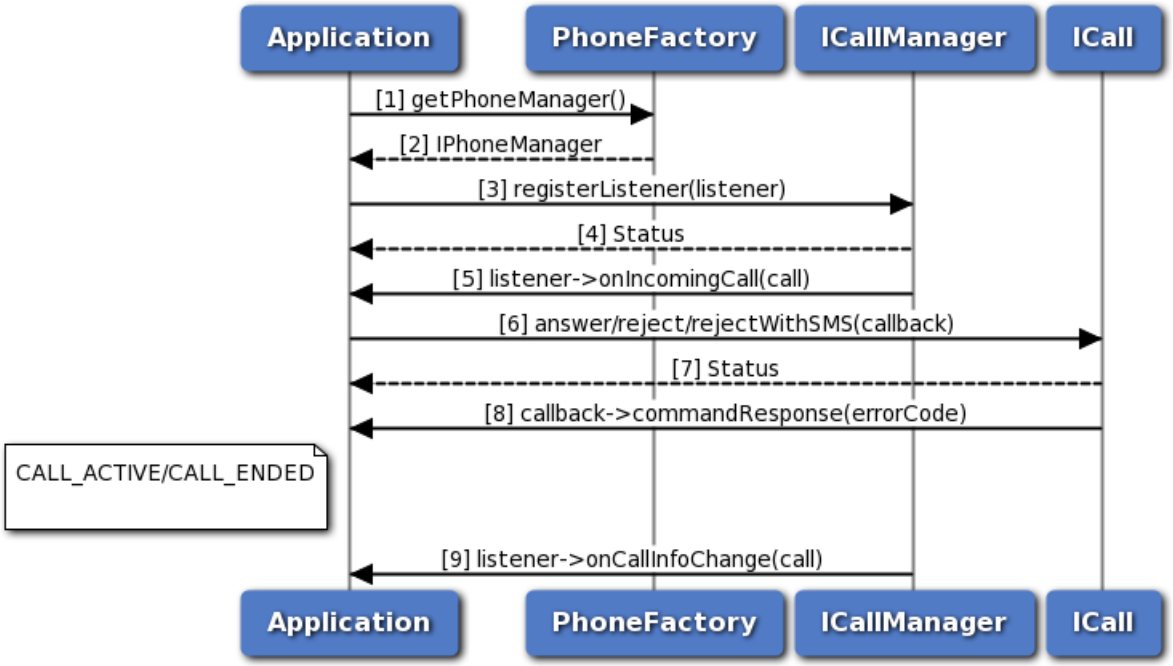

Figure 3-6 Answer, Reject, RejectWithSMS call flow

1. The application gets the PhoneManager object using PhoneFactory.
2. The application receives the PhoneManager object in order to register listener.
3. The application registers a listener with CallManager to listen incoming call notifications.
4. The application receives the status like SUCCESS, FAILED etc based on registration of listener from CallManager.
5. The application receives onIncomingCall notification when there is an incoming call.
6. The application performs answer/reject/rejectWithSMS operation using ICall.
7. The application receives the status like SUCCESS, FAILED etc based on execution of answer/reject/rejectWithSMS.
8. Optionally, the application gets asynchronous response for answer/reject/rejectWithSMS using CommandResponseCallback.
9. The CallManager sends call info change i.e CALL_ACTIVE or CALL_ENDED to the application by using onCallInfoChange callback function.

3.2.6 Hold call flow

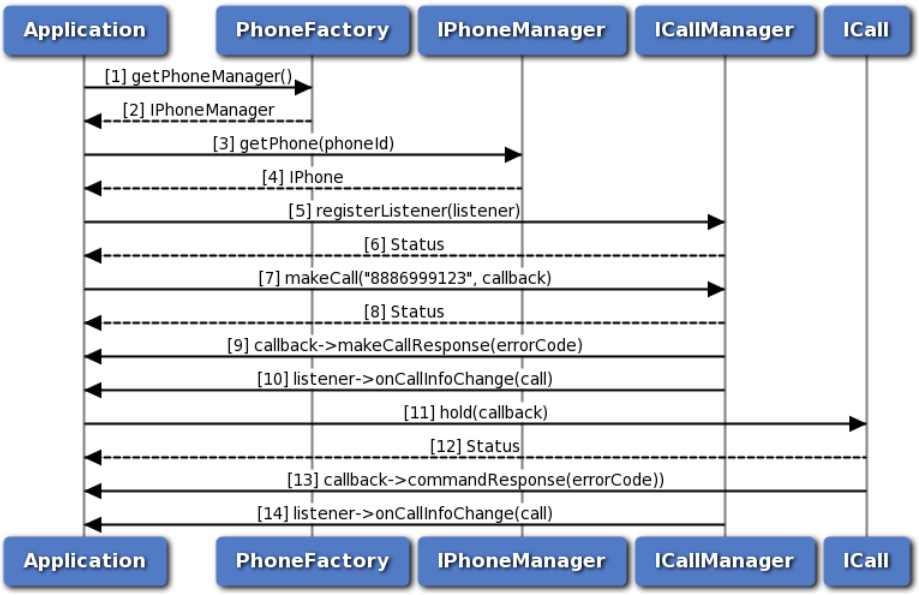

Figure 3-7 Hold call flow

1. The application gets the PhoneManager object using PhoneFactory.
2. The application receives the PhoneManager object in order to get Phone.
3. The application gets the Phone object for given phone identifier using PhoneManager.
4. PhoneManager returns Phone object to application, returns default phone in case phone identifier is not specified.
5. The application registers a listener with CallManager to listen to the call info change notifications like DIALING, ALERTING, ACTIVE etc.
6. The application receives the status like SUCCESS or INVALIDPARAM based on registration of listener to CallManager.
7. The application dials a number by using makeCall API, optionally specifying callback to get asynchronous response.
8. The application receives the status like SUCCESS, INVALIDPARAM and FAILED etc based on the execution of makeCall API.
9. Optionally, the application gets asynchronous response for makeCall using makeCallResponseCallback.
10. The CallManager sends call info change i.e CALL_ACTIVE to application by using onCallInfoChange API.
11. The application sends hold command to hold the call, optionally specifying callback to get asynchronous response.
12. The application receives the status like SUCCESS, FAILED etc based on the execution of hold API.
13. Optionally, the application gets asynchronous response for hold using CommandResponseCallback.
14. The application receives call info change i.e CALL_ON_HOLD from CallManager.

3.2.7 Hold, Conference, Swap call flow

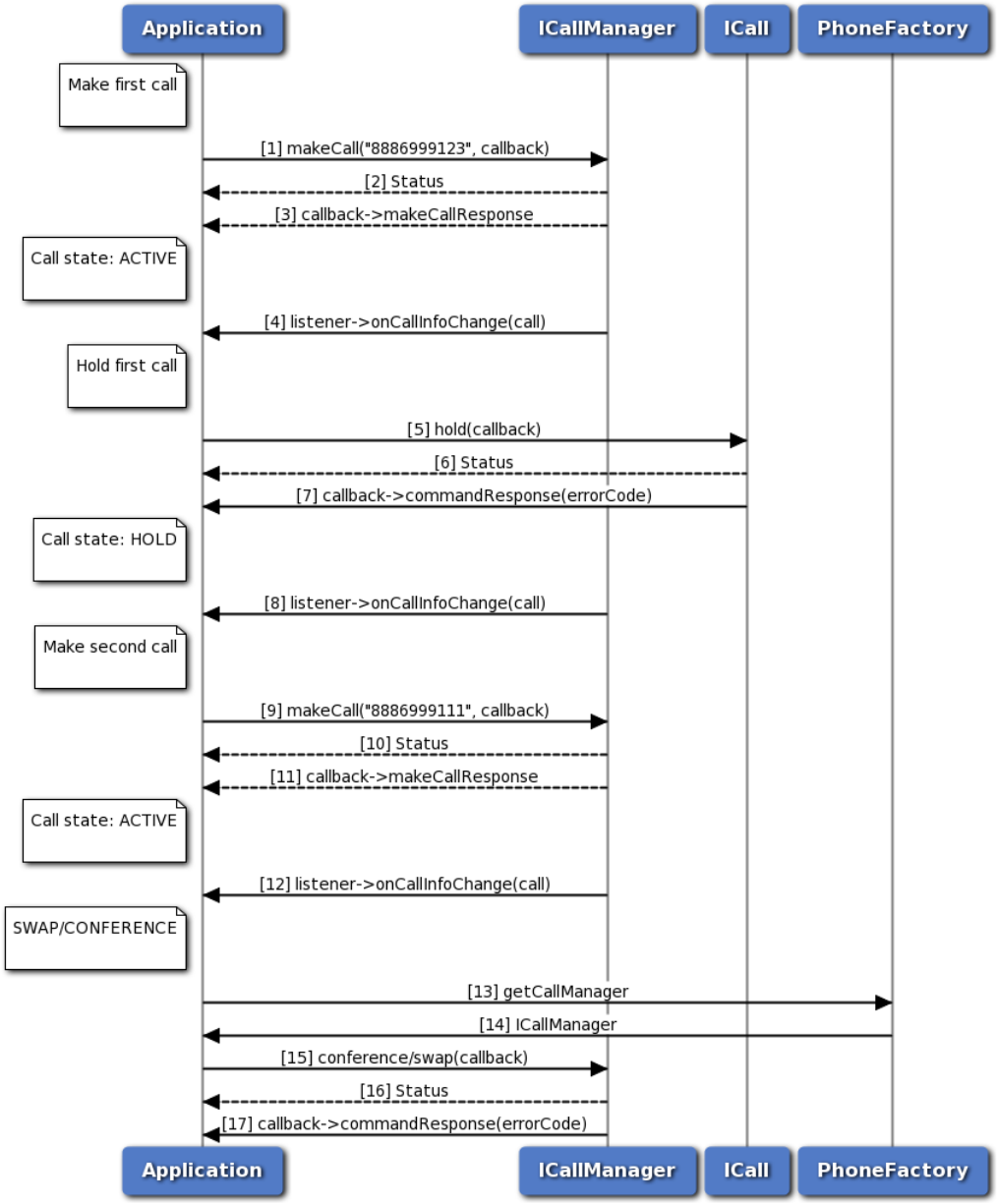

Figure 3-8 Hold, Conference, Swap call flow

1. The application makes first call using ICall.
2. The application receives the status like SUCCESS, FAILED etc based on execution of makeCall operation.
3. Optionally, the application gets asynchronous response for makeCall using makeCallResponseCallback.
4. The CallManager sends call info change i.e CALL_ACTIVE to application by using onCallInfoChange API.
5. The application sends hold command to hold the call, optionally specifying callback to get

asynchronous response.

6. The application receives the status like SUCCESS, FAILED etc based on the execution of hold API.
7. Optionally, the application gets asynchronous response for hold using CommandResponseCallback.
8. The application receives call info change i.e CALL_ON_HOLD from CallManager.
9. The application makes second call using ICall.
10. The application receives the status like SUCCESS, FAILED etc based on execution of makeCall operation.
11. Optionally, the application gets asynchronous response for makeCall using makeCallResponseCallback.
12. The CallManager sends call info change i.e CALL_ACTIVE to application by using onCallInfoChange API.
13. The application requests the PhoneFactory to get ICallManager object.
14. The application receives the ICallManager object using PhoneFactory.
15. The application performs conference/swap operation using ICallManager by passing first call and second call. optionally application can pass callback to receive hold response asynchronously.
16. The application receives the status like SUCCESS, FAILED etc based on the execution of conference/swap API.
17. Optionally, the application gets asynchronous response for conference/swap using CommandResponseCallback.

3.2.8 SMS call flow

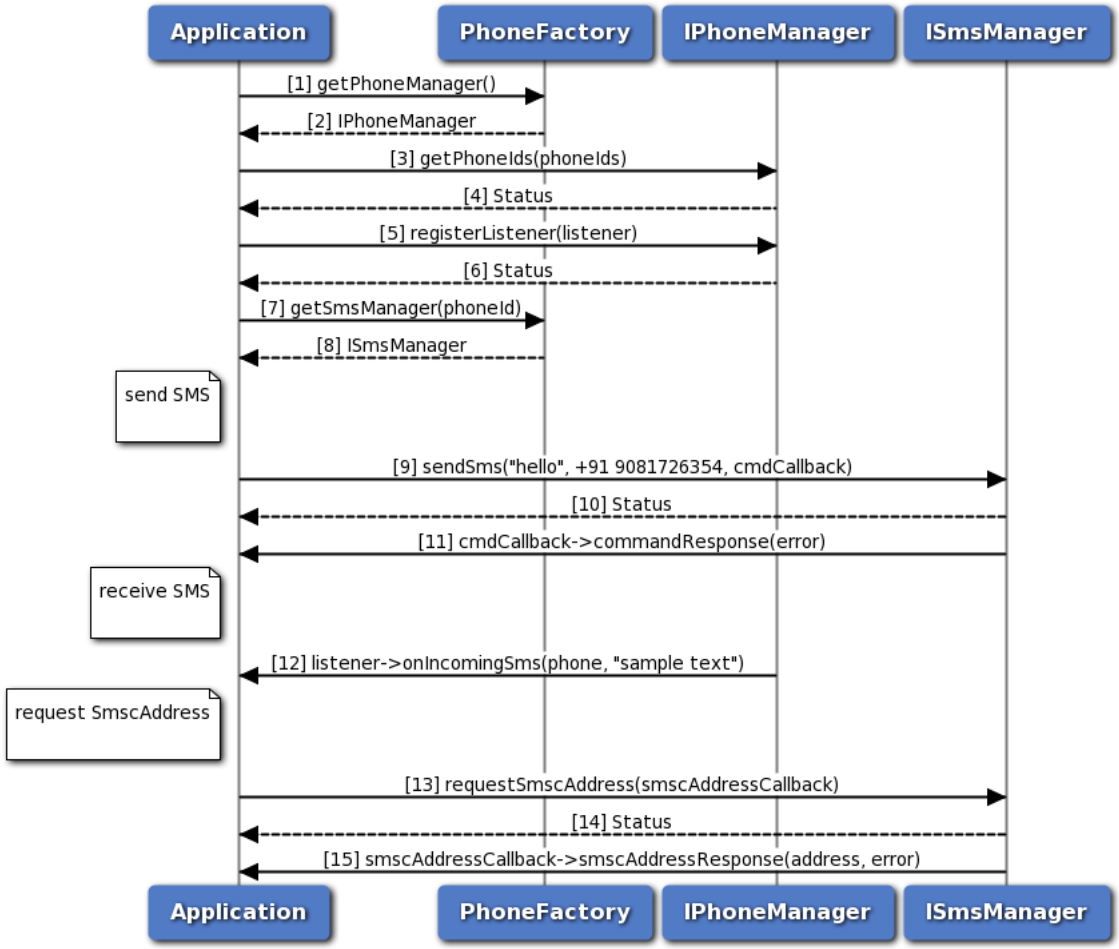

Figure 3-9 SMS call flow

1. Application gets PhoneManager object using PhoneFactory.
2. PhoneFactory returns the PhoneManager object to application in order to get list of phone identifiers.
3. The application retrieves the list of phone identifier on the device using PhoneManager object.
4. Application receives the status i.e. either SUCCESS or FAILED based on the execution of getPhoneIds API in PhoneManager.
5. Application registers the listener for incoming SMS with IPhoneManager.
6. The application receives the status i.e. either SUCCESS or INVALIDPARAM based on successful registration of the listener.
7. The application gets SmsManager object corresponding to specific phone identifier.
8. PhoneFactory returns the SmsManager object to application in order to perform operations like send SMS and get SMSC address.
9. The application sends SMS to the receiver address and optionally gets asynchronous response using CommandResponseCallback.
10. Application receives the status i.e. either SUCCESS or FAILED based on execution of sendSms API

in SmsManager.

11. Optionally, the response for send SMS is received by the application.
12. Application gets notified for incoming SMS.
13. The application requests for SmscAddress and optionally gets asynchronous response using ISmscAddressCallback.
14. Application receives the status i.e. either SUCCESS or FAILED based on successful execution of requestSmscAddress API in SmsManager.
15. Optionally, the application receives the SMSC address on success or gets error on failure in the command response callback.

3.2.9 Radio and Service state call flow

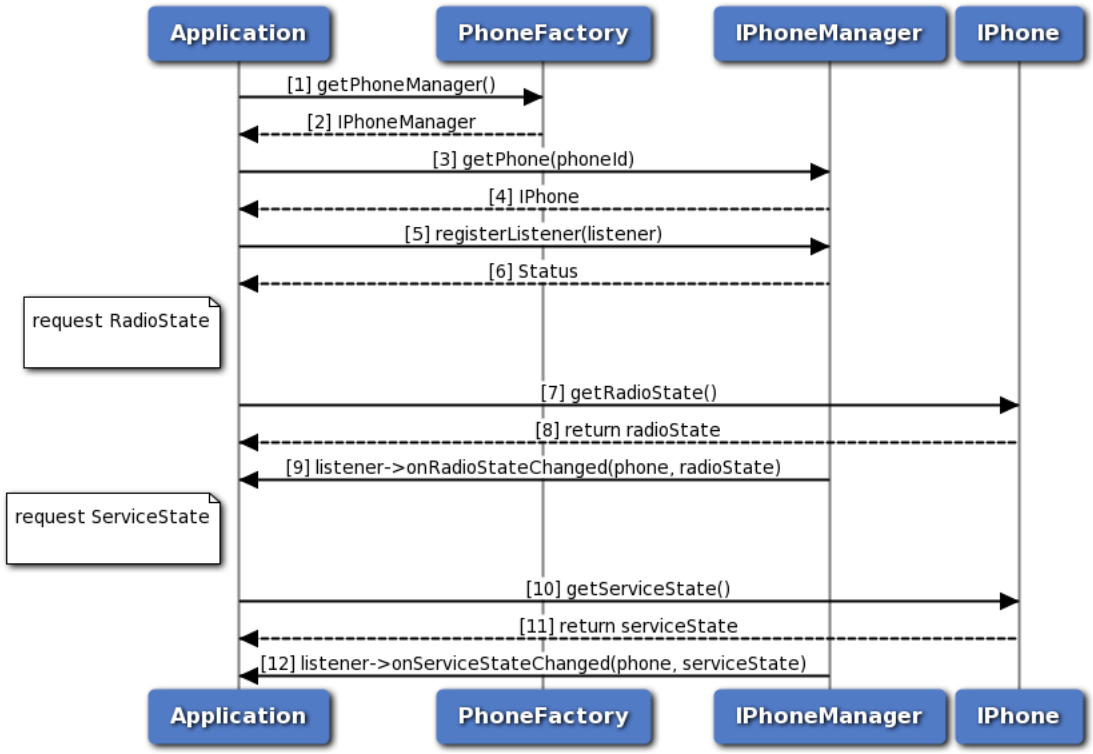

Figure 3-10 Radio and Service state call flow

1. The application gets PhoneManager object using PhoneFactory.
2. The application receives the PhoneManager object in order to get Phone instance.
3. The application gets phone instance for a given phone identifier using PhoneManager object.
4. PhoneManager returns IPhone object to the application.
5. Application registers the listener to get notifications for radio and service state change.
6. The application receives the status i.e. either SUCCESS or FAILED based on the registration of the listener.
7. The application request the Phone to get radio state.

8. The application receives the radio state like RADIO_STATE_ON, OFF or UNAVAILABLE.
9. Application receives a notification when there is a change in radio state.
10. The application request the Phone to get service state.
11. The application receives the service state like IN_SERVICE, OUT_OF_SERVICE, EMERGENCY_ONLY or RADIO_OFF.
12. Application receives a notification when there is a change in service state.

3.2.10 Network Selection Manager call flow

Network selection manager provides APIs to get and set network selection mode, get and set preferred networks and perform network scan for availbale networks. Registered listener will get notified for the change in network selection mode.

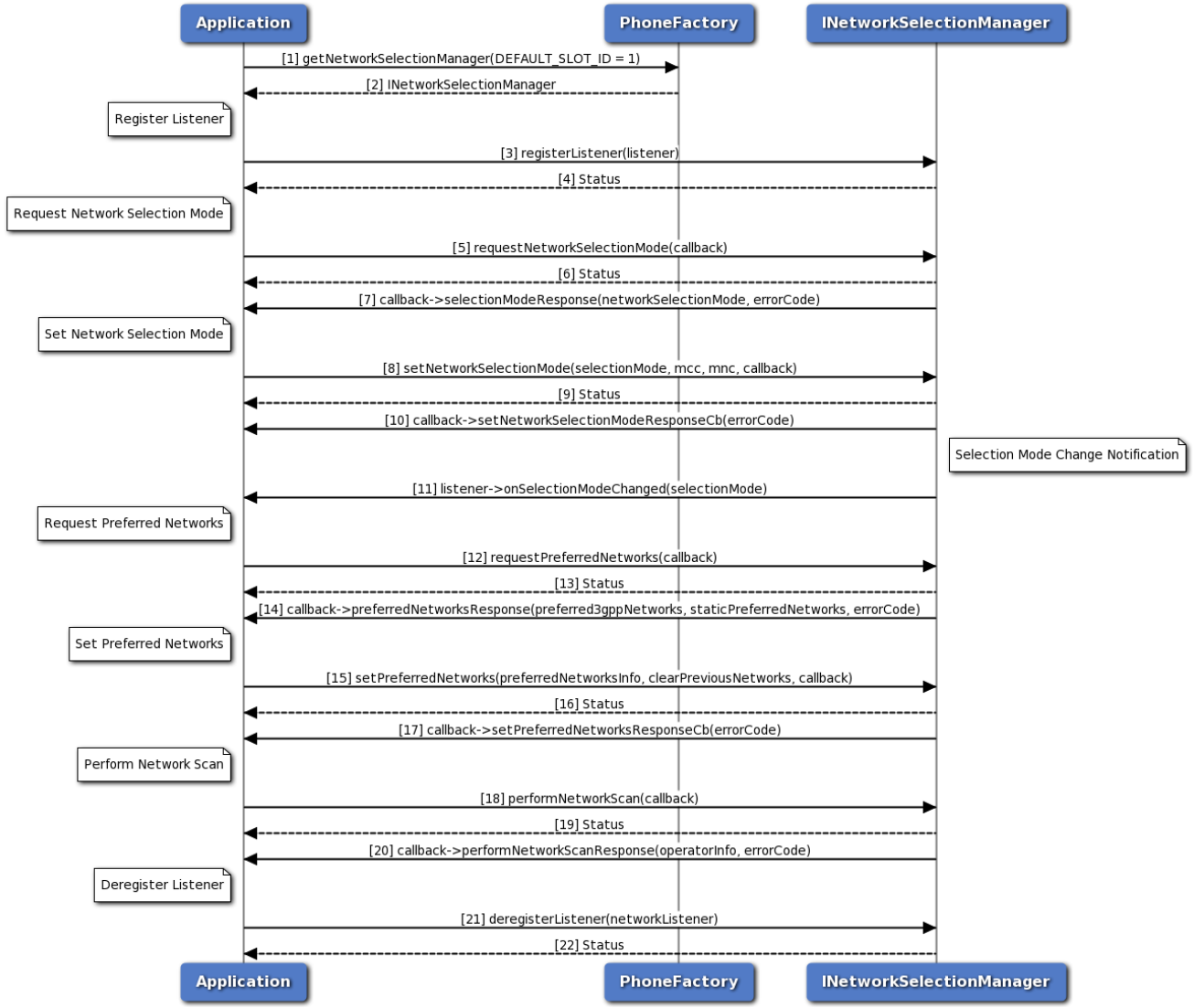

Figure 3-11 Network selection manager call flow

1. Application requests phone factory for network selection manager.
2. Phone factory returns INetworkSelectionManager object using which application will register or deregister a listener.
3. Application can register a listener to get notifications for network selection mode change.
4. Status of register listener i.e. either SUCCESS or other status will be returned to the application.
5. Application requests for network selection mode using INetworkSelectionManager object and gets asynchronous response using SelectionModeResponseCallback.
6. The application receives the status i.e. either SUCCESS or other status based on the execution of requestNetworkSelectionMode API.
7. The response for get network selection mode request is received by the application.
8. The application can also set network selection mode and optionally gets asynchronous response using ResponseCallback. MCC and MNC are optional for AUTOMATIC network selection mode.
9. Application receives the status i.e. either SUCCESS or other status based on the execution of setNetworkSelectionMode API.
10. Optionally the response for set network selection mode request is received by the application.
11. Registered listener will get notified for the network selection mode change.
12. Similarly, the application requests for preferred networks using INetworkSelectionManager object and gets asynchronous response using PreferredNetworksCallback.
13. The application receives the status i.e. either SUCCESS or other status based on the execution of requestPreferredNetworks API.
14. The response for get preferred networks request i.e. 3GPP preferred network list and static 3GPP preferred network list is received by the application asynchronously. Higher priority networks appear first in the list. The networks that appear in the 3GPP Preferred Networks list get higher priority than the networks in the static 3GPP preferred networks list.
15. The application can set 3GPP preferred network list and optionally gets asynchronous response using ResponseCallback. If clear previous networks flag is false then new 3GPP preferred network list is appended to existing preferred network list. If flag is true then old list is flushed and new 3GPP preferred network list is added.
16. Application receives the status i.e. either SUCCESS or other status based on the execution of setPreferredNetworks API.
17. Optionally the response for set preferred networks request is received by the application.
18. The application can perform network scan for available networks using INetworkSelectionManager object and gets asynchronous response using NetworkScanCallback.
19. Application receives the status i.e. either SUCCESS or other status based on the execution of performNetworkScan API.
20. Network name, MCC, MNC and status of the operator will be received by the application.
21. Application can deregister a listener there by it would not get notifications.
22. Status of deregister listener i.e. either SUCCESS or other status will be returned to the application.

3.2.11 Serving System Manager Call Flow

Serving system manager provides APIs to get and set RAT mode preference and get and set service domain preference. Registered listener will get notified for the change in RAT mode and service domain preference change.

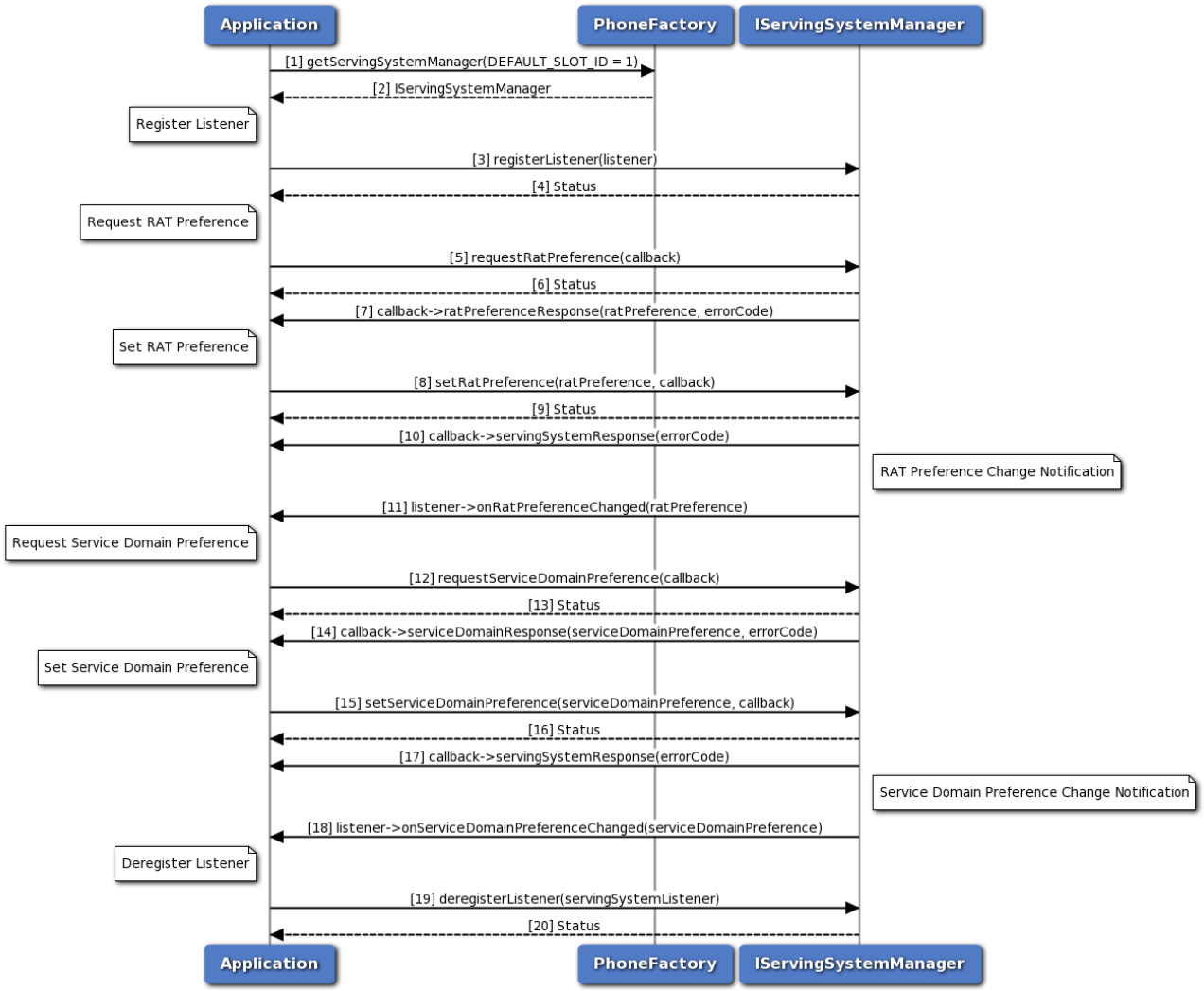

Figure 3-12 Serving System Manager Call Flow

1. Application requests phone factory for serving system manager.
2. Phone factory returns IServingSystemManager object using which application will register or deregister a listener.
3. Application can register a listener to get notifications for RAT mode and service domain preference changes.
4. Status of register listener i.e. either SUCCESS or other status will be returned to the application.
5. Application requests for RAT mode preference using IServingSystemManager object and gets asynchronous response using RatPreferenceCallback.

6. The application receives the status i.e. either SUCCESS or other status based on the execution of requestRatPreference API.
7. The response for get RAT preference request is received by the application.
8. The application can also set RAT mode preference and optionally gets asynchronous response using ResponseCallback.
9. Application receives the status i.e. either SUCCESS or other status based on the execution of setRatPreference API.
10. Optionally the response for set RAT preference request is received by the application.
11. Registered listener will get notified for the RAT mode preference change.
12. Application requests for service domain preference using IServingSystemManager object and gets asynchronous response using ServiceDomainPreferenceCallback.
13. The application receives the status i.e. either SUCCESS or other status based on the execution of requestServiceDomainPreference API.
14. The response for get service domain preference request is received by the application.
15. The application can also set service domain preference and optionally gets asynchronous response using ResponseCallback.
16. Application receives the status i.e. either SUCCESS or other status based on the execution of setServiceDomainPreference API.
17. Optionally the response for set service domain preference request is received by the application.
18. Registered listener will get notified for the service domain preference change.
19. Application can deregister a listener there by it would not get notifications.
20. Status of deregister listener i.e. either SUCCESS or other status will be returned to the application.

3.2.12 Remote SIM Provisioning Call Flow

Remote SIM provisioning provides API to add profile, delete profile, activate/deactivate profile on the embedded SIMs (eUICC) , get list of profiles, get server address like SMDP+ and SMDS and update SMDP+ address, update nick name of profile and retrieve Embedded Identity Document(EID) of the SIM.

3.2.12.1 Download and deletion of profile call flow

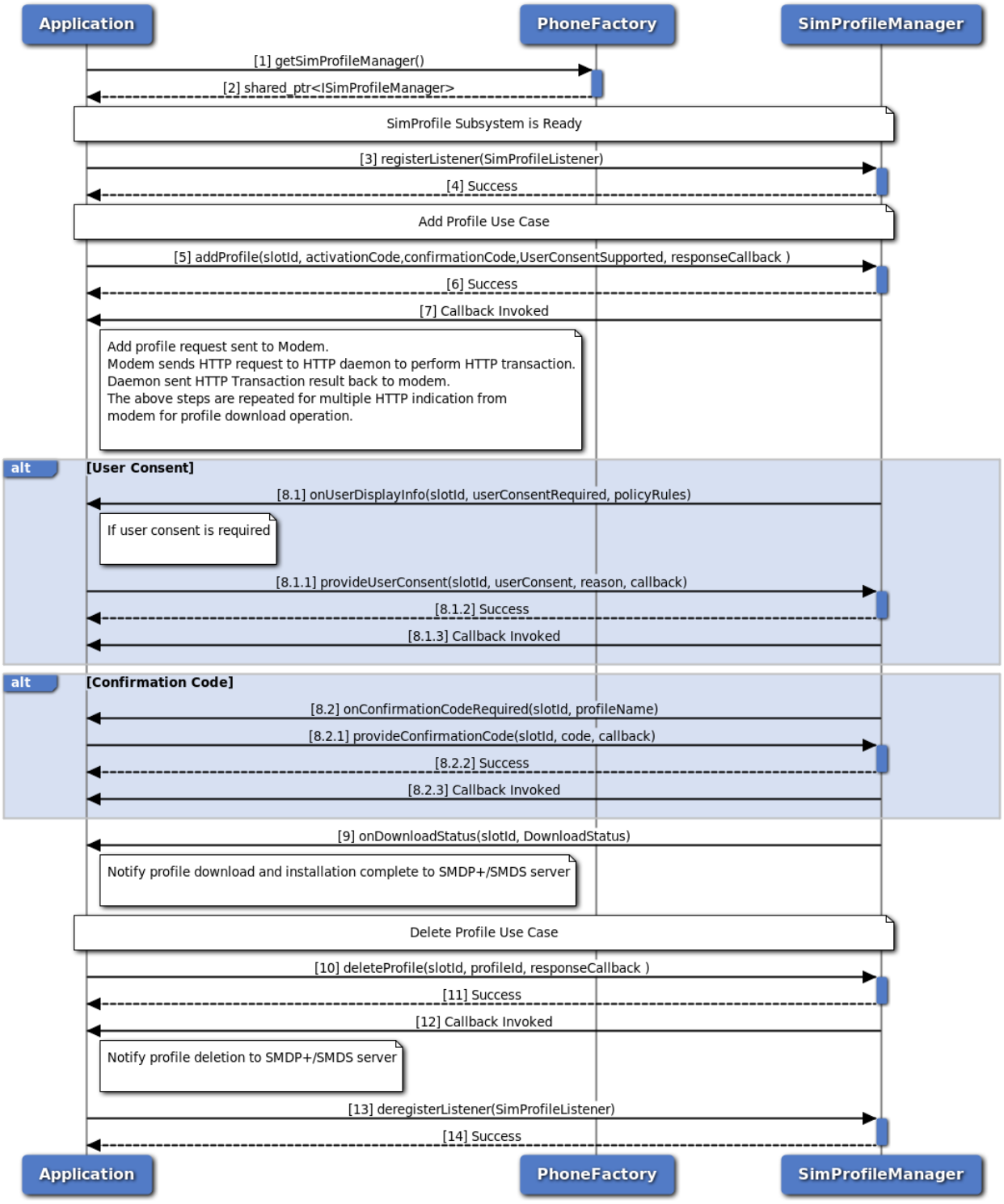

Figure 3-13 Download and deletion of profile call flow

1. Application requests Phonefactory for SimProfileManager object.

2. Phonefactory returns shared pointer to SimProfileManager object to application using which application performs profile related operations. Wait for subsystem to get ready.
3. If subsystem is ready, create a listener of type ISimProfileListener which would receive notifications about profile download status, user display info and confirmation code is required. Register the created listener with the ISimProfileManager object.
4. Status of register listener i.e. either SUCCESS or FAILED will be returned to the application.
5. Application can send a request to add profile with activation code. The confirmation code can be optional and user consent supported can be specified in order to receive user consent info.
6. Application receives synchronous status which indicates if the add profile request was sent successfully.
7. The response of addProfile request can be received by application in the application-supplied callback. The modem sends indication about the HTTP request in order to download profile to AP and the AP performs HTTP transaction on behalf of modem and sends HTTP request to SMDP+/SMDS server and response of HTTP request from SMDP+/SMDS is sent back to AP and then to modem. The HTTP response contains information related to profile.
8. Optionally, the application receives notification about user consent required and profile policy rules. The application needs to provide the user consent and also reason (if in case consent not provided) in order to proceed with downloading of profile. The application receives the synchronous status which indicates provide user consent sent successfully and response of provide user consent can be received by application in the application-supplied callback. Optionally, the application receives notification about confirmation code required. The application needs to provide the code in order to proceed with downloading of profile. The application receives the synchronous status which indicates confirmation code required sent successfully and response of provide user consent can be received by application in the application-supplied callback.
9. The application receives notification about the download and installation status of profile on the eUICC. When download and installation of profile is completed, notification is also sent to SMDP+/SMDS server.
10. Application can send a delete request associated with profile identifier on SimProfileManager object.
11. Application receives synchronous status which indicates if the delete profile request was sent successfully.
12. The response of delete request can be received by application in the application-supplied callback. Once deletion of profile is completed notification is sent to SMDP+/SMDS server in order to synchronise the profile state on the server.
13. De-register the listener with the ISimProfileManager object.
14. Application receives the status i.e. either SUCCESS or FAILED based on the execution of de-register listener.

3.2.12.2 SIM profile management operations call flow

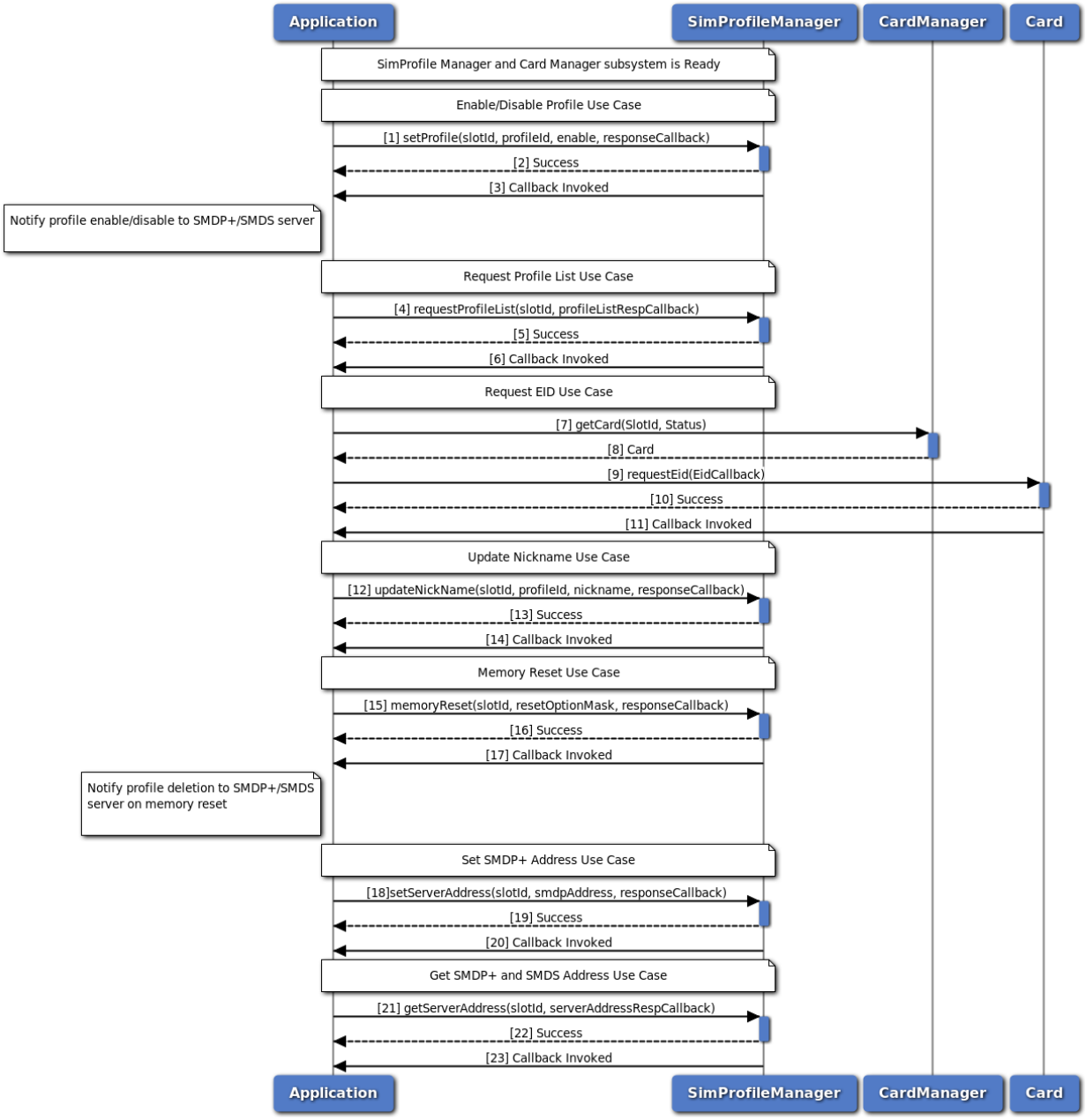

Figure 3-14 SIM profile management operations call flow

The SIM profile sub-system should have been initialized successfully with SERVICE_AVAILABLE as a pre-requisite for remote SIM provisioning operations and a valid SimProfileManager object is available.

1. Application can send a set profile request associated with profile identifier on SimProfileManager object. This allows to enable or disable the profile on the eUICC.
2. Application receives synchronous status which indicates if the set profile request was sent successfully.

3. The response of set profile request can be received by application in the application-supplied callback. Once enable or disable of profile is completed notification is sent to SMDP+/SMDS server in order to synchronise the profile state on the server.
4. Application can send a get profile list request on SimProfileManager object to retrieve the all available profiles on the eUICC with all profile related details like service provider name(SPN), ICCID etc.
5. Application receives synchronous status which indicates if the get profile list request was sent successfully.
6. The response of get profile list request can be received by application in the application-supplied callback along with all the profile details.
7. Application can get Card object from CardManager associated with the slot.
8. CardManager returns shared pointer to Card object to application using which application can retrieve EID.
9. Application can send a get EID request on Card.
10. Application receives synchronous status which indicates if the get EID request was sent successfully.
11. The response of get EID request can be received by application in the application-supplied callback along with EID details.
12. Application can send update nickname of profile request on SimProfileManager.
13. Application receives synchronous status which indicates if the update nickname request was sent successfully.
14. The response of update nickname request can be received by application in the application supplied callback.
15. Application can send memory reset request on SimProfileManager to delete test or operational profiles or set SMDP address to default.
16. Application receives synchronous status which indicates if the memory reset request was sent successfully.
17. The response of memory reset request can be received by application in the application-supplied callback. Once deletion of profiles happens notification is sent to SMDP+/SMDS to synchronise profile state on the server.
18. Application can send set address name on SimProfileManager in order to set SMDP+ address.
19. Application receives synchronous status which indicates if the set address request was sent successfully.
20. The response of set address request can be received by application in the application-supplied callback.
21. Application can send get address name on SimProfileManager in order to get SMDP+/SMDS address.
22. Application receives synchronous status which indicates if the get address request was sent successfully.
23. The response of get address request can be received by application in the application-supplied callback along with SMDP+/SMDS address.

3.2.12.3 HttpTransaction subsystem readiness and handling of indication from modem call flow

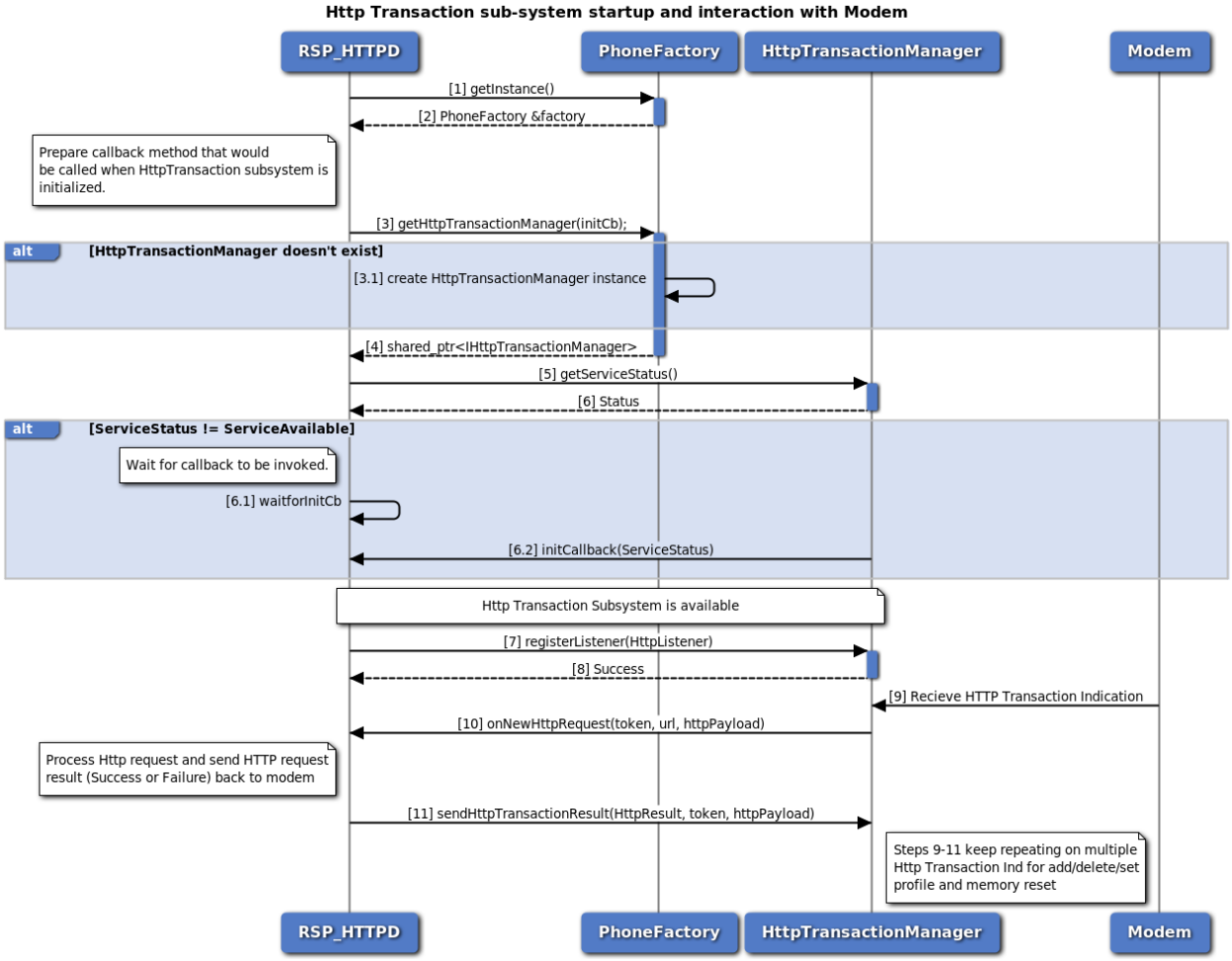

Figure 3-15 HttpTransaction subsystem readiness and handling of indication from modem call flow

1. Get the reference to the PhoneFactory, with which we can further acquire other remote SIM provisioning sub-system objects.
2. The reference daemon gets the Phonefactory object.
3. Reference daemon requests Phonefactory for HttpTransactionManager object along with init callback which gets called once initialization is completed.
4. Phonefactory returns shared pointer to HttpTransactionManager object to reference daemon using which reference daemon sends HTTP response to modem and register the listener to listen for HTTP transaction indication. If HttpTransactionManager does not exists then create new object.
5. Wait for subsystem to be ready and get the Service status.
6. The reference daemon check the service status whether service is UNAVAILABLE, AVAILABLE or FAILED. If the service status is not AVAILABLE, then wait for init callaback to be called. The subsystem either gets AVAILABLE or FAILED and Service status is recieved in init callback. If the

service status is notified as SERVICE_FAILED, retry intitialization starting with step (3).

7. If subsystem is AVAILABLE , then create a listener of type IHttpListener which would receive notifications about HTTP transaction indication. Register the created listener with the IHttpTransactionManager object.
8. Status of register listener i.e. either SUCCESS or FAILED will be returned to the reference daemon.
9. Modem sends the HTTP transaction indication with HTTP request payload and other details to HTTPTransactionManager.
10. The reference daemon on AP receives notification about HTTP transaction which process HTTP request.
11. The reference daemon sends HTTP request result (SUCCESS or FAILURE) with HTTP response back to HTTPTransactionManager.
12. Steps (9-11) keep repeating for multiple HTTP Transaction indication for add, delete, set profile and memory reset operations.

3.3 Card Services

3.3.1 Get applications call flow

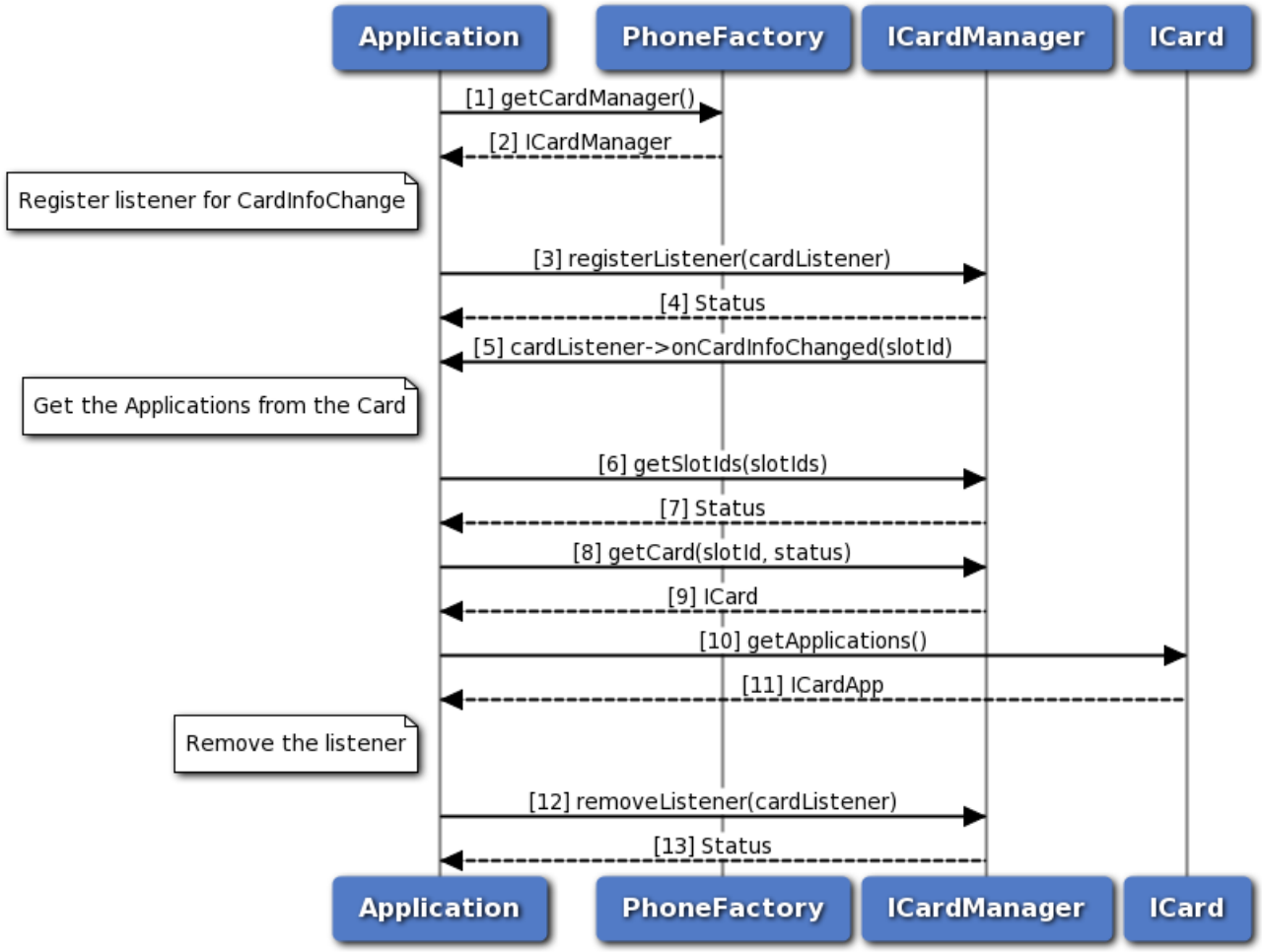

Figure 3-16 Get applications call flow

1. Application gets the CardManager object from PhoneFactory.
2. Application receives CardManager object in order to perform operations like getSlotIds and getCard.
3. The application registers a listener for Card info change event with CardManager.
4. Application receives the status i.e. either SUCCESS or INVALIDPARAM based on the registration of listener.
5. The response from onCardInfoChanged is received by the application whenever there is card info change.
6. The application gets the slotIds from the sub-system using CardManager.
7. Application receives the status i.e. either SUCCESS or NOTREADY along with the updated slotIds.
8. Then, the application sends request to CardManager to get Card object for a specific slotId.
9. Application receives Card object from CardManager in order to perform card operation like getApplications.

10. The application gets the CardApps from Card object.
11. The application receives CardApps which contain information such as AppId, AppType and AppState.
12. Now the application removes the listener associated with CardManager.
13. Application receives the status i.e. either SUCCESS or NOSUCH for the removal of listener.

3.3.2 Transmit APDU call flow

3.3.2.1 On logical channel

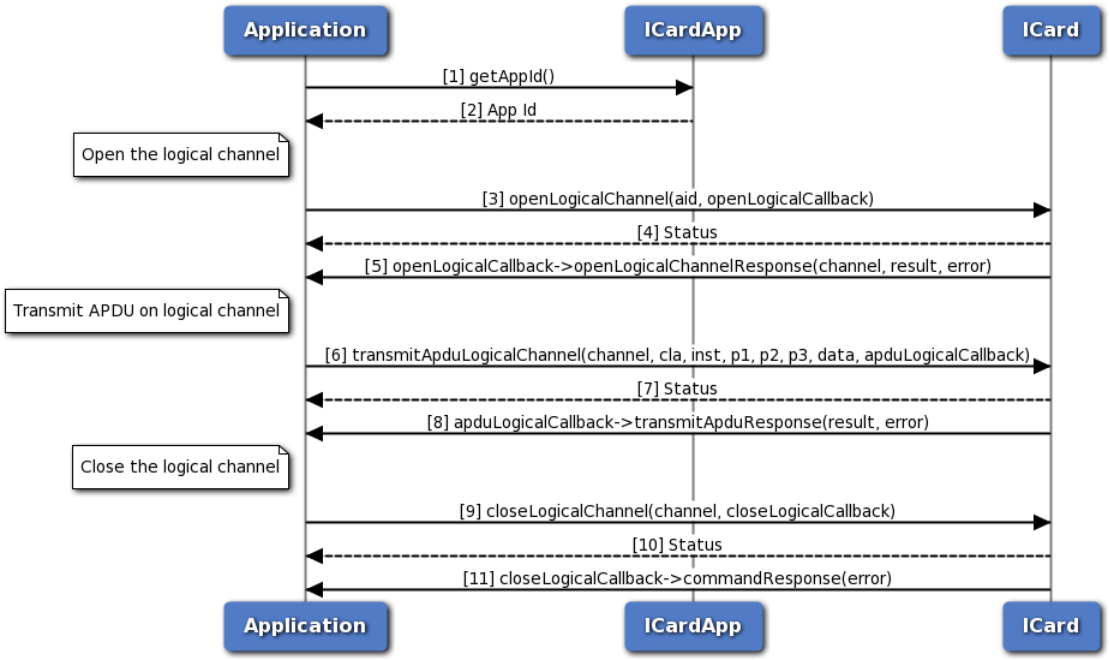

Figure 3-17 On logical channel

1. The Application requests CardApp for the SIM application identifier.
2. The application receives the application identifier to perform open logical channel.
3. Application sends request to open the logical channel with the application identifier and optionally, gets asynchronous response in OpenLogicalChannelCallback.
4. The application receives the status i.e. either SUCCESS or FAILED based on execution of openLogicalChannel API.
5. Optionally, the application receives the response which contains either channel number on success or error in case of failure.
6. Then, the application transmits the APDU data on logical channel using the channel obtained earlier. Optionally, gets asynchronous response in TransmitApduResponseCallback.
7. The application receives the status i.e. either SUCCESS or FAILED based on execution of transmitApduLogicalChannel API.
8. Optionally, the application receives the response which contains either result on success or error in

case of failure.

9. Finally, the application closes the logical channel that is opened to transmit APDU and optionally, gets asynchronous response in CommandResponseCallback.
10. The application receives the status i.e. either SUCCESS or FAILED based on execution of closeLogicalChannel API.
11. Optionally, the application receives the response which contains error in case of failure.

3.3.2.2 On basic channel

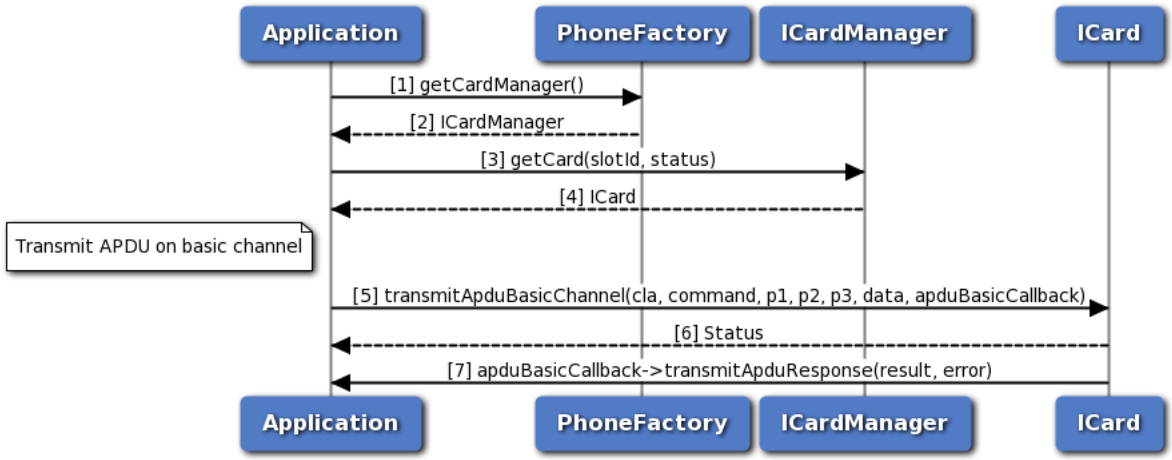

Figure 3-18 On basic channel

1. Application gets the ICardManager object from PhoneFactory.
2. Application receives ICardManager object in order to perform operation like getCard.
3. The application gets ICard object for a specific slotId from ICardManager.
4. Application receives ICard object in order to perform card operation like transmitApduBasicChannel.
5. The application transmits APDU data on basic channel and optionally, gets asynchronous response in TransmitApduResponseCallback.
6. The application receives the status i.e. either SUCCESS or FAILED based on execution of transmitApduBasicChannel API.
7. Optionally, the application receives the response which contains either result on success or error in case of failure.

3.3.3 SAP card manager call flow

3.3.3.1 Request card reader status, Request ATR, Transmit APDU call flow

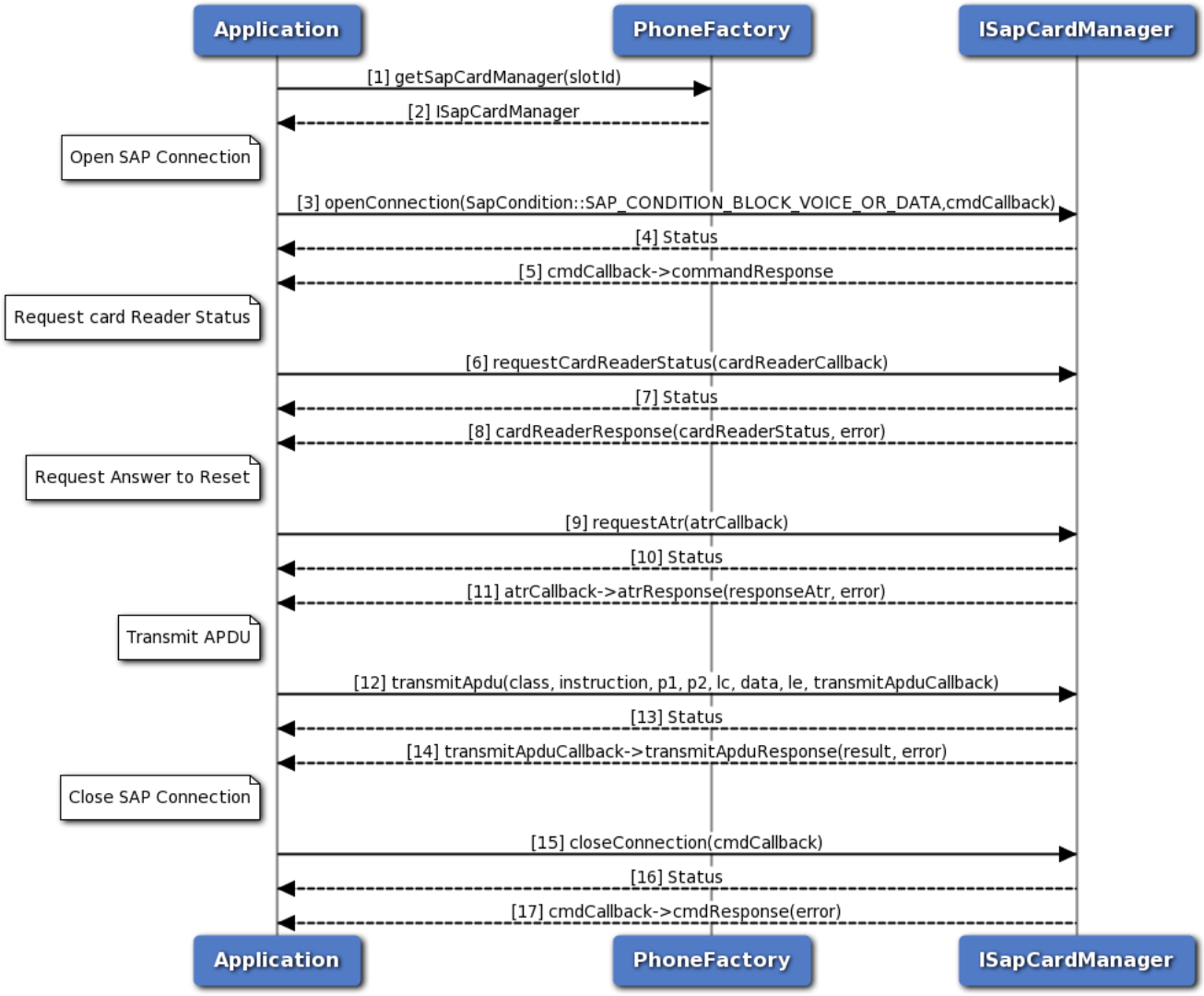

Figure 3-19 Request card reader status, Request ATR, Transmit APDU call flow

{sap_card_operations.png,Request card reader status, Request ATR, Transmit APDU call flow,80,80,Request card reader status, Request ATR, Transmit APDU call flow}

1. The application gets SapCardManager object corresponding to slotId using PhoneFactory.
2. The application receives the SapCardManager object in order to perform SAP operations like request ATR, Card Reader Status and transmit APDU.
3. The application opens SIM Access Profile(SAP) connection with SIM card using default SAP condition (i.e. SAP_CONDITION_BLOCK_VOICE_OR_DATA) and optionally, gets asynchronous response using CommandResponseCallback.
4. The application receives the status i.e. either SUCCESS or FAILED based on execution of openConnection API in SapCardManager.
5. Optionally, the response for openConnection is received by the application.

6. The application sends request card reader status command and optionally, gets asynchronous response using ICardReaderCallback.
7. The application receives the status i.e. either SUCCESS or FAILED based on execution of requestCardReaderStatus API in SapCardManager.
8. Optionally, the response for card reader status is received by the application.
9. Similarly, the application can send SAP Answer To Reset command and optionally, gets asynchronous response using IAtrResponseCallback.
10. The application receives the status i.e. either SUCCESS or FAILED based on execution of requestAtr API in SapCardManager.
11. Optionally, the response for SAP Answer To Reset is received by the application.
12. Similarly, the application sends the APDU on SAP mode and optionally, gets asynchronous response using ISapTransmitApduResponseCallback.
13. The application receives the status i.e. either SUCCESS or FAILED based on execution of transmitApdu API in SapCardManager.
14. Optionally, the response for transmit APDU is received by the application.
15. Now the application closes the SAP connection with SIM and optionally, gets asynchronous response using CommandResponseCallback.
16. The application receives the status i.e. either SUCCESS or FAILED based on execution of closeConnection API in SapCardManager.
17. Optionally, the response for SAP close connection is received by the application.

3.3.3.2 SIM Turn off, Turn on and Reset call flow

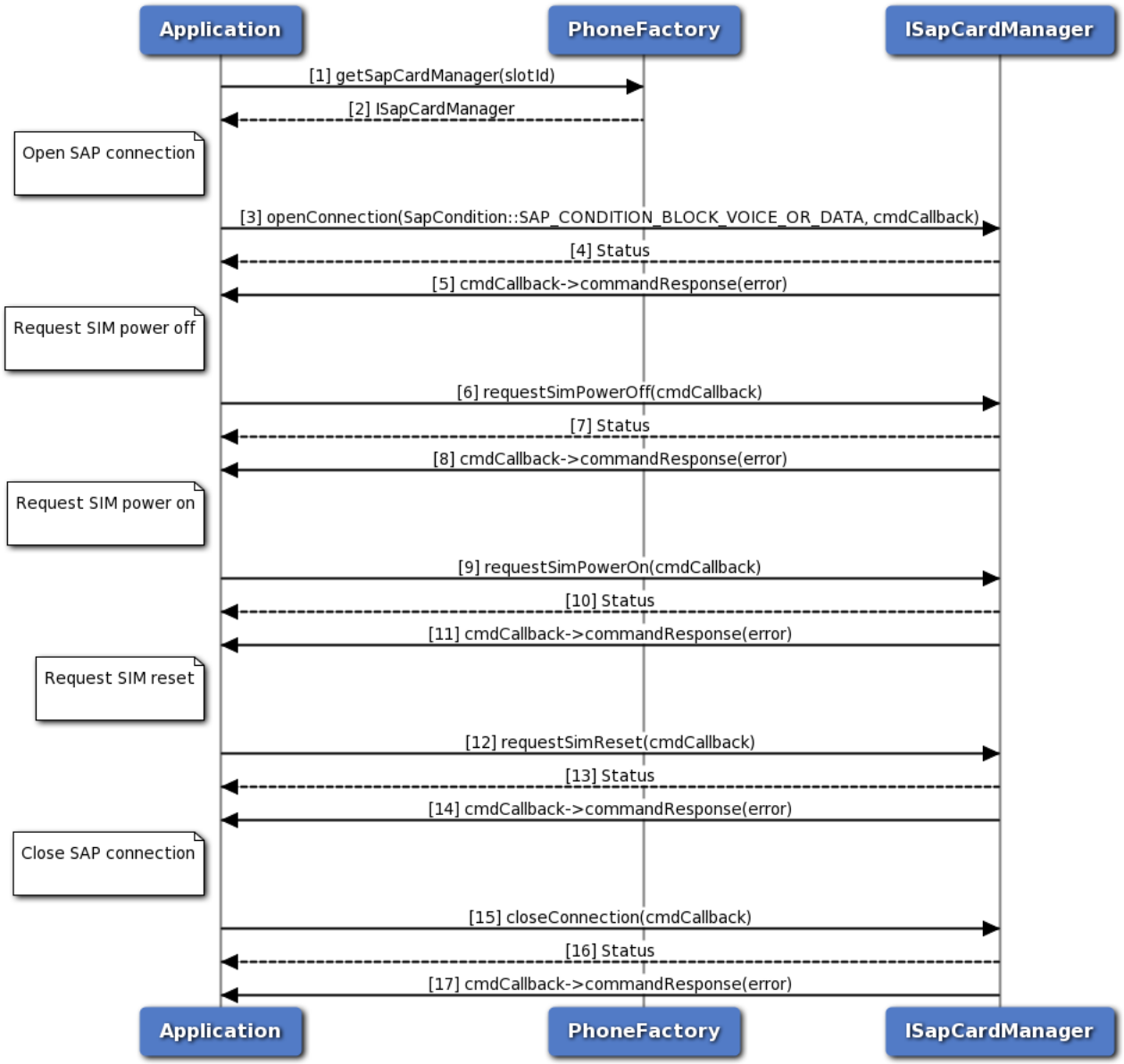

Figure 3-20 SIM Turn off, Turn on and Reset call flow

{sap_mgr_sim_call_flow.png,SIM Turn off, Turn on and Reset call flow,80,80,SIM Turn off, Turn on and Reset call flow}

1. Application gets SapCardManager object corresponding to slotID using PhoneFactory.
2. PhoneFactory returns the SapCardManager object to application in order to perform SAP operations like SIM power off, on or reset.
3. The application opens SIM Access Profile(SAP) connection with SIM card using default SAP condition (i.e. SAP_CONDITION_BLOCK_VOICE_OR_DATA) and optionally, gets asynchronous response using CommandResponseCallback.

4. Application receives the status i.e. either SUCCESS or FAILED based on execution of openConnection API in SapCardManager.
5. Optionally, the response for openConnection is received by the application.
6. The application sends SIM Power Off command to turn off the SIM and optionally, gets asynchronous response using CommandResponseCallback.
7. The application receives the status i.e. either SUCCESS or FAILED based on execution of requestSimPowerOff API in SapCardManager.
8. Optionally, the response for SIM Power Off is received by the application.
9. Similarly, the application can send SIM Power On command to turn on the SIM and optionally, gets asynchronous response using CommandResponseCallback.
10. The application receives the status i.e. either SUCCESS or FAILED based on execution of requestSimPowerOn API in SapCardManager.
11. Optionally, the response for SIM Power On is received by the application.
12. Similarly, the application sends SIM Reset command to perform SIM Reset and optionally, gets asynchronous response.
13. The application receives the status i.e. either SUCCESS or FAILED based on execution of requestSimReset API in SapCardManager.
14. Optionally, the response for SIM Reset is received by the application.
15. Now the application closes the SAP connection with SIM and optionally, gets asynchronous response using CommandResponseCallback.
16. The application receives the status i.e. either SUCCESS or FAILED based on execution of closeConnection API in SapCardManager.
17. Optionally, the response for SAP close connection is received by the application.

3.3.4 Subscription Call flow

3.3.4.1 Subscription initialization

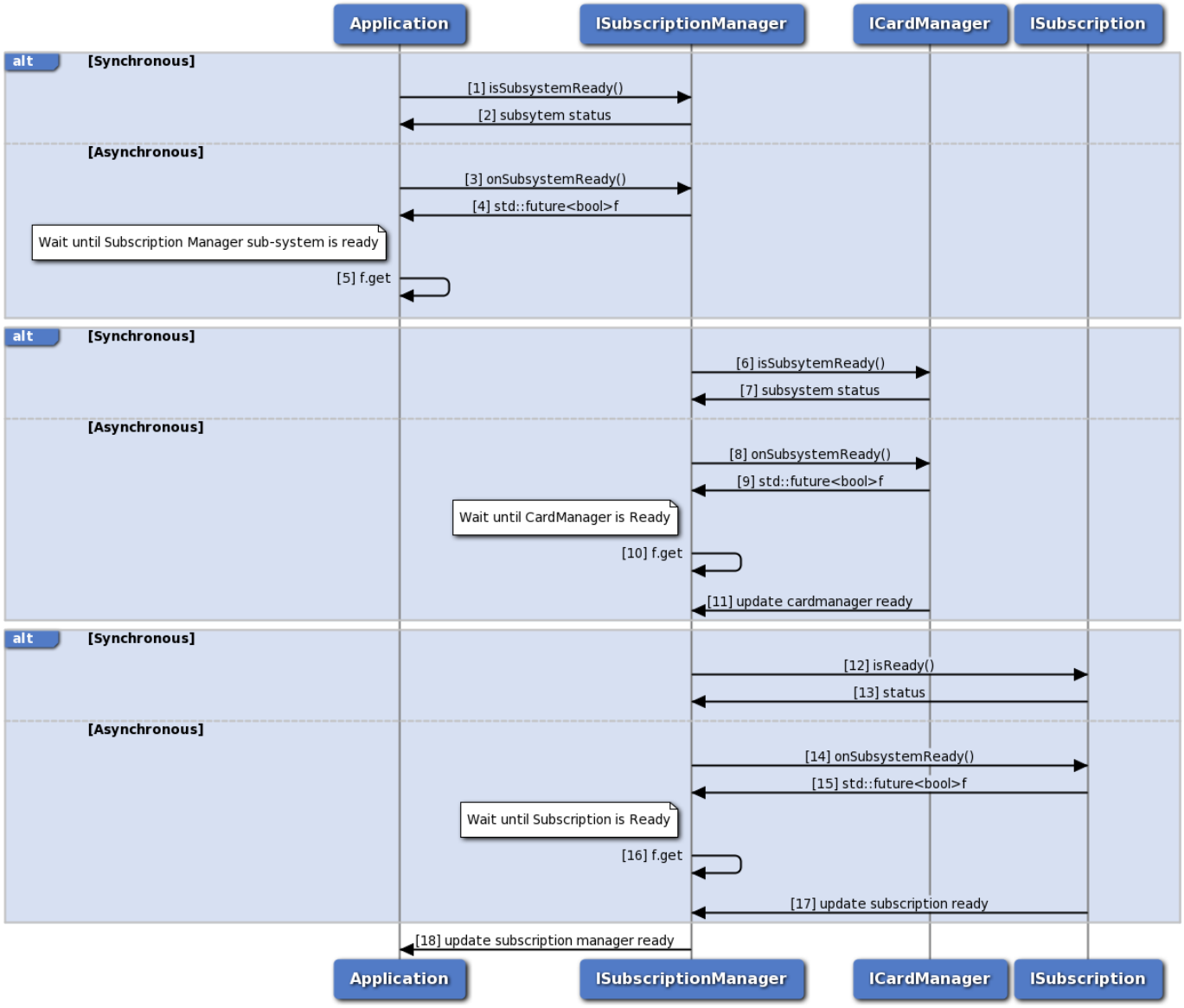

Figure 3-21 Subscription initialization call flow

1. Application can use ISubscriptionManager::isSubsystemReady to determine if the SubscriptionManager is ready.
2. The application receives the status i.e either true or false whether sub-system is ready or not.
3. If it is not ready, then the application could use onSubsystemReady which returns std::future.
4. SubscriptionManager notifies the application when the subsystem is ready through the std::future object.
5. The application waits until the asynchronous operation i.e onSubsystemReady completes.
6. SubscriptionManager uses ICardManager::isSubsystemReady to determine if the CardManager is ready.

7. The SubscriptionManager receives the status i.e either true or false whether sub-system is ready or not.
8. If it is not ready, then the SubscriptionManager could use onSubsystemReady which returns std::future.
9. CardManager notifies the SubscriptionManager when the subsystem is ready through the std::future object.
10. The SubscriptionManager waits until the asynchronous operation i.e onSubsystemReady completes.
11. CardManager updates the SubscriptionManager once the card manager sub-system is ready
12. SubscriptionManager uses ISubscription::isReady to determine if the Subscription is ready.
13. The SubscriptionManager receives the status i.e either true or false whether sub-system is ready or not.
14. If it is not ready, then the SubscriptionManager could use onSubsystemReady which returns std::future.
15. Subscription notifies the SubscriptionManager when the subsystem is ready through the std::future object.
16. The SubscriptionManager waits until the asynchronous operation i.e onSubsystemReady completes.
17. Subscription updates the SubscriptionManager when the subscription is ready.
18. SubscriptionManager updates the application once subscription manager initialization completes.

3.3.4.2 Subscription call flow

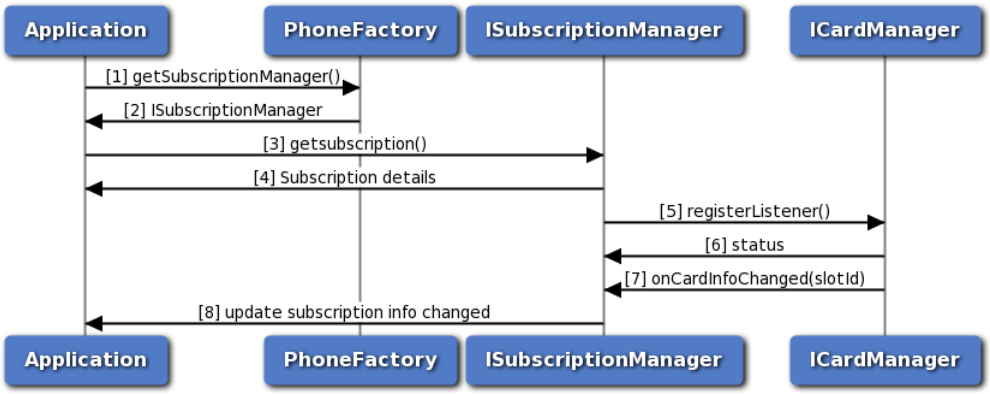

Figure 3-22 Subscription call flow

1. The application gets the PhoneManager object using PhoneFactory.
2. The application receives the PhoneManager object in order to get Subscription.
3. The application gets the Subscription object for given slot identifier using SubscriptionManager.
4. SubscriptionManager returns Subscription object to application. Subscription can be used to get subscription details like countryISO, operator details etc.
5. The Subscription manager registers a listener with CardManager to listen to the card info change notifications like card state PRESENT, ABSENT, UNKNOWN, ERROR and RESTRICTED.

6. The SubscriptionManager receives the status like SUCCESS or INVALIDPARAM based on registration of listener to CardManager.
7. The SubscriptionManager receives callback card info change i.e subscription info changed or removed.
8. The SubscriptionManager updates the application once the subscription info is updated.

3.4 Call flow for location services

Application will get the location manager object from location factory. The caller needs to register a listener. Application would then need to start the reports using one of 2 APIs depending on if the detailed or basic reports are needed. When reports are no longer required, the app needs to stop the report and de-register the listener.

NOTE: Applications need to have "locclient" Linux group permissions to be able to operate successfully with underlying services.

3.4.1 Call flow to register/remove listener for generating basic reports

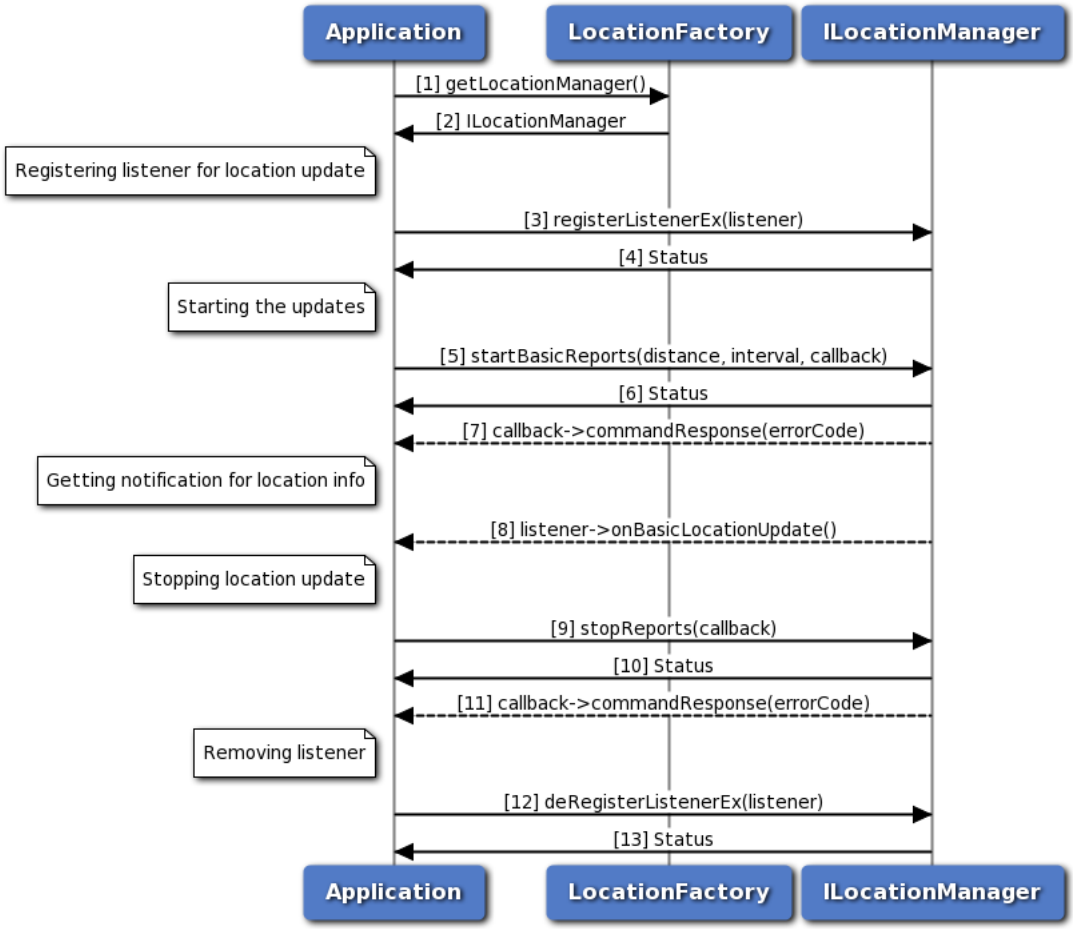

Figure 3-23 Call flow to register/remove listener for generating basic reports

1. Application requests location factory for location manager object.

2. Location factory returns ILocationManager object using which application will register or remove a listener.
3. Application can register a listener for getting notifications for location updates.
4. Status of register listener i.e. either SUCCESS or FAILED will be returned to the application.
5. Application starts the basic reports using startBasicReports API for getting location updates.
6. Status of startBasicReports i.e. either SUCCESS or FAILED will be returned to the application.
7. The response for startBasicReports is received by the application.
8. Application will get location updates like latitude, longitude and altitude etc.
9. Application stops receiving the report through stopReports API.
10. Status of stopReports i.e. either SUCCESS or FAILED will be returned to the application.
11. The response for stopReports is received by the application.
12. Application can remove listener and when the number of listeners are zero then location service will get stopped automatically.
13. Status of remove listener i.e. either SUCCESS or FAILED will be returned to the application.

3.4.2 Call flow to register/remove listener for generating detailed reports

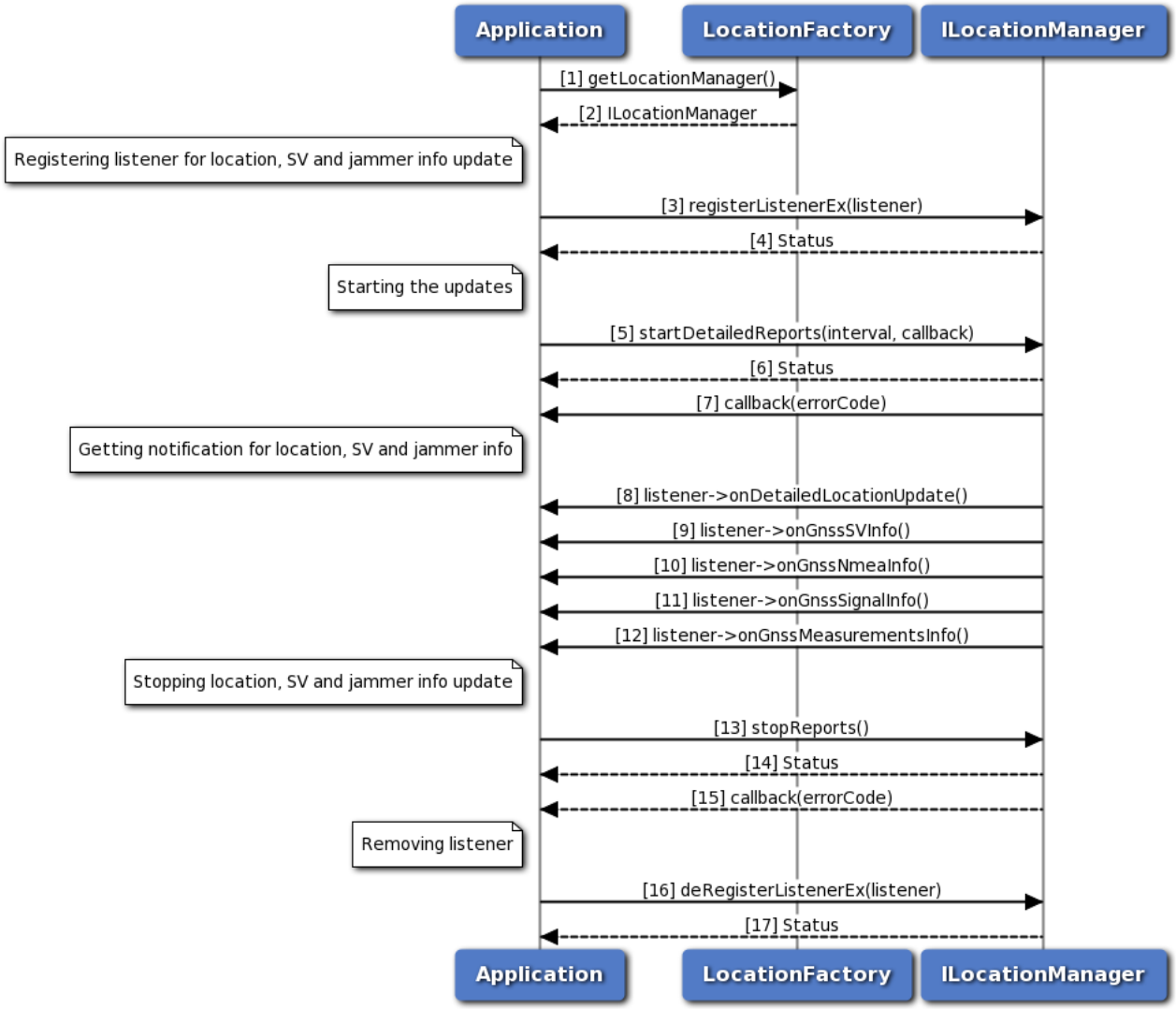

Figure 3-24 Call flow to register/remove listener for generating detailed reports

1. Application requests location factory for location manager object.
2. Location factory returns ILocationManager object using which application will register or remove a listener.
3. Application can register a listener for getting notifications for location, satellite vehicle, jammer signal, nmea and measurements updates.
4. Status of register listener i.e. either SUCCESS or FAILED will be returned to the application.
5. Application starts the detailed reports using startDetailedReports API for getting location, satellite vehicle, jammer signal, nmea and measurements updates.
6. Status of startDetailedReports i.e. either SUCCESS or FAILED will be returned to the application.
7. The response for startDetailedReports is received by the application.

8. Application will get location updates like latitude, longitude and altitude etc.
9. Application will receive satellite vehicle information like SV status and constellation etc.
10. Application will receive nmea information.
11. Application will receive jammer information etc.
12. Application will receive measurement information.
13. Application stops receiving all the reports through stopReports API.
14. Status of stopReports i.e. either SUCCESS or FAILED will be returned to the application.
15. The response for stopReports is received by the application.
16. Application can remove listener and when the number of listeners are zero then location service will get stopped automatically.
17. Status of remove listener i.e. either SUCCESS or FAILED will be returned to the application.

3.4.3 Call flow to register/remove listener for generating detailed engine reports

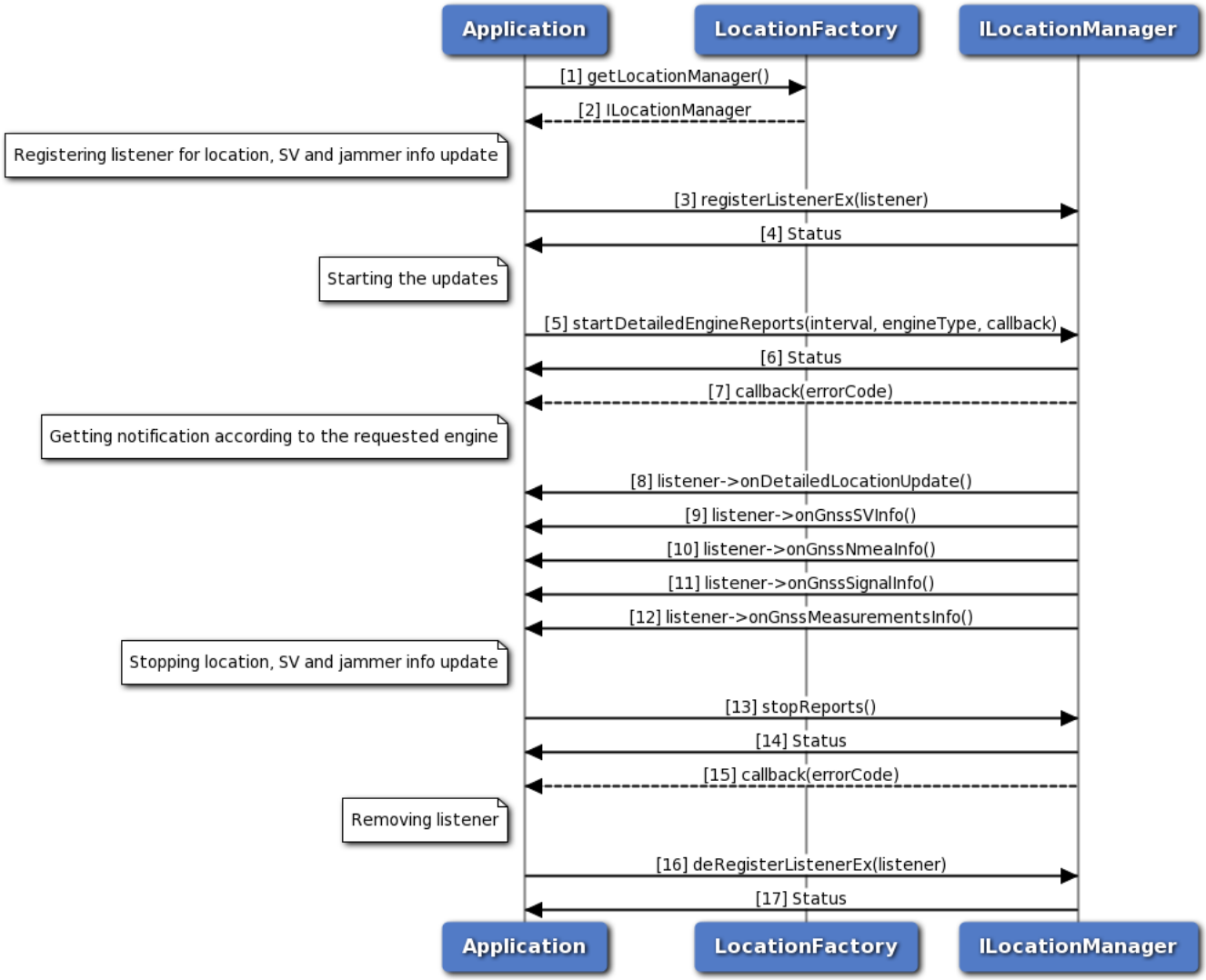

Figure 3-25 Call flow to register/remove listener for generating detailed engine reports

1. Application requests location factory for location manager object.
2. Location factory returns ILocationManager object using which application will register or remove a listener.
3. Application can register a listener for getting notifications for location, satellite vehicle and jammer signal, nmea and measurements updates.
4. Status of register listener i.e. either SUCCESS or FAILED will be returned to the application.
5. Application starts the detailed engine reports using startDetailedEngineReports API for getting location, satellite vehicle, jammer signal, nmea and measurements updates.
6. Status of startDetailedReports i.e. either SUCCESS or FAILED will be returned to the application.

7. The response for startDetailedReports is received by the application.
8. Application will get location updates like latitude, longitude and altitude etc from the requested engine type(SPE/PPE/Fused).
9. Application will receive satellite vehicle information like SV status and constellation etc depending on the requested SPE/PPE/Fused engine type.
10. Application will receive nmea information depending on the requested SPE/PPE/Fused engine type
11. Application will receive jammer information etc depending on the requested SPE/PPE/Fused engine type.
12. Application will receive measurement information etc depending on the requested SPE/PPE/Fused engine type.
13. Application stops receiving all the reports through stopReports API.
14. Status of stopReports i.e. either SUCCESS or FAILED will be returned to the application.
15. The response for stopReports is received by the application.
16. Application can remove listener and when the number of listeners are zero then location service will be stopped automatically.
17. Status of remove listener i.e. either SUCCESS or FAILED will be returned to the application.

3.4.4 Call flow to register/remove listener for system info updates

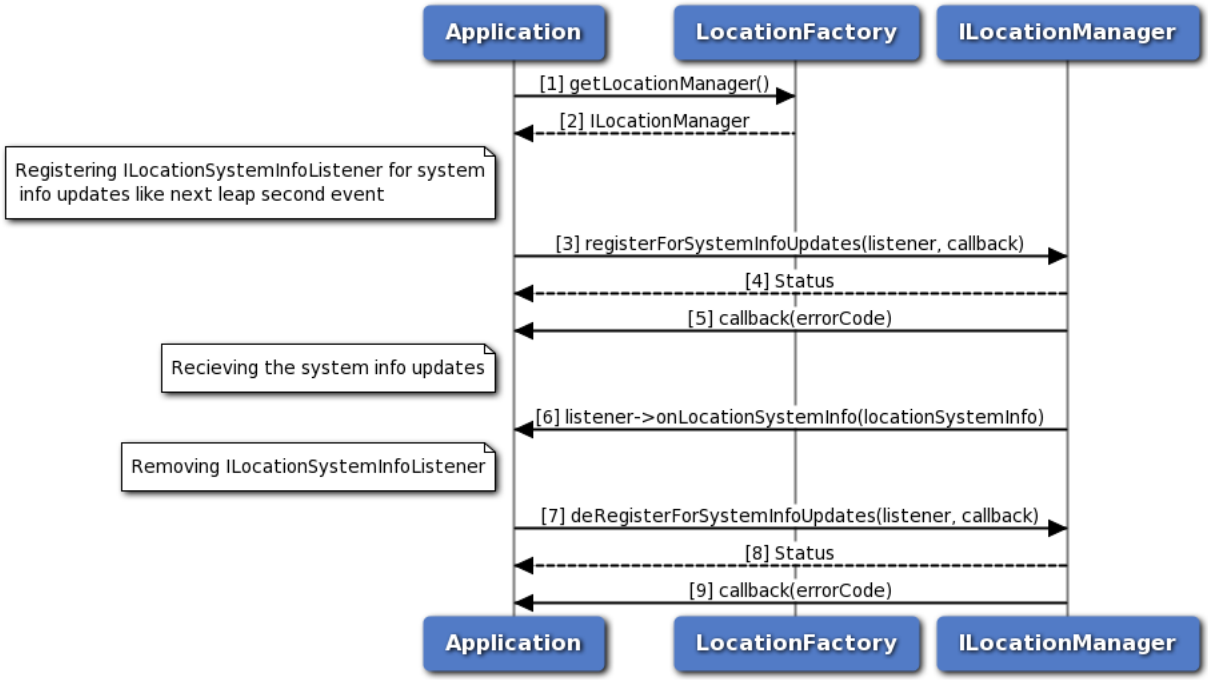

Figure 3-26 Call flow to register/remove listener for system info updates

1. Application requests location factory for location manager object.
2. Location factory returns ILocationManager object using which application will register or remove a listener.

3. Application can register a listener for system information updates with registerForSystemInfoUpdates.
4. Status of registerForSystemInfoUpdates i.e. either SUCCESS or FAILED will be returned to the application.
5. The response for registerForSystemInfoUpdates is received by the application.
6. Application will get system information update.
7. Application can remove listener with deRegisterForSystemInfoUpdates.
8. Status of deRegisterForSystemInfoUpdates i.e. either SUCCESS or FAILED will be returned to the application.
9. The response for deRegisterForSystemInfoUpdates is received by the application.

3.4.5 Call flow to request energy consumed information

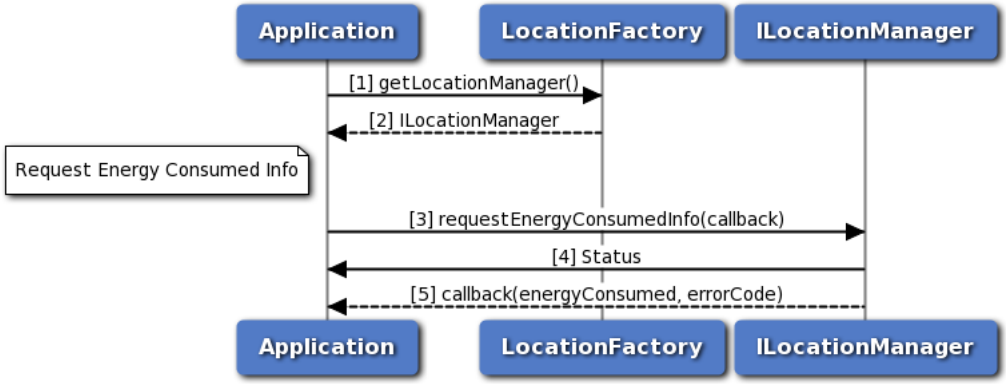

Figure 3-27 Call flow to request energy consumed information

1. Application requests location factory for location manager object.
2. Location factory returns ILocationManager object.
3. Application can request for energy consumed information with requestEnergyConsumedInfo.
4. Status of requestEnergyConsumedInfo i.e. either SUCCESS or FAILED will be returned to the application.
5. The response for requestEnergyConsumedInfo is received by the application.

3.4.6 Call flow to get year of hardware information

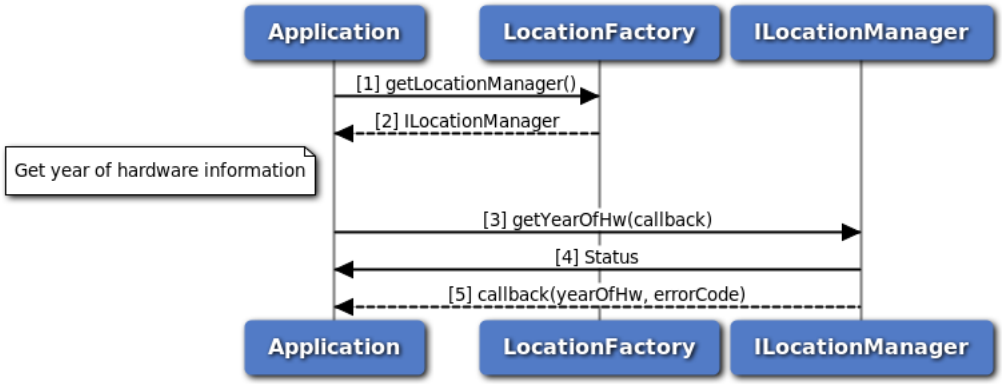

Figure 3-28 Call flow to get year of hardware information

1. Application requests location factory for location manager object.
2. Location factory returns ILocationManager object.
3. Application can request for year of hardware information with getYearOfHw.
4. Status of getYearOfHw i.e. either SUCCESS or FAILED will be returned to the application.
5. The response for getYearOfHw is received by the application.

3.4.7 Call flow to get terrestrial positioning information

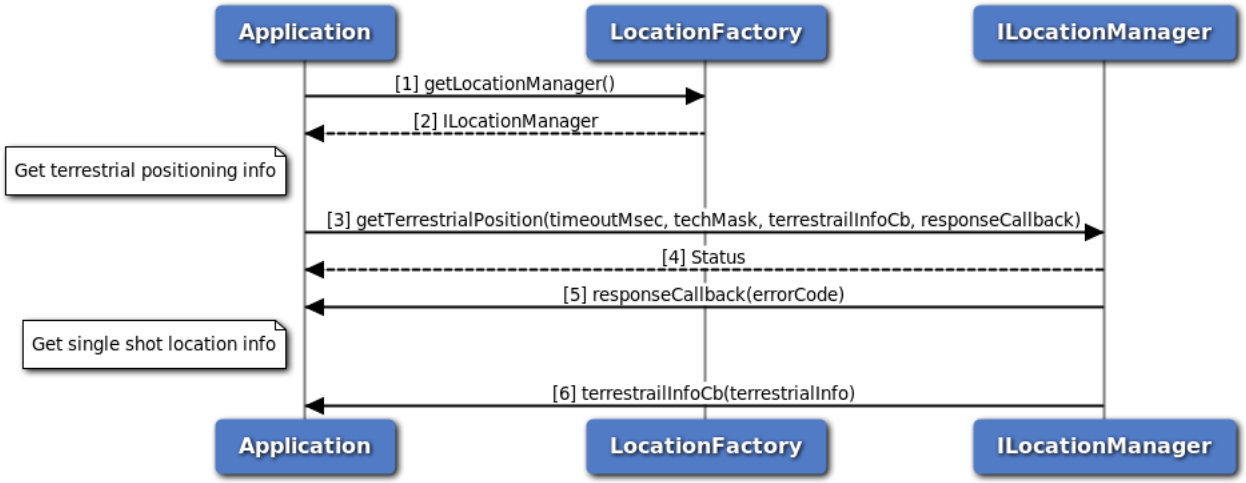

Figure 3-29 Call flow to get terrestrial positioning information

1. Application requests location factory for location manager object.
2. Location factory returns ILocationManager object.
3. Application can request for terrestrial positioning information with getTerrestrialPosition API.

4. Status of getTerrestrialPosition i.e. either SUCCESS or FAILED will be returned to the application.
5. The response for getTerrestrialPosition is received by the application.
6. Single shot terrestrial position information is received by the application.

3.4.8 Call flow to cancel terrestrial positioning information

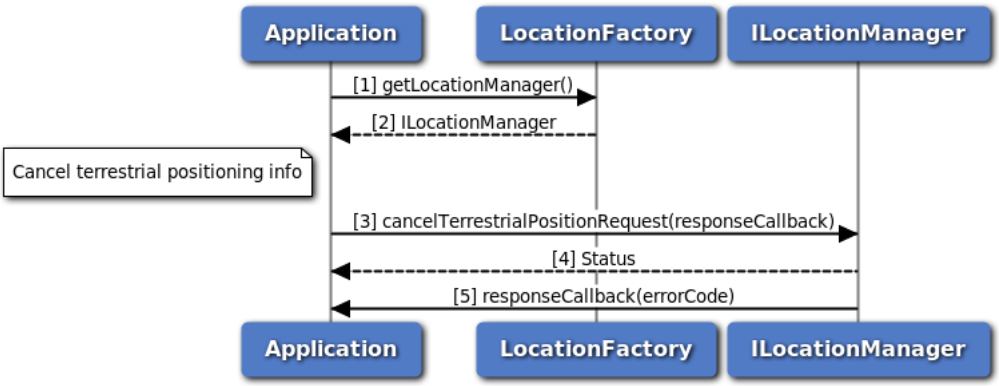

Figure 3-30 Call flow to cancel terrestrial positioning information

1. Application requests location factory for location manager object.
2. Location factory returns ILocationManager object.
3. Application can cancel request for terrestrial positioning information with cancelTerrestrialPositionRequest API.
4. Status of cancelTerrestrialPositionRequest i.e. either SUCCESS or FAILED will be returned to the application.
5. The response for cancelTerrestrialPositionRequest is received by the application.

3.4.9 Call flow to enable/disable constraint time uncertainty

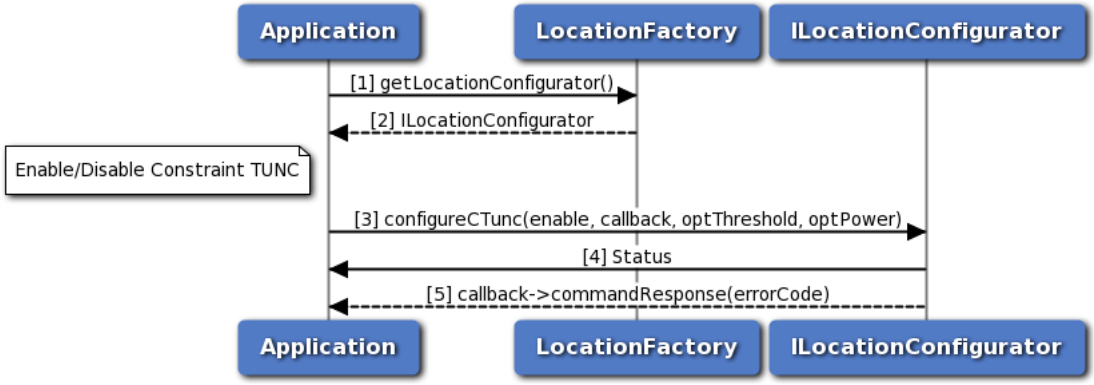

Figure 3-31 Call flow to enable/disable constraint time uncertainty

1. Application requests location factory for location configurator object.

2. Location factory returns ILocationConfigurator object.
3. Application enables/disables constraint tunc using configureCTunc API.
4. Status of configureCTunc i.e. either SUCCESS or FAILED will be returned to the application.
5. The response for configureCTunc is received by the application.

3.4.10 Call flow to enable/disable PACE

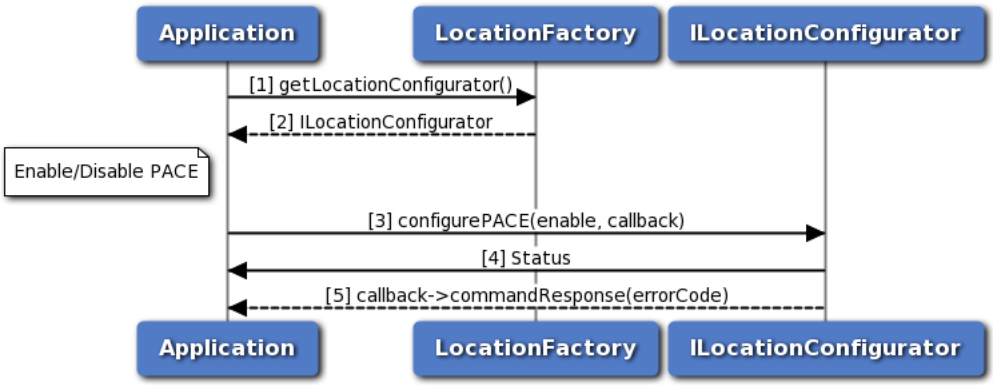

Figure 3-32 Call flow to enable/disable PACE

1. Application requests location factory for location configurator object.
2. Location factory returns ILocationConfigurator object.
3. Application enables/disables PACE using configurePACE API.
4. Status of configurePACE i.e. either SUCCESS or FAILED will be returned to the application.
5. The response for configurePACE is received by the application.

3.4.11 Call flow to delete all aiding data

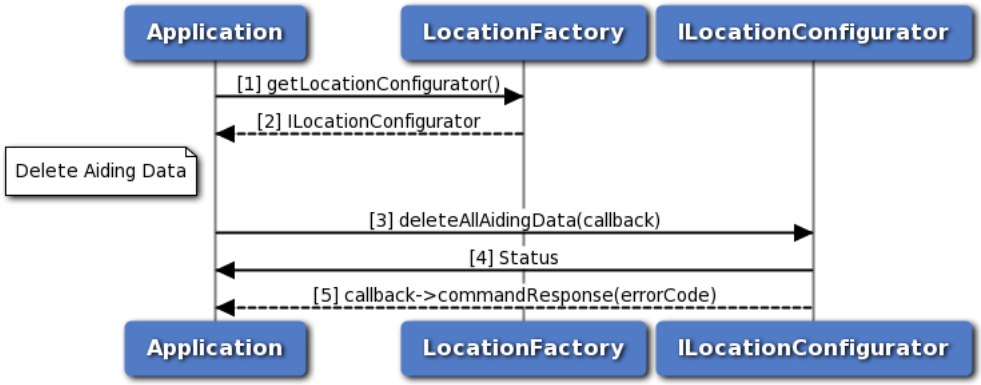

Figure 3-33 Call flow to delete all aiding data

1. Application requests location factory for location configurator object.
2. Location factory returns ILocationConfigurator object.

3. Application deletes all aiding data using deleteAllAidingData API.
4. Status of deleteAllAidingData i.e. either SUCCESS or FAILED will be returned to the application.
5. The response for deleteAllAidingData is received by the application.

3.4.12 Call flow to configure lever arm parameters

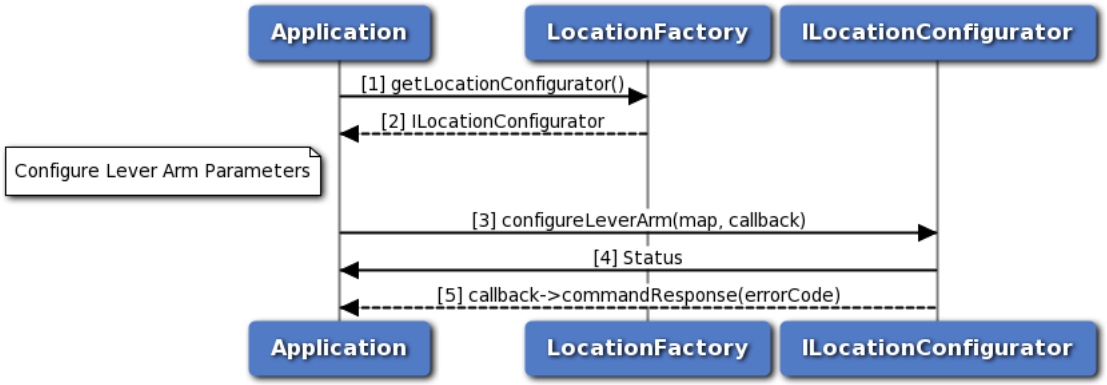

Figure 3-34 Call flow to configure lever arm parameters

1. Application requests location factory for location configurator object.
2. Location factory returns ILocationConfigurator object.
3. Application configures lever arm parameters using configureLeverArm API.
4. Status of configureLeverArm i.e. either SUCCESS or FAILED will be returned to the application.
5. The response for configureLeverArm is received by the application.

3.4.13 Call flow to configure blacklisted constellations

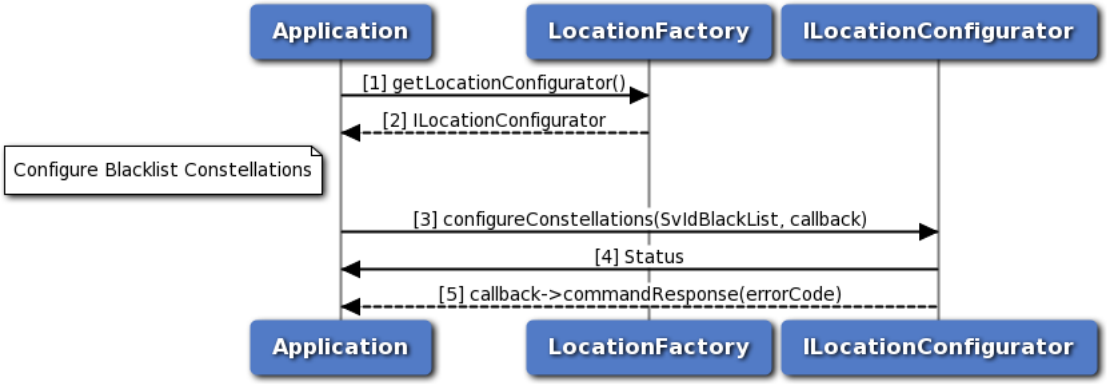

Figure 3-35 Call flow to configure blacklisted constellations

1. Application requests location factory for location configurator object.
2. Location factory returns ILocationConfigurator object.
3. Application configures blacklisted constellations using configureConstellations API.

4. Status of configureConstellations i.e. either SUCCESS or FAILED will be returned to the application.
5. The response for configureConstellations is received by the application.

3.4.14 Call flow to configure robust location

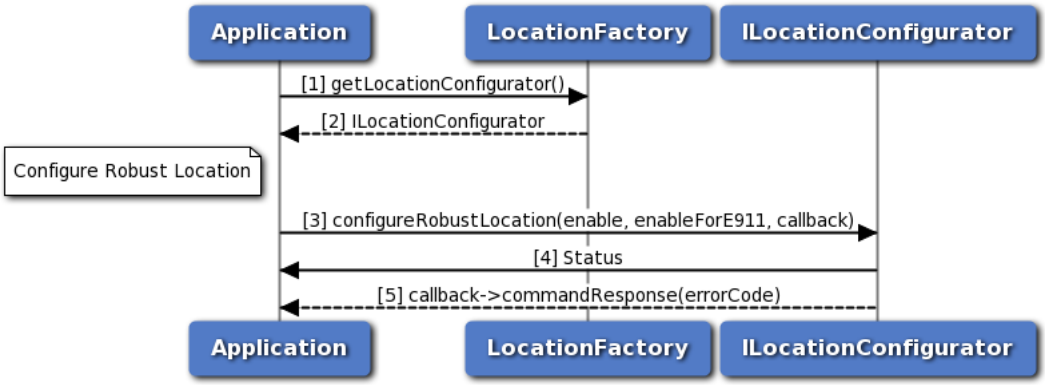

Figure 3-36 Call flow to configure robust location

1. Application requests location factory for location configurator object.
2. Location factory returns ILocationConfigurator object.
3. Application configures robust location using configureRobustLocation API.
4. Status of configureRobustLocation i.e. either SUCCESS or FAILED will be returned to the application.
5. The response for configureRobustLocation is received by the application.

3.4.15 Call flow to configure min gps week

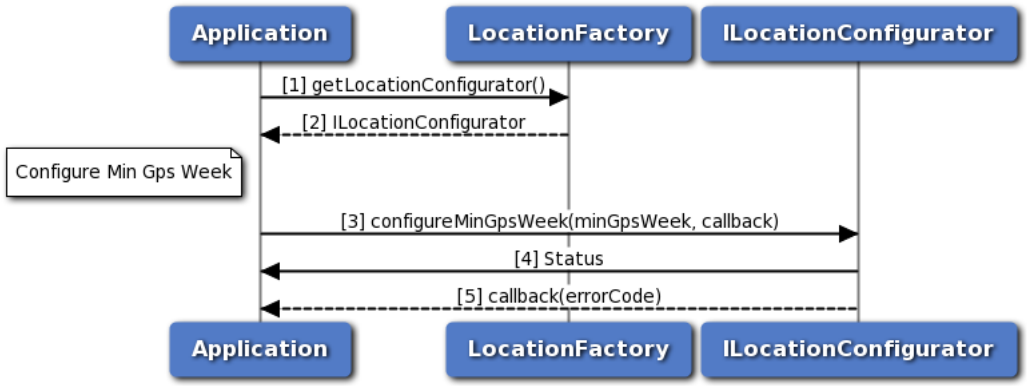

Figure 3-37 Call flow to configure min gps week

1. Application requests location factory for location configurator object.
2. Location factory returns ILocationConfigurator object.
3. Application configures min gps week using configureMinGpsWeek API.

4. Status of configureMinGpsWeek i.e. either SUCCESS or FAILED will be returned to the application.
5. The response for configureMinGpsWeek is received by the application.

3.4.16 Call flow to request min gps week

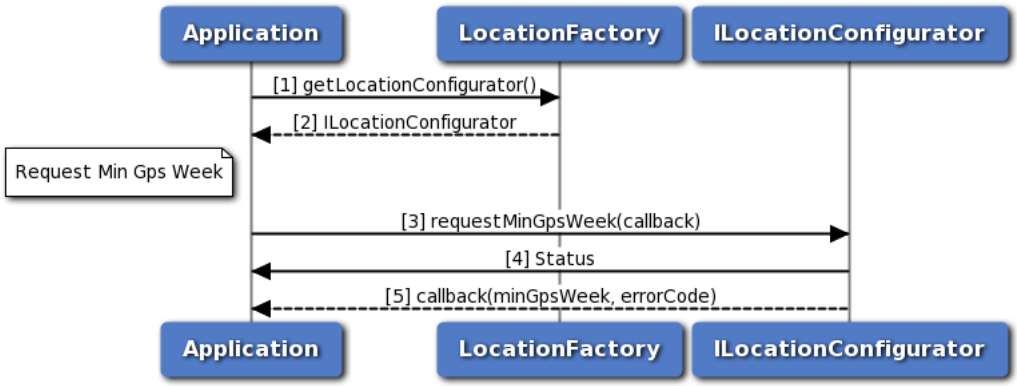

Figure 3-38 Call flow to request min gps week

1. Application requests location factory for location configurator object.
2. Location factory returns ILocationConfigurator object.
3. Application requests min gps week using requestMinGpsWeek API.
4. Status of requestMinGpsWeek i.e. either SUCCESS or FAILED will be returned to the application.
5. The response for requestMinGpsWeek is received by the application.

3.4.17 Call flow to delete specified data

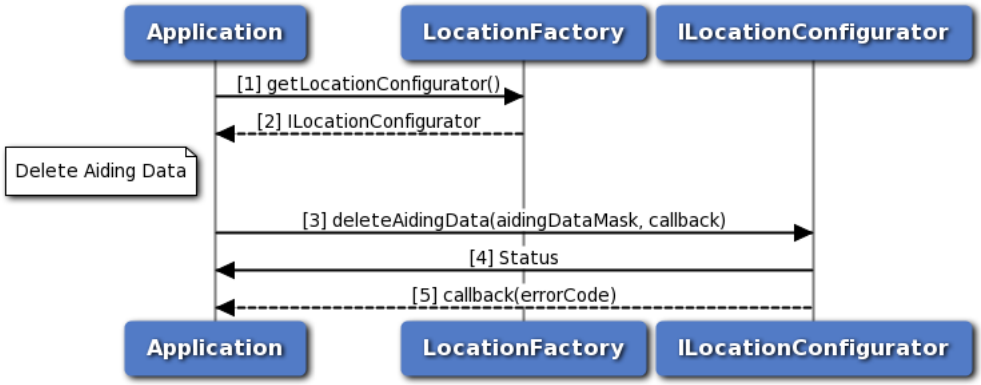

Figure 3-39 Call flow to delete specified data

1. Application requests location factory for location configurator object.
2. Location factory returns ILocationConfigurator object.
3. Application requests delete specified data using deleteAidingData API.
4. Status of deleteAidingData i.e. either SUCCESS or FAILED will be returned to the application.

5. The response for deleteAidingData is received by the application.

3.4.18 Call flow to configure min sv elevation

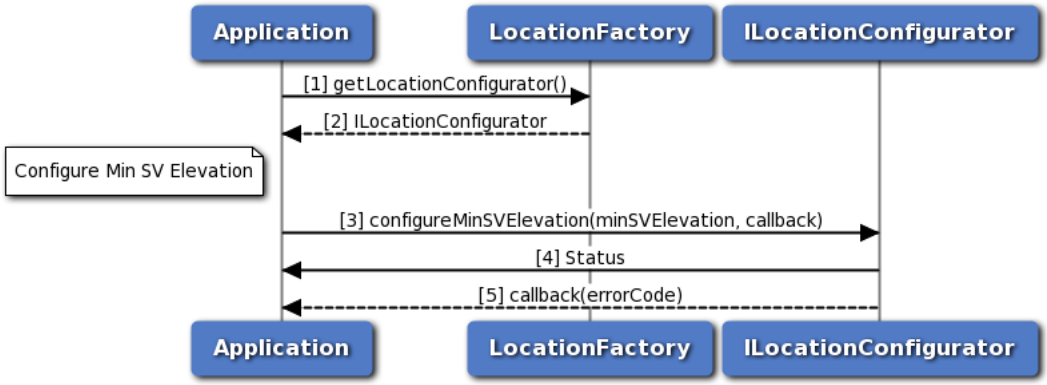

Figure 3-40 Call flow to configure min sv elevation

1. Application requests location factory for location configurator object.
2. Location factory returns ILocationConfigurator object.
3. Application configures min SV elevation using configureMinSVElevation API.
4. Status of configureMinSVElevation i.e. either SUCCESS or FAILED will be returned to the application.
5. The response for configureMinSVElevation is received by the application.

3.4.19 Call flow to request min sv elevation

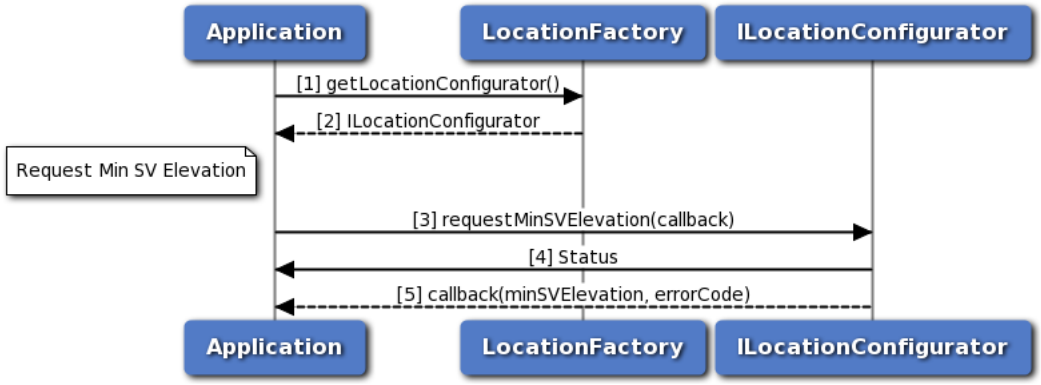

Figure 3-41 Call flow to request min sv elevation

1. Application requests location factory for location configurator object.
2. Location factory returns ILocationConfigurator object.
3. Application requests min SV elevation using requestMinSVElevation API.

4. Status of requestMinSVElevation i.e. either SUCCESS or FAILED will be returned to the application.
5. The response for requestMinSVElevation is received by the application.

3.4.20 Call flow to request robust location

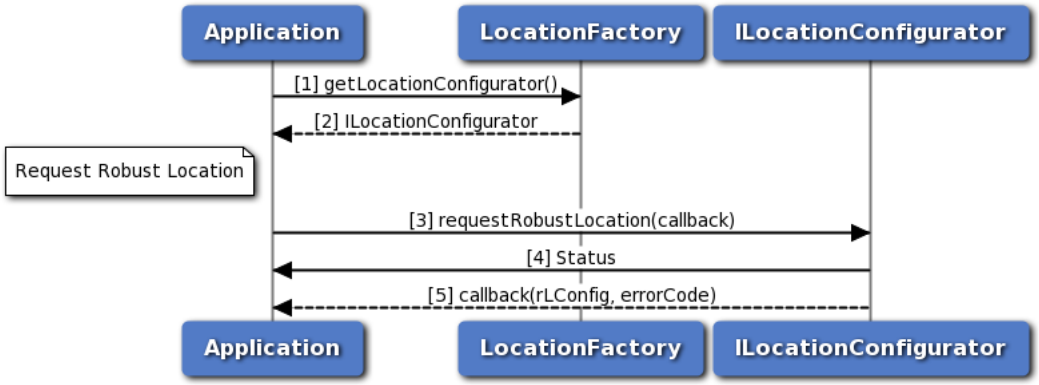

Figure 3-42 Call flow to request robust location

1. Application requests location factory for location configurator object.
2. Location factory returns ILocationConfigurator object.
3. Application requests robust location using requestRobustLocation API.
4. Status of requestRobustLocation i.e. either SUCCESS or FAILED will be returned to the application.
5. The response for requestRobustLocation is received by the application.

3.4.21 Call flow to configure dead reckoning engine

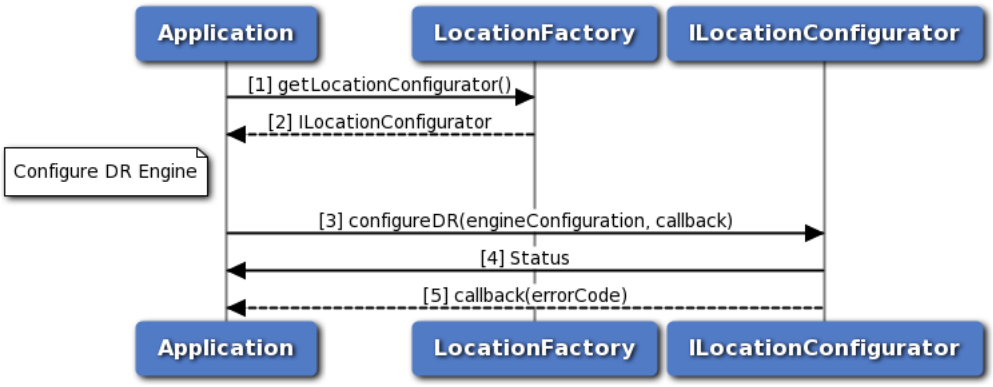

Figure 3-43 Call flow to configure dead reckoning engine

1. Application requests location factory for location configurator object.
2. Location factory returns ILocationConfigurator object.
3. Application configures dead reckoning engine using configureDR API.

4. Status of configureDR i.e. either SUCCESS or FAILED will be returned to the application.
5. The response for configureDR is received by the application.

3.4.22 Call flow to configure secondary band

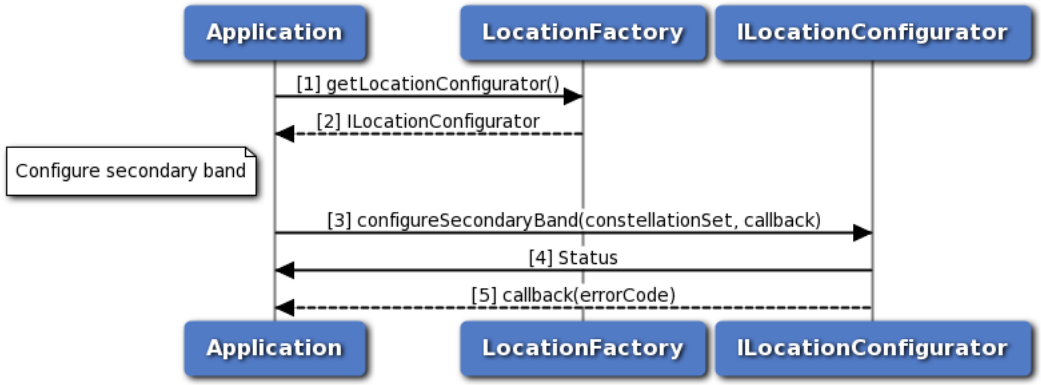

Figure 3-44 Call flow to configure secondary band

1. Application requests location factory for location configurator object.
2. Location factory returns ILocationConfigurator object.
3. Application configures secondary band using configureSecondaryBand API.
4. Status of configureSecondaryBand i.e. either SUCCESS or FAILED will be returned to the application.
5. The response for configureSecondaryBand is received by the application.

3.4.23 Call flow to request secondary band

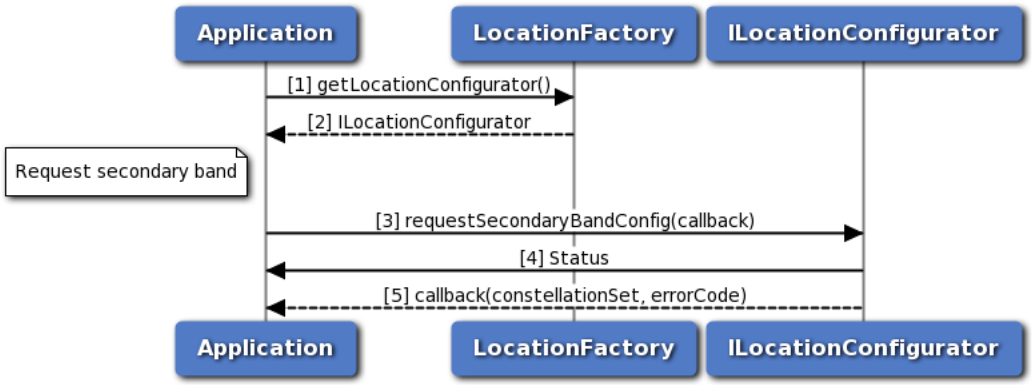

Figure 3-45 Call flow to request secondary band

1. Application requests location factory for location configurator object.
2. Location factory returns ILocationConfigurator object.
3. Application requests secondary band using requestSecondaryBandConfig API.

4. Status of requestSecondaryBandConfig i.e. either SUCCESS or FAILED will be returned to the application.
5. The response for requestSecondaryBandConfig is received by the application.

3.4.24 Call flow to configure engine state

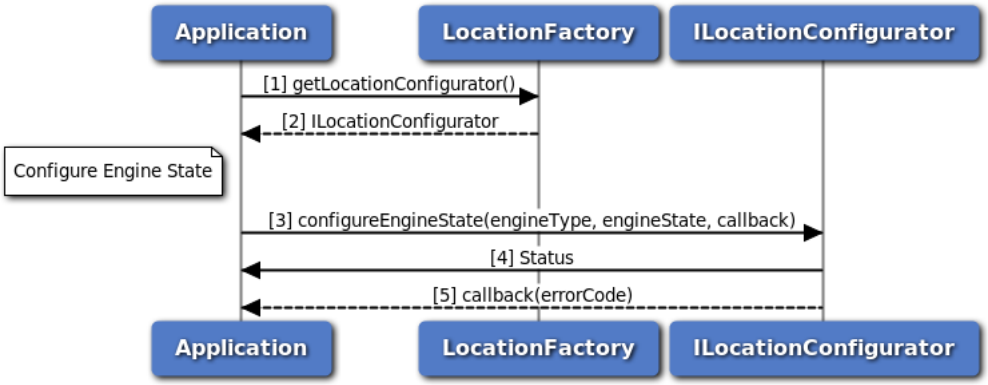

Figure 3-46 Call flow to configure engine state

1. Application requests location factory for location configurator object.
2. Location factory returns ILocationConfigurator object.
3. Application configures engine state using configureEngineState API.
4. Status of configureEngineState i.e. either SUCCESS or FAILED will be returned to the application.
5. The response for configureEngineState is received by the application.

3.4.25 Call flow to provide user consent for terrestrial positioning

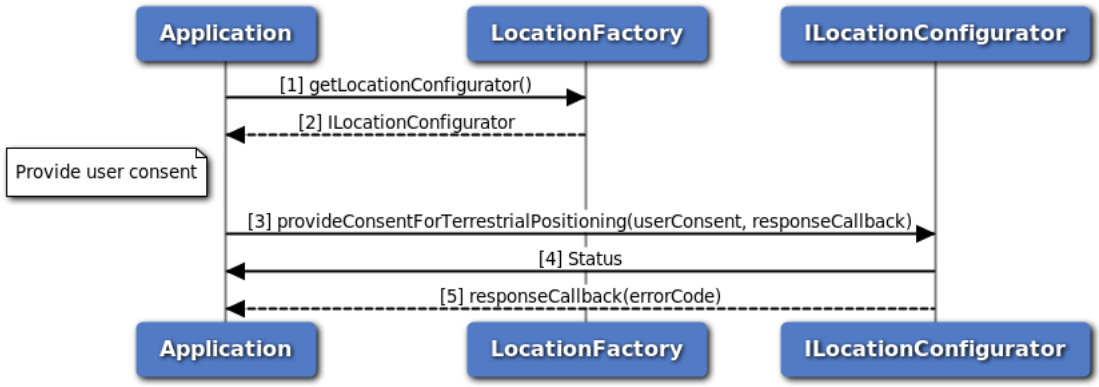

Figure 3-47 Call flow to provide user consent for terrestrial positioning

1. Application requests location factory for location configurator object.
2. Location factory returns ILocationConfigurator object.

3. Application provides user consent for terrestrial positioning using provideConsentForTerrestrialPositioning API.
4. Status of provideConsentForTerrestrialPositioning i.e. either SUCCESS or FAILED will be returned to the application.
5. The response for provideConsentForTerrestrialPositioning is received by the application.

3.4.26 Call flow to configure NMEA sentence type

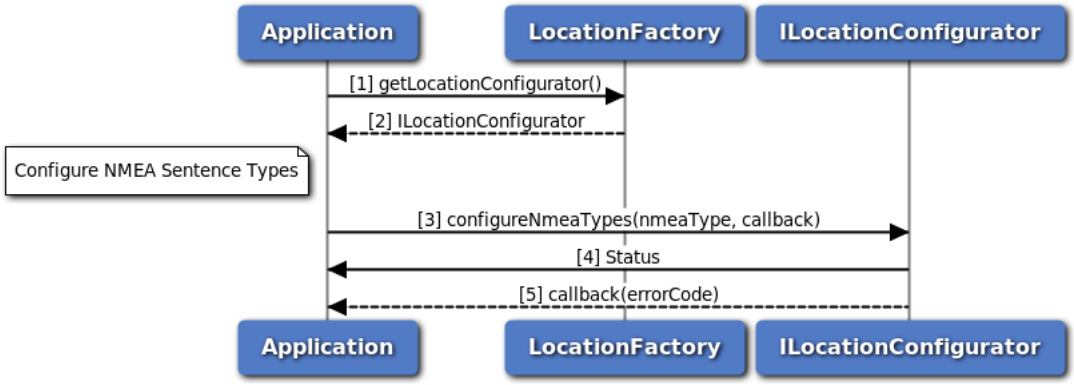

Figure 3-48 Call flow to configure NMEA sentence type

1. Application requests location factory for location configurator object.
2. Location factory returns ILocationConfigurator object.
3. Application configures NMEA sentence type using configureNmeaTypes API.
4. Status of configureNmeaTypes i.e. either SUCCESS or FAILED will be returned to the application.
5. The response for configureNmeaTypes is received by the application.

3.4.27 Call flow to represent Xtra Feature

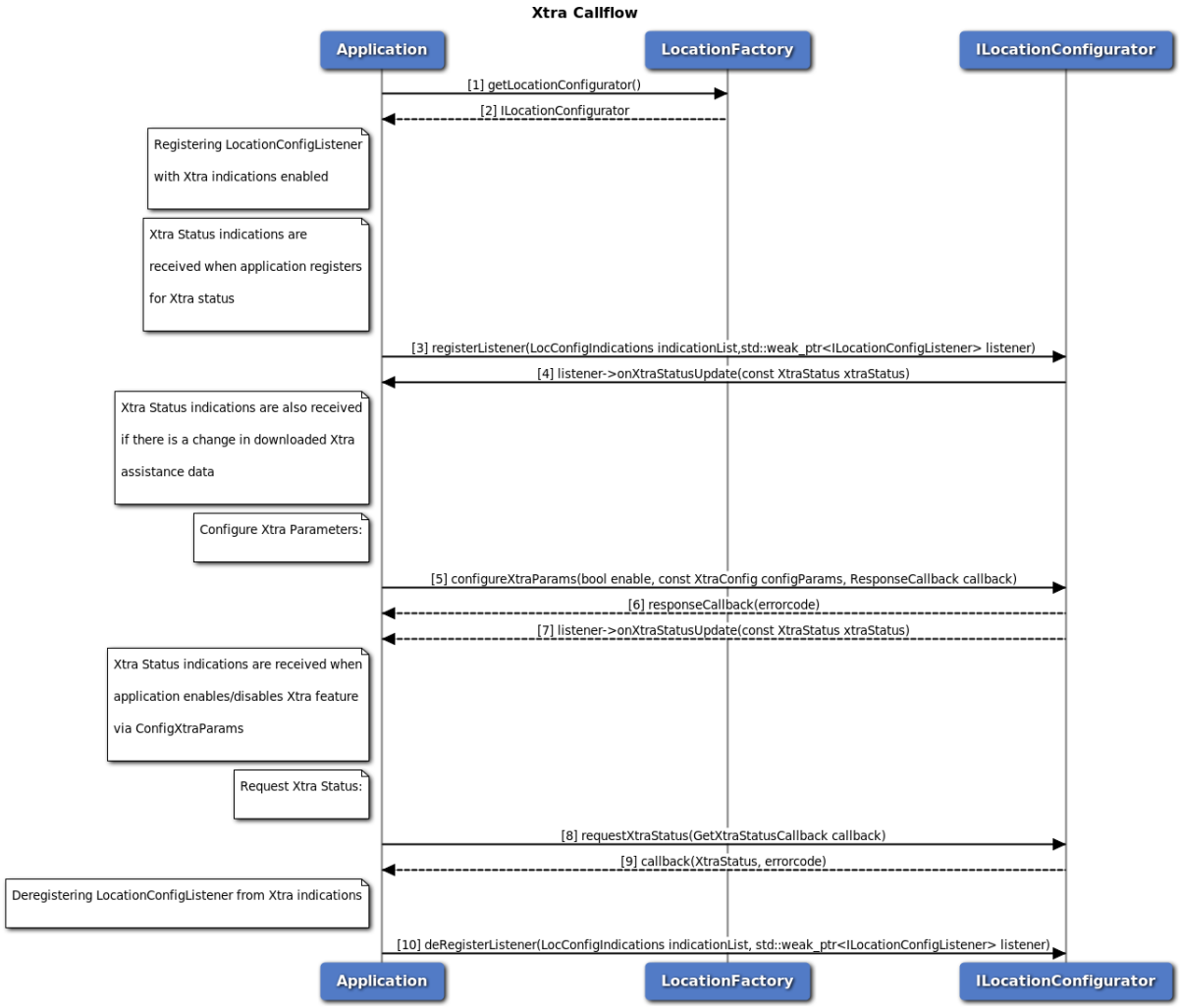

Figure 3-49 Call flow to represent Xtra Feature

1. Application requests location factory for location configurator object.
2. Location factory returns ILocationConfigurator object.
3. Application registers config listener for Xtra Status indications.
4. Listener API onXtraStatusUpdate is invoked while registering for Xtra indications.
5. Application configures Xtra Params using configureXtraParams.
6. Location Configurator invokes response callback with errorcode.
7. Listener API onXtraStatusUpdate is invoked when Xtra feature is enabled/disabled.
8. Application requests Xtra Status using requestXtraStatus.
9. Location Configurator invokes GetXtraStatus callback with XtraStatus and errorcode.
10. Application deregisters config listener from Xtra Status indications.

3.4.28 Call flow to inject RTCM correction data with dgnss manager

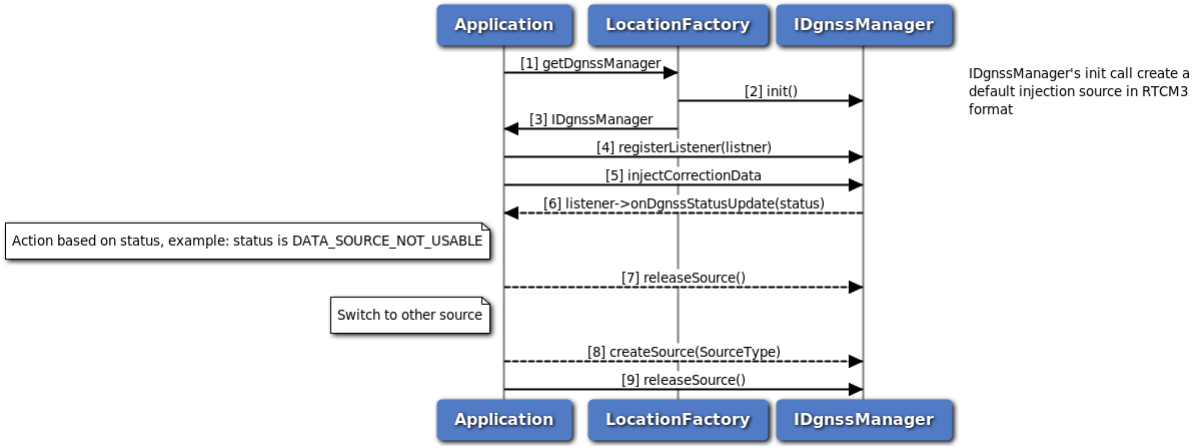

Figure 3-50 Call flow to inject RTCM correction data with dgnss manager

1. Application requests location factory for Dgnss manager object.
2. Location factory create IDgnssManager instance and perform initialization.
3. Location factory returns IDgnssManager object.
4. Application register a status listener to get notification of the dgnss manager status change.
5. Application start injecting RTCM data.
6. If the status listener received any error notification, it perform desired operation.
7. An example case is the listener received DATA_SOURCE_NOT_USABLE notification, it then release the current source and create a new source and perform RTCM data injection.

3.5 Data Services

Applications need to have "radio" Linux group permissions to be able to operate successfully with underlying services.

3.5.1 Start/Stop for data connection manager call flow

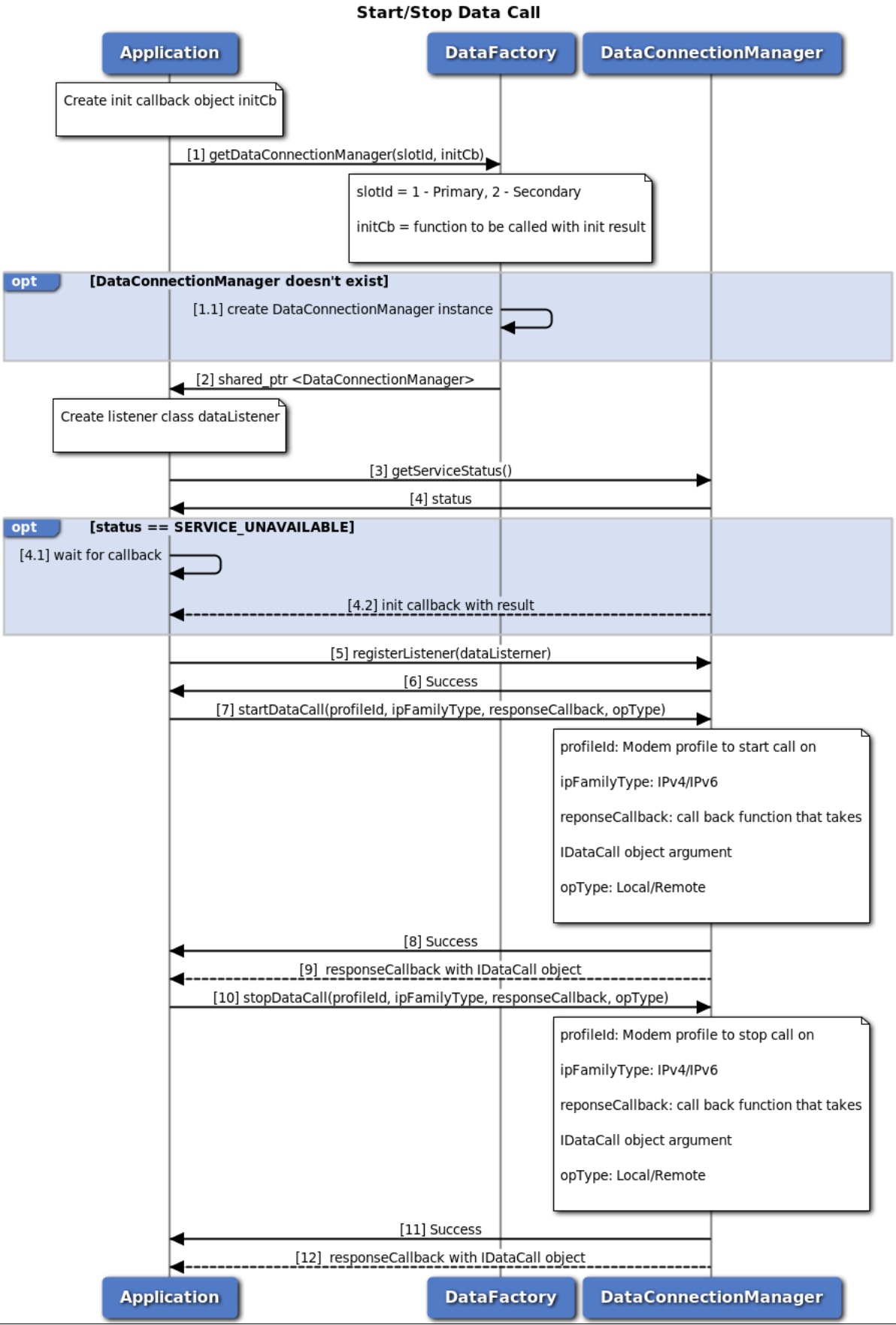

Figure 3-51 Start/Stop data call call flow

1. Application requests Data Connection Manager object associated with sim id from data factory. Application can optionally provide callback to be called when manager initialization is completed.
2. Data factory returns shared pointer to data connection manager object to application.
3. Application request current service status of data connection manager returned by data factory.
4. Data connection manager returns current service status. 4.1 If status returned is SERVICE_UNAV↩ AILABLE (manager is not ready), application should wait for init callback provided in step 1. 4.2 Data connection manager calls application callback with initialization result (success/failure).
5. Application registers as listener to get notifications for data call change.
6. The application receives the status i.e. either SUCCESS or FAILED based on the registration of the listener.
7. Application requests for start data call and optionally gets asynchronous response using startDataCallback.
8. Application receives the status i.e. either SUCCESS or FAILED based on the execution of startDataCall.
9. Optionally, the application gets asynchronous response for startDataCall using startDataCallback.
10. Application requests for stop data call and optionally gets asynchronous response using stopDataCallback.
11. Application receives the status i.e. either SUCCESS or FAILED based on the execution of stopDataCall.
12. Optionally, the application gets asynchronous response for stopDataCall using stopDataCallback.

3.5.2 Request data profile list call flow

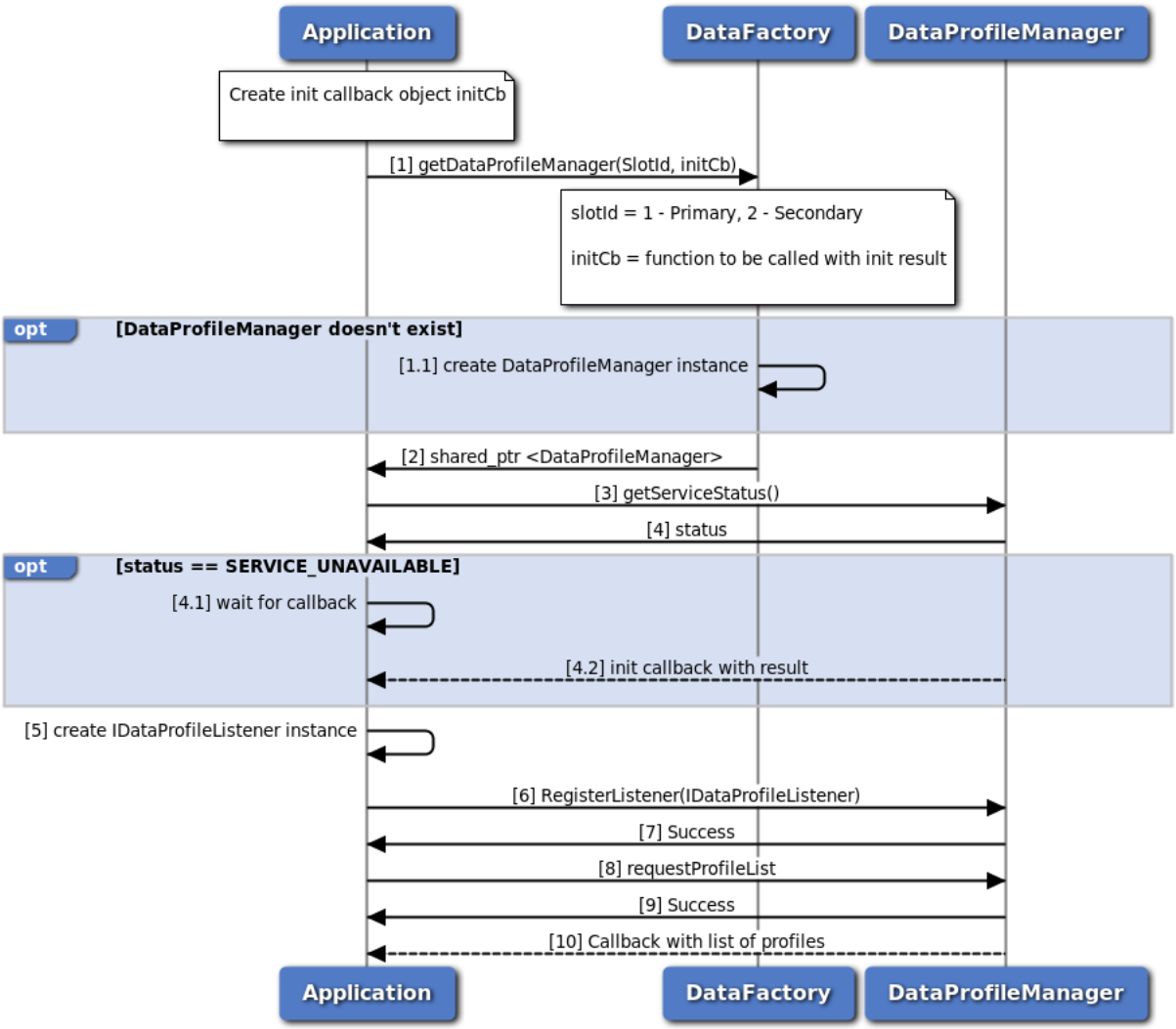

Figure 3-52 Request data profile list call flow

1. Application requests Data profile Manager object associated with sim id from data factory. Application can optionally provide callback to be called when manager initialization is completed.
2. Data factory returns shared pointer to data profile manager object to application.
3. Application request current service status of data profile manager returned by data factory.
4. Data profile manager returns current service status. 4.1 If status returned is SERVICE_UNAVAIL↩ ABLE (manager is not ready), application should wait for init callback provided in step 1. 4.2 Data profile manager calls application callback with initialization result (success/failure).
5. Application creates listener class of type IDataProfileListener
6. Application register created object in step 5 as listener to data profile changes
7. Application receives the status i.e. either SUCCESS or FAILED based on the execution of registerListener
8. Application requests list of profile

9. Application receives the status i.e. either SUCCESS or FAILED based on the execution of requestProfileList
10. Application gets callback with list of all profiles

3.5.3 Data Serving System Manager Call Flow

Data Serving System manager provides the interface to access network and modem low level services. It provides APIs to get current dedicated radio bearer, get current service status and preferred RAT, and get current roaming status. Serving System Listener provides an interface for application to receive data serving system notification events such as change in dedicated radio bearer, change in service status, or change in roaming status. The application must register as a listener for Serving System updates.

3.5.3.1 Get Dedicated Radio Bearer Call Flow

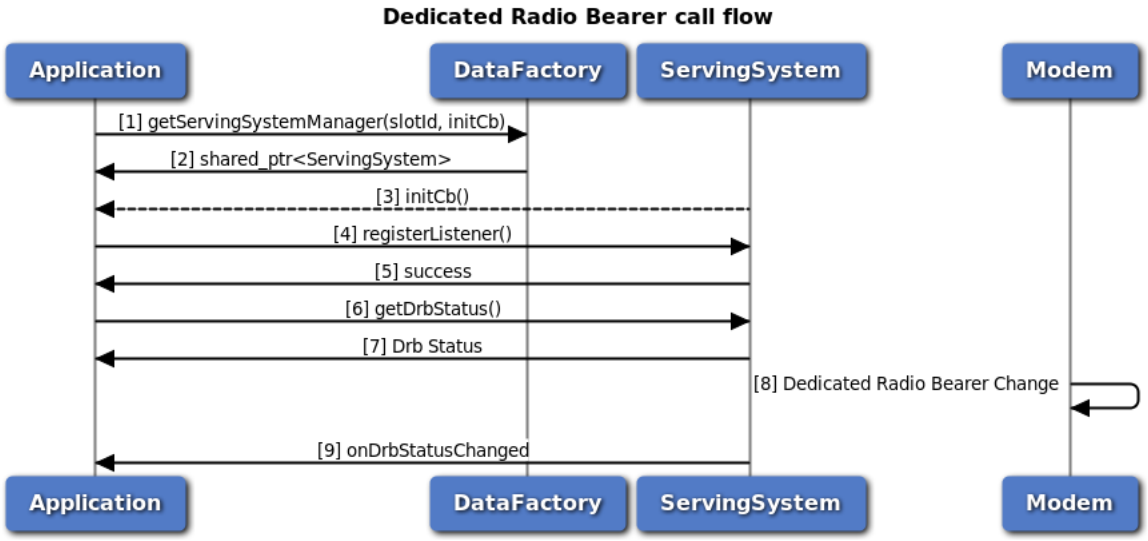

Figure 3-53 Get Dedicated Radio Bearer Call Flow

1. Application requests data factory for data serving system manager object with slot Id and init callback.
2. Data factory returns IServingSystemManager object to application.
3. Serving System Manager calls application callback provided in step 1 with initialization result pass/fail.
4. Application register itself as listener with Serving System manager to receive dedicated radio bearer changes notification.
5. Application gets success for registering as listener.
6. Application calls getDrbStatus to get current dedicated radio bearer status.
7. Application receives the current dedicated bearer status.
8. Dedicated radio bearer changes in modem
9. Application gets onDrbStatusChanged indication with new status

3.5.3.2 Request Service Status Call Flow

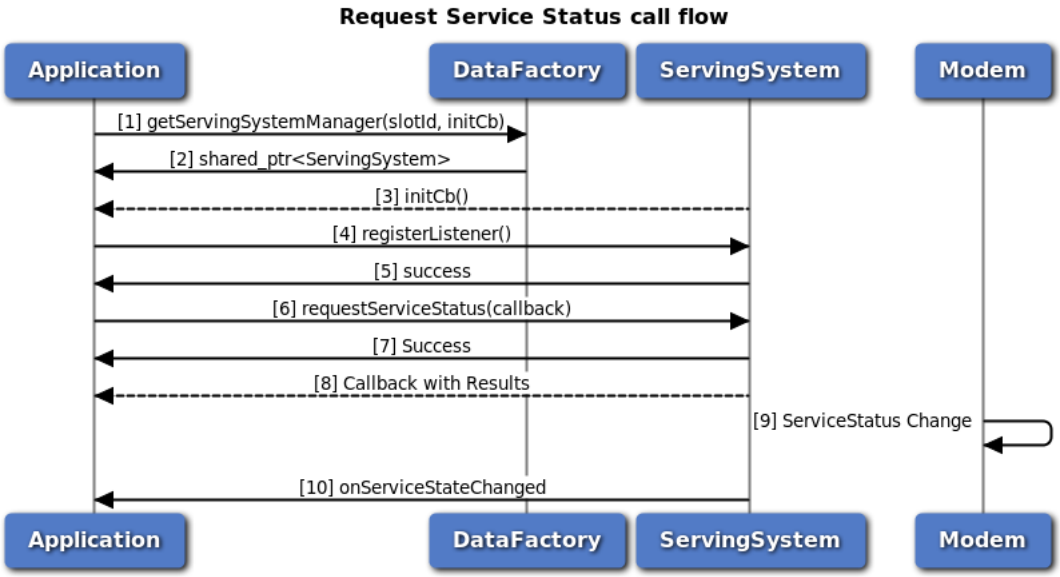

Figure 3-54 Request service status call flow

1. Application requests data factory for data serving system manager object with slot Id and init callback.
2. Data factory returns IServingSystemManager object to application.
3. Serving System Manager calls application callback provided in step 1 with initialization result pass/fail.
4. Application register itself as listener with Serving System manager to receive service status changes notification.
5. Application gets success for registering as listener.
6. Application calls requestServiceStatus to get current service status and provides callback.
7. Application receives the status i.e. either SUCCESS or FAILED based on the execution of requestServiceStatus.
8. Application gets asynchronous response for requestServiceStatus through callback provided in step 4 with current service status.
9. Service status changes in modem
10. Application gets onServiceStateChanged indication with new status

3.5.3.3 Request Roaming Status Call Flow

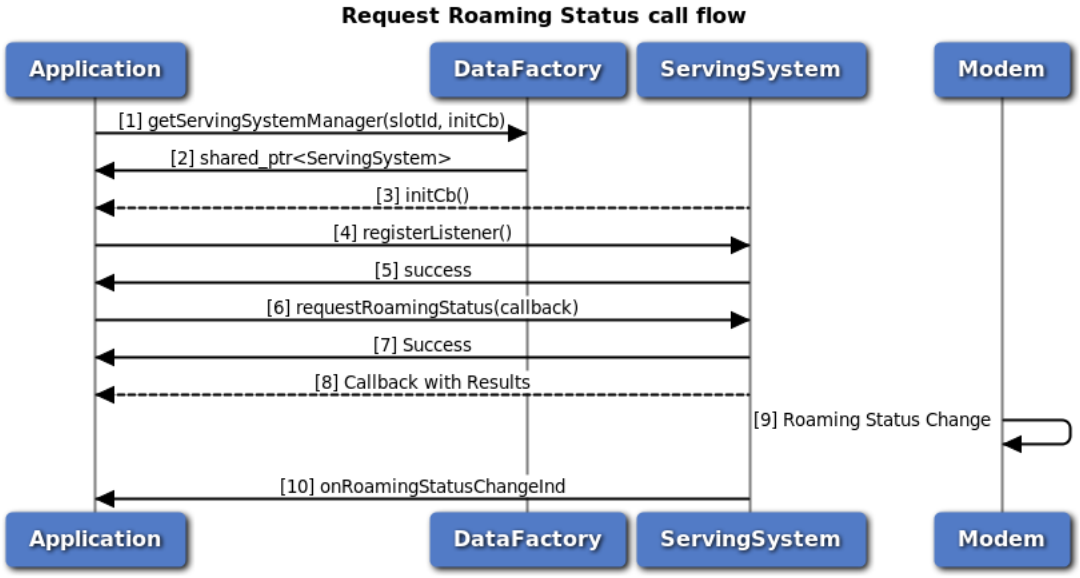

Figure 3-55 Request roaming status call flow

1. Application requests data factory for data serving system manager object with slot Id and init callback.
2. Data factory returns IServingSystemManager object to application.
3. Serving System Manager calls application callback provided in step 1 with initialization result pass/fail.
4. Application register itself as listener with Serving System manager to receive roaming status changes notification.
5. Application gets success for registering as listener.
6. Application calls requestRoamingStatus to get current service status and provides callback.
7. Application receives the status i.e. either SUCCESS or FAILED based on the execution of requestRoamingStatus.
8. Application gets asynchronous response for requestRoamingStatus through callback provided in step 4 with current service status.
9. Roaming status changes in modem
10. Application gets onRoamingStatusChanged indication with new status

3.5.4 Data Filter Manager Call Flow

Data Filter manager provides APIs to get/set data filter mode, add/remove data restrict filters. Its API can used per data call or globally to apply the same changes to all the underlying currently up data call. It also has listener interface for notifications for data filter status update. Application will get the Data Filter manager object from data factory. The application can register a listener for data filter mode change updates.

3.5.4.1 Call flow to Set/Get data filter mode

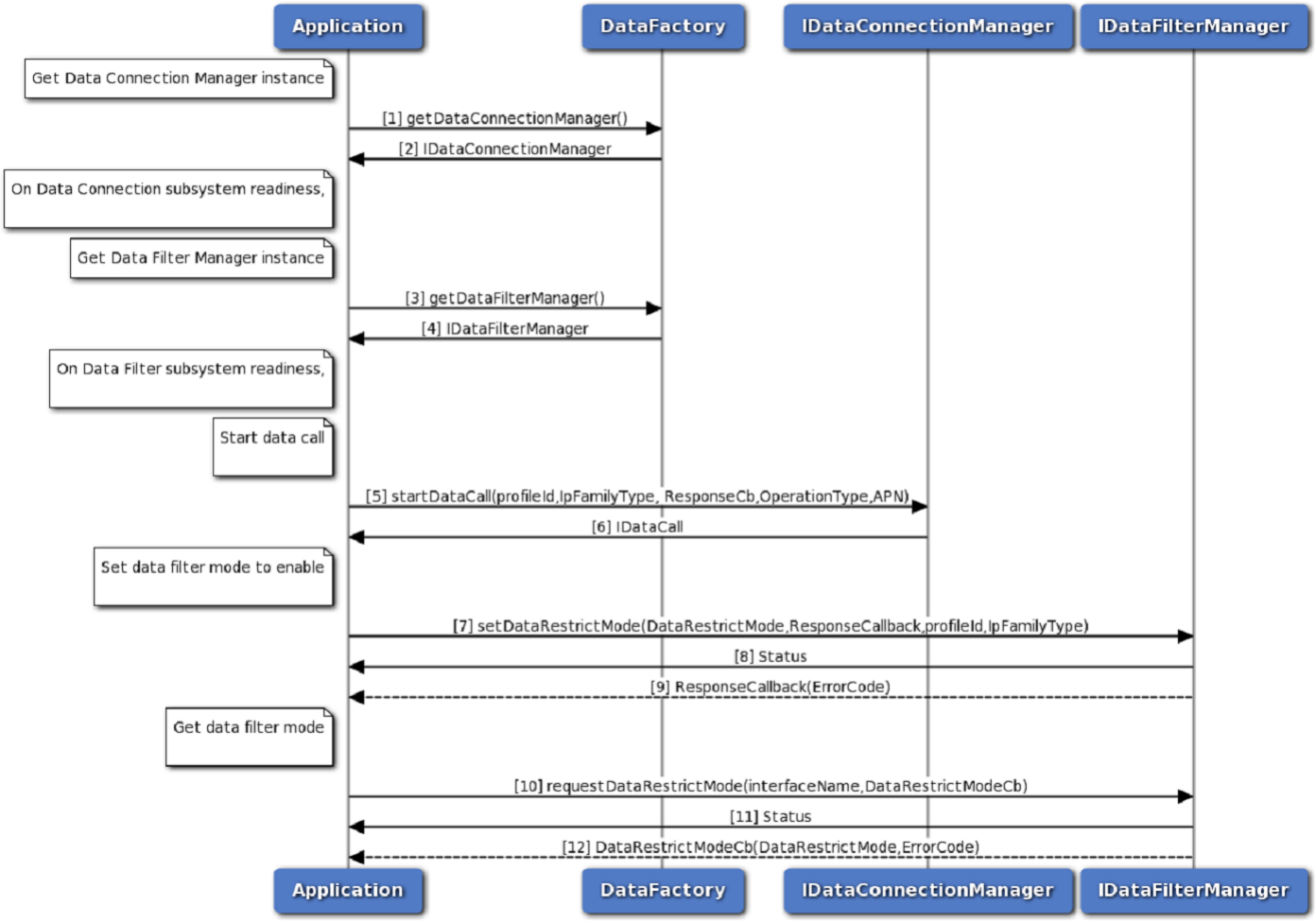

Figure 3-56 Get/Set data filter mode call flow

1. Application requests data factory for data connection manager object.
2. Data factory returns IDataConnectionManager object to application.
3. Application requests data factory for data filter manager object.
4. Data factory returns IDataFilterManager object to application.
5. Application requests for start data call and optionally gets asynchronous response using startDataCallback.
6. Application receives the status i.e. either SUCCESS or FAILED based on the execution of startDataCall.
7. Optionally, the application gets asynchronous response for startDataCall using startDataCallback.
8. Application requests for set data filter mode to enable and optionally gets asynchronous response using ResponseCallback.

9. Application receives the status i.e. either SUCCESS or FAILED based on the execution of setDataRestrictMode.
10. Optionally, the application gets asynchronous response for setDataRestrictMode using ResponseCallback.
11. Application requests for get data filter mode and optionally gets asynchronous response using DataRestrictModeCb.
12. Application receives the status i.e. either SUCCESS or FAILED based on the execution of requestDataRestrictMode.
13. Optionally, the application gets asynchronous response for requestDataRestrictMode using DataRestrictModeCb.

3.5.4.2 Call flow to Add data restrict filter

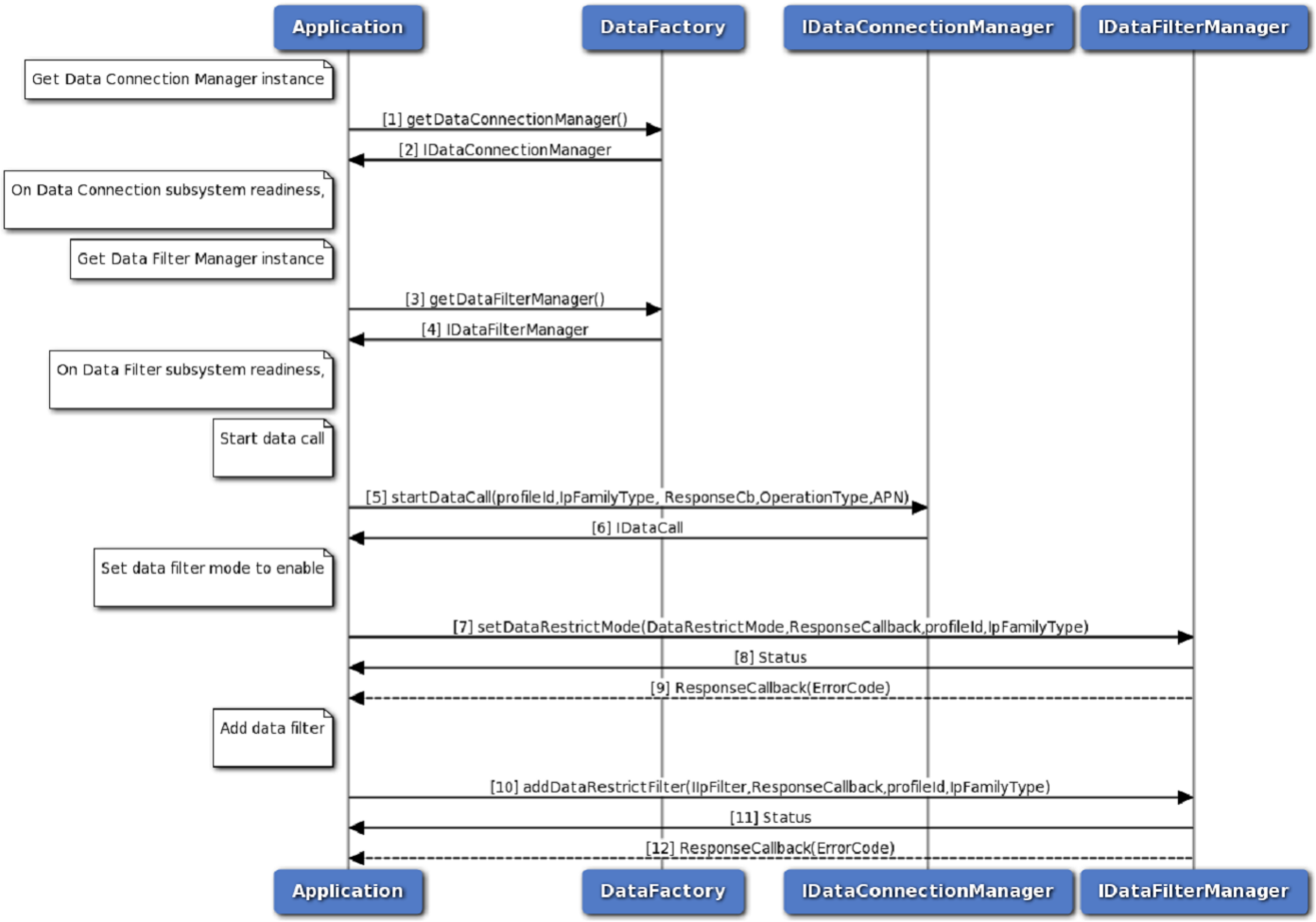

Figure 3-57 Add data restrict filter call flow

1. Application requests data factory for data connection manager object.
2. Data factory returns IDataConnectionManager object to application.

3. Application requests data factory for data filter manager object.
4. Data factory returns IDataFilterManager object to application.
5. Application requests for start data call and optionally gets asynchronous response using startDataCallback.
6. Application receives the status i.e. either SUCCESS or FAILED based on the execution of startDataCall.
7. Optionally, the application gets asynchronous response for startDataCall using startDataCallback.
8. Application requests for add data filter and optionally gets asynchronous response using ResponseCallback.
9. Application receives the status i.e. either SUCCESS or FAILED based on the execution of addDataRestrictFilter.
10. Optionally, the application gets asynchronous response for addDataRestrictFilter using ResponseCallback.

3.5.4.3 Call flow to Remove data restrict filter

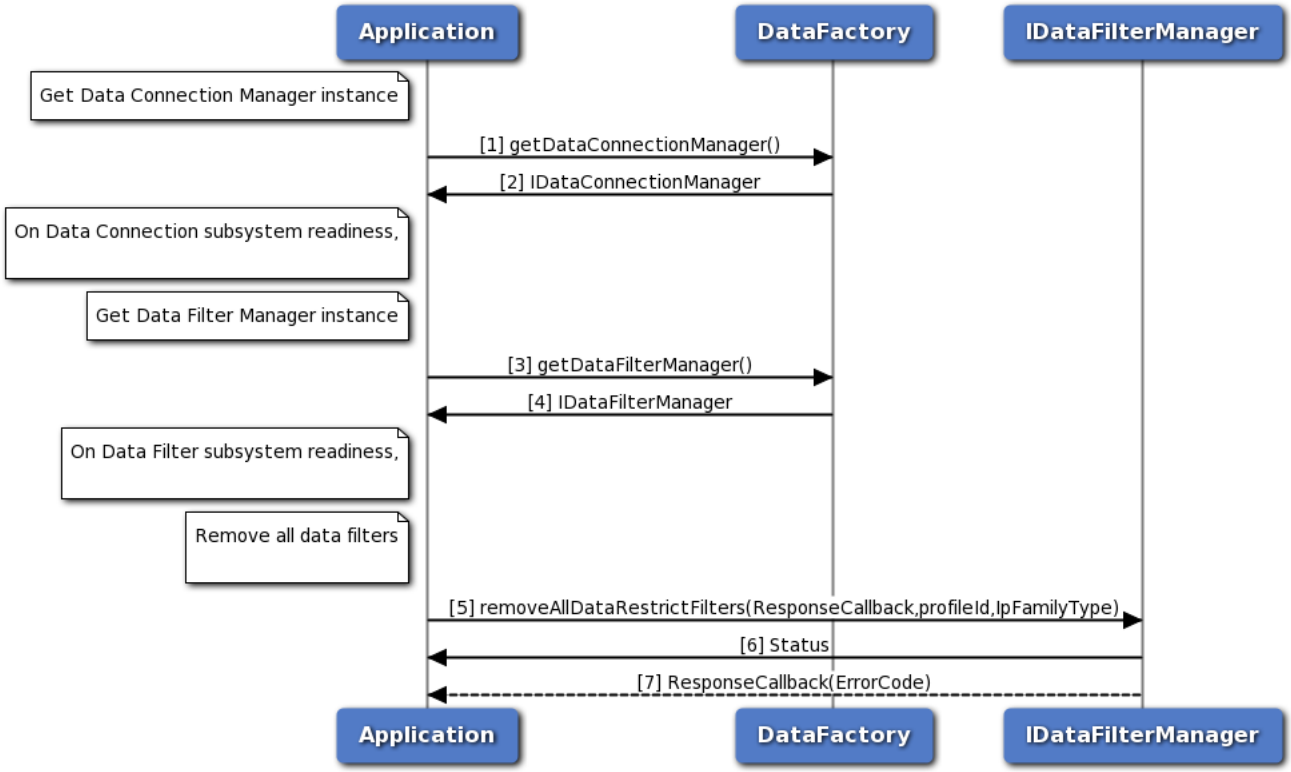

Figure 3-58 Remove data restrict filter call flow

1. Application requests data factory for data connection manager object.
2. Data factory returns IDataConnectionManager object to application.
3. Application requests data factory for data filter manager object.

4. Data factory returns IDataFilterManager object to application.
5. Application requests for start data call and optionally gets asynchronous response using startDataCallback.
6. Application receives the status i.e. either SUCCESS or FAILED based on the execution of startDataCall.
7. Optionally, the application gets asynchronous response for startDataCall using startDataCallback.
8. Application requests for add data filter and optionally gets asynchronous response using ResponseCallback.
9. Application receives the status i.e. either SUCCESS or FAILED based on the execution of removeAllDataRestrictFilters.
10. Optionally, the application gets asynchronous response for removeAllDataRestrictFilters using ResponseCallback.

3.5.5 Data Networking Call Flow

Application will get the following manager objects from data factory to configure networking. IVlanManager is used to access all VLAN APIs. INatManager is used to access all Static NAT APIs. IFirewallManager is used to access all Firewall APIs.

3.5.5.1 Create VLAN and Bind it to PDN in data vlan manager call flow

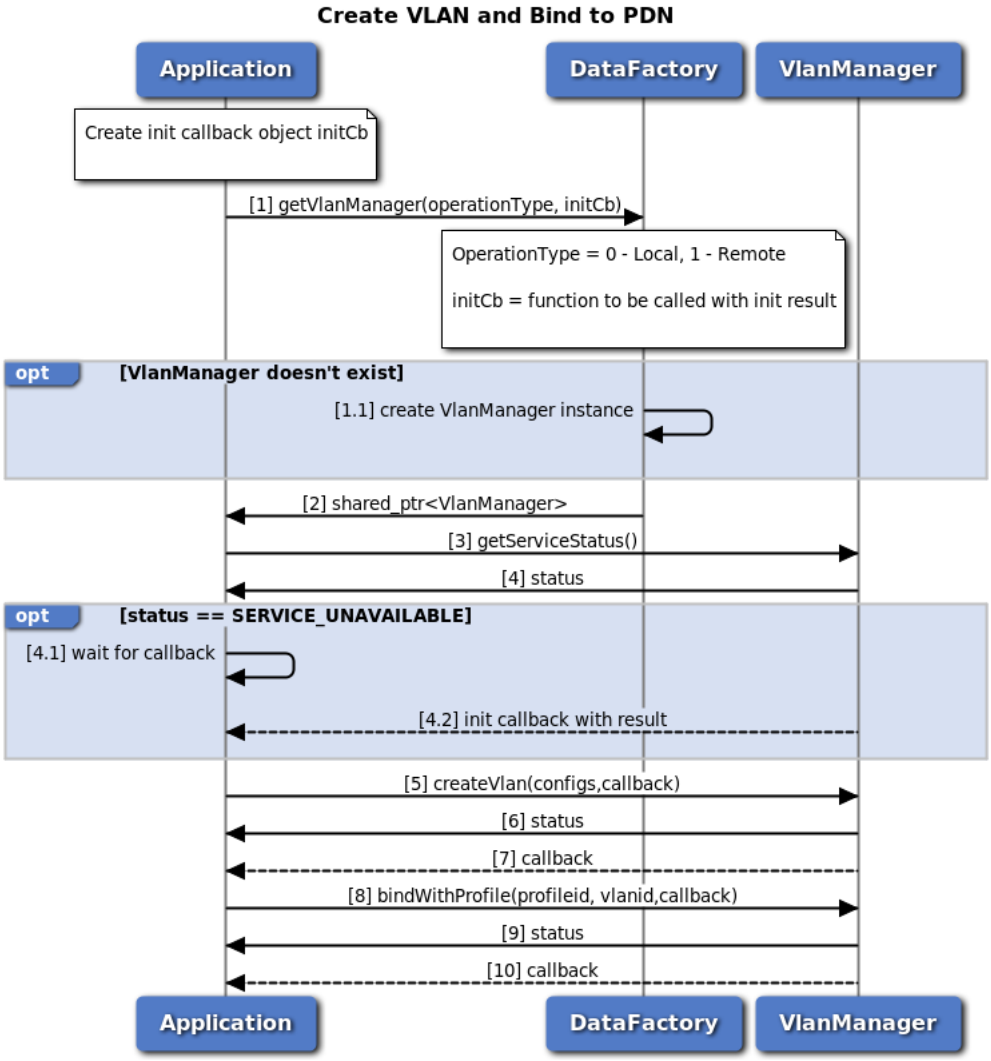

Figure 3-59 Create VLAN and bind it to PDN for data VLAN manager call flow

1. Application requests data factory for data IVlanManager object. Application can optionally provide callback to be called when manager initialization is completed. 1.1. If IVlanManager object does not exist, data factory will create new object.
2. Data factory returns shared pointer to IVlanManager object to application.
3. Application request current service status of vlan manager returned by data factory
4. The application receives the Status i.e. either true or false to indicate whether sub-system is ready or not. 4.1. If status returned is SERVICE_UNAVAILABLE (manager is not ready), application should wait for init callback provided in step 1 4.2. Vlan manager calls application callback with initialization result (success/failure).
5. On success, application calls IVlanManager::createVlan with assigned id, interface, and acceleration type.
6. Application receives synchronous Status which indicates if the IVlanManager::createVlan request

was sent successfully.

7. Application is notified of the Status of the IVlanManager::createVlan request (either SUCCESS or FAILED) via the application-supplied callback.
8. Application calls IVlanManager::bindWithProfile with Vlan id and profile id.
9. Application receives synchronous Status which indicates if the IVlanManager::bindWithProfile request was sent successfully.

1. Application is notified of the Status of the IVlanManager::bindWithProfile request (either SUCCESS or FAILED) via the application-supplied callback.

3.5.5.2 LAN-LAN VLAN Configuration from EAP usecase call flow

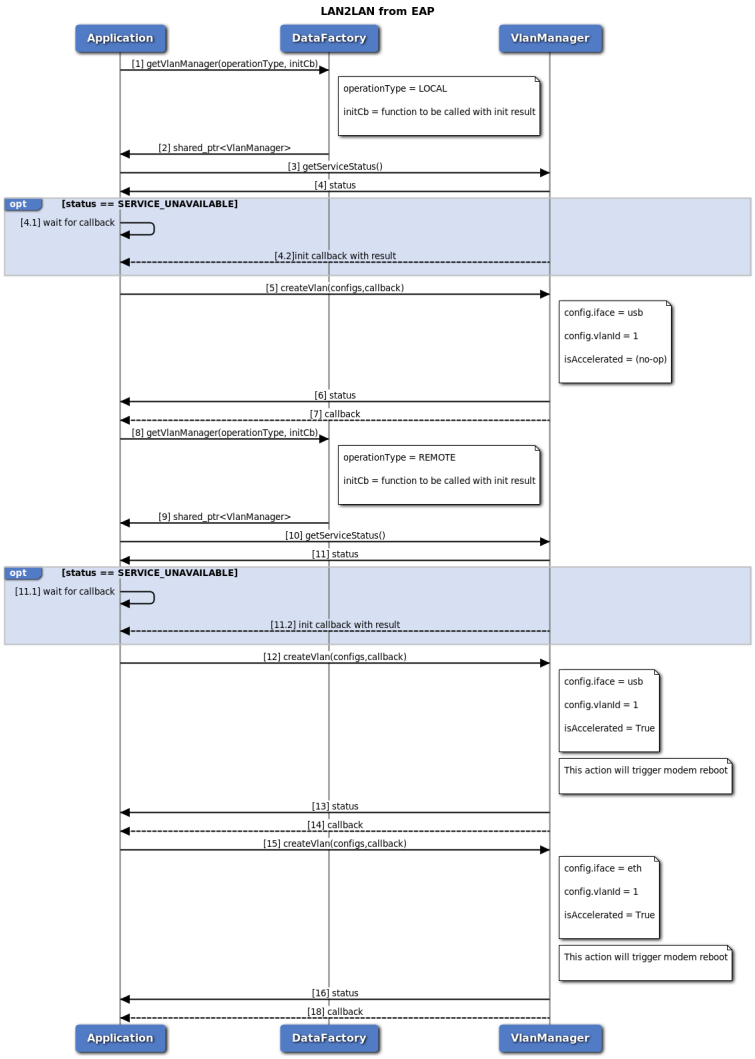

Figure 3-60 LAN-LAN VLAN Configuration Usecase from EAP call flow

1. Application requests data factory for data IVlanManager for local operation object. Aplication can optionally provide callback to be called when manager initialization is completed.
2. Data factory returns shared pointer to local vlan manager object to application.

3. Application request current service status of local vlan manager returned by data factory.
4. Vlan manager returns current service status. 4.1 If status returned is SERVICE_UNAVAILABLE (manager is not ready), application should wait for init callback provided in step 1. 4.2 Vlan manager calls application callback with initialization result (success/failure).
5. On success, application calls IVlanManager::createVlan with USB interface, Vlan id 1 and no acceleration.
6. Vlan manager returns synchronous response to application (success/fail).
7. Vlan manager calls application provided callback in step 5 with createVlan results
8. Application requests data factory for data IVlanManager for remote operation object. Aplication can optionally provide callback to be called when manager initialization is completed.
9. Data factory returns shared pointer to remote vlan manager object to application.
10. Application request current service status of remote vlan manager returned by data factory.
11. Vlan manager returns current service status. 11.1 If status returned is SERVICE_UNAVAILABLE (manager is not ready), application should wait for init callback provided in step 1. 11.2 Vlan manager calls application callback with initialization result (success/failure).
12. On success, application calls IVlanManager::createVlan with USB interface, Vlan id 1 and acceleration.
13. Vlan manager returns synchronous response to application (success/fail).
14. Vlan manager calls application provided callback in step 12 with createVlan results.
15. Application calls IVlanManager::createVlan with ETH interface, Vlan id 1 and acceleration.
16. Vlan manager returns synchronous response to application (success/fail).
17. Vlan manager calls application provided callback in step 15 with createVlan results.

3.5.5.3 LAN-LAN VLAN Configuration from A7 usecase call flow

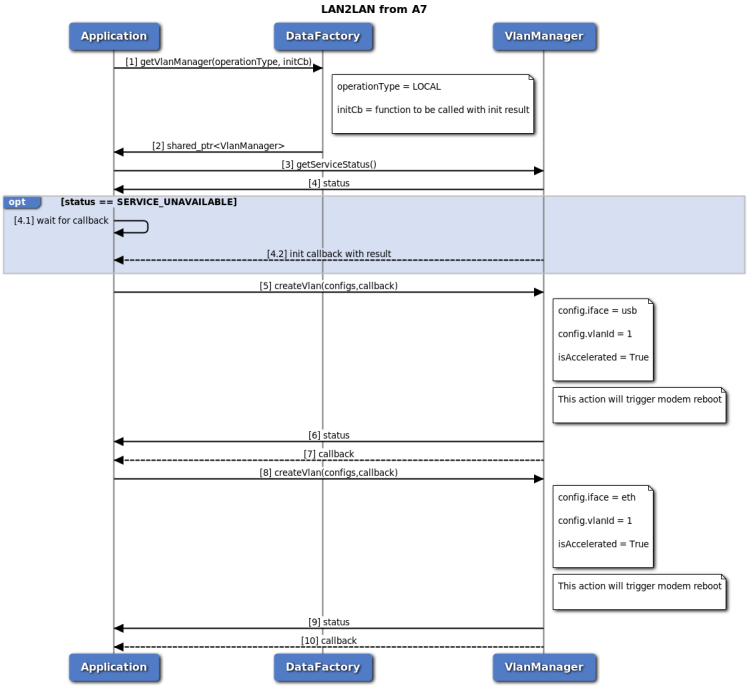

Figure 3-61 LAN-LAN VLAN Configuration Usecase from A7 call flow

1. Application requests data factory for data IVlanManager for local operation object. Aplication can optionally provide callback to be called when manager initialization is completed.
2. Data factory returns shared pointer to local vlan manager object to application.
3. Application request current service status of local vlan manager returned by data factory.
4. Vlan manager returns current service status. 4.1 If status returned is SERVICE_UNAVAILABLE (manager is not ready), application should wait for init callback provided in step 1. 4.2 Vlan manager calls application callback with initialization result (success/failure).
5. On success, application calls IVlanManager::createVlan with USB interface, Vlan id 1 and acceleration.
6. Vlan manager returns synchronous response to application (success/fail).
7. Vlan manager calls application provided callback in step 5 with createVlan results.
8. Application calls IVlanManager::createVlan with ETH interface, Vlan id 1 and acceleration.
9. Vlan manager returns synchronous response to application (success/fail).
10. Vlan manager calls application provided callback in step 8 with createVlan results.

3.5.5.4 LAN-WAN VLAN Configuration from EAP usecase call flow

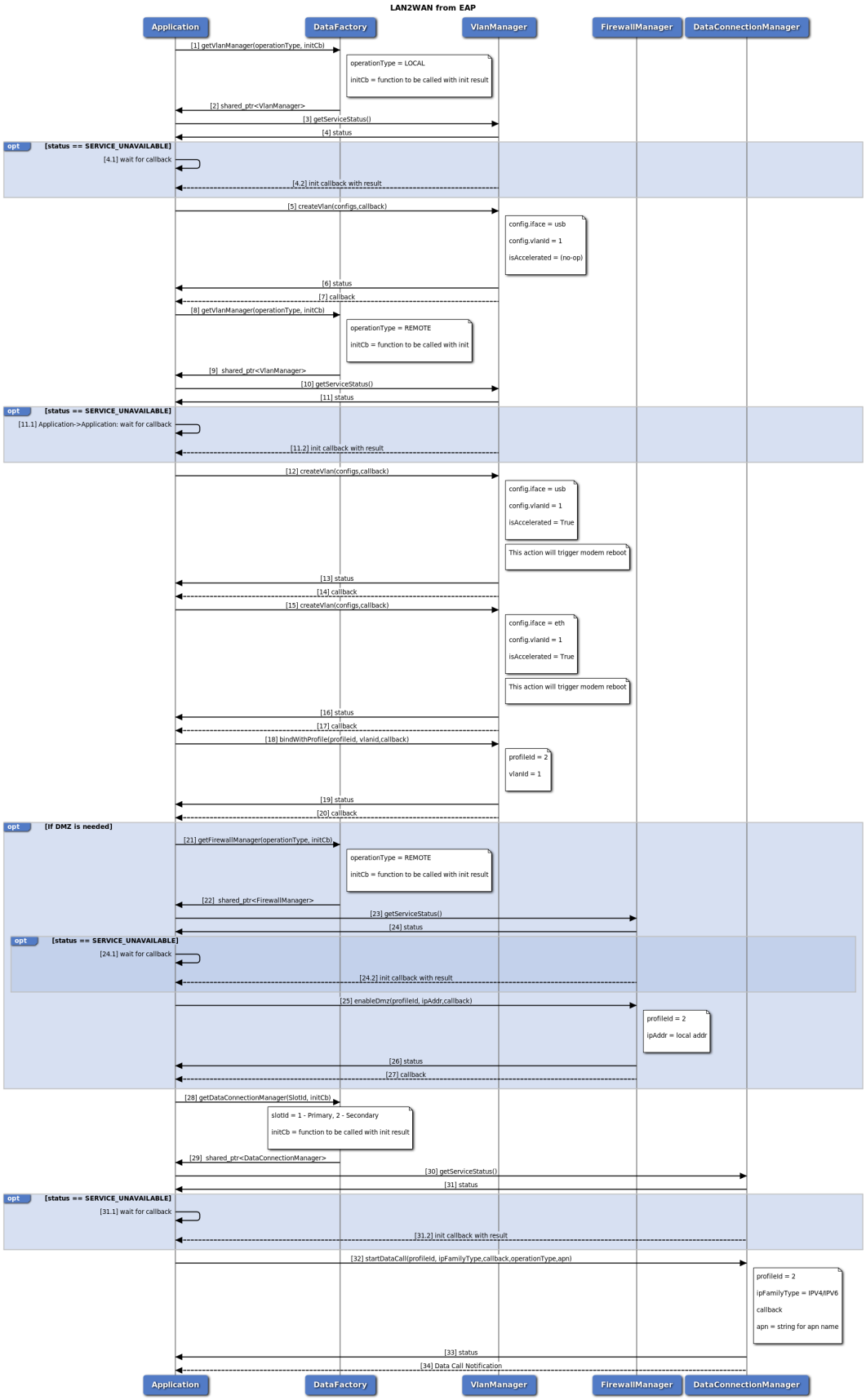

Figure 3-62 LAN-WAN VLAN Configuration Usecase from EAP call flow

1. Application requests data factory for local data vlan manager object.
2. Data factory returns shared pointer to local vlan manager to application.
3. Application request current service status of local vlan manager returned by data factory.
4. Vlan manager returns current service status. 4.1 If status returned is SERVICE_UNAVAILABLE (manager is not ready), application should wait for init callback provided in step 1. 4.2 Vlan manager calls application callback with initialization result (success/failure).
5. On success, application calls IVlanManager::createVlan with USB interface, Vlan id 1 and no acceleration.
6. Vlan manager returns synchronous response to application (success/fail).
7. Vlan manager calls application provided callback in step 5 with createVlan results.
8. Application requests data factory for remote data vlan manager object.
9. Data factory returns shared pointer to remote vlan manager to application.
10. Application request current service status of remote vlan manager returned by data factory.
11. Vlan manager returns current service status. 11.1 If status returned is SERVICE_UNAVAILABLE (manager is not ready), application should wait for init callback provided in step 8. 11.2 Vlan manager calls application callback with initialization result (success/failure).
12. On success, application calls IVlanManager::createVlan with USB interface, Vlan id 1 and acceleration.
13. Vlan manager returns synchronous response to application (success/fail).
14. Vlan manager calls application provided callback in step 12 with createVlan results.
15. Application calls IVlanManager::createVlan with ETH interface, Vlan id 1 and acceleration.
16. Vlan manager returns synchronous response to application (success/fail).
17. Vlan manager calls application provided callback in step 15 with createVlan results.
18. Application calls IVlanManager::bindWithProfile with profile id to bind with.
19. Vlan manager returns synchronous response to application (success/fail).
20. Vlan manager calls application provided callback in step 18 with bindWithProfile results. If DMZ is needed:
    (a) Application requests data factory for firewall manager object.
    (b) Data factory returns shared pointer to firewall manager to application.
    (c) Application request current service status of firewall manager returned by data factory.
    (d) Firewall manager returns current service status. 24.1 If status returned is SERVICE_UNAVA↩ ILABLE (manager is not ready), application should wait for init callback provided in step 21. 24.2 Firewall manager calls application callback with initialization result (success/failure).
    (e) Application calls firewall manager enableDmz with profile id and local address to be enable Dmz on.
    (f) Firewall manager returns synchronous response to application (success/fail).
    (g) Firewall manager calls application provided callback in step 25 with enableDmz results.
21. Application requests data factory for data connection manager object.

22. Data factory returns shared pointer to data connection manager to application.
23. Application request current service status of data connection manager returned by data factory.
24. Data connection manager returns current service status. 31.1 If status returned is SERVICE_UNA↩ VAILABLE (manager is not ready), application should wait for init callback provided in step 28. 31.2 Data connection manager calls application callback with initialization result (success/failure).
25. Application can call data Connection manager startDataCall with profile id to start data call on, Ip Family type, operation Type and APN Name.
26. Data connection Manager returns synchronous response to application (success/fail).
27. Data connection Manager returns notification to application with data call details.

3.5.5.5 LAN-WAN VLAN Configuration from A7 usecase call flow

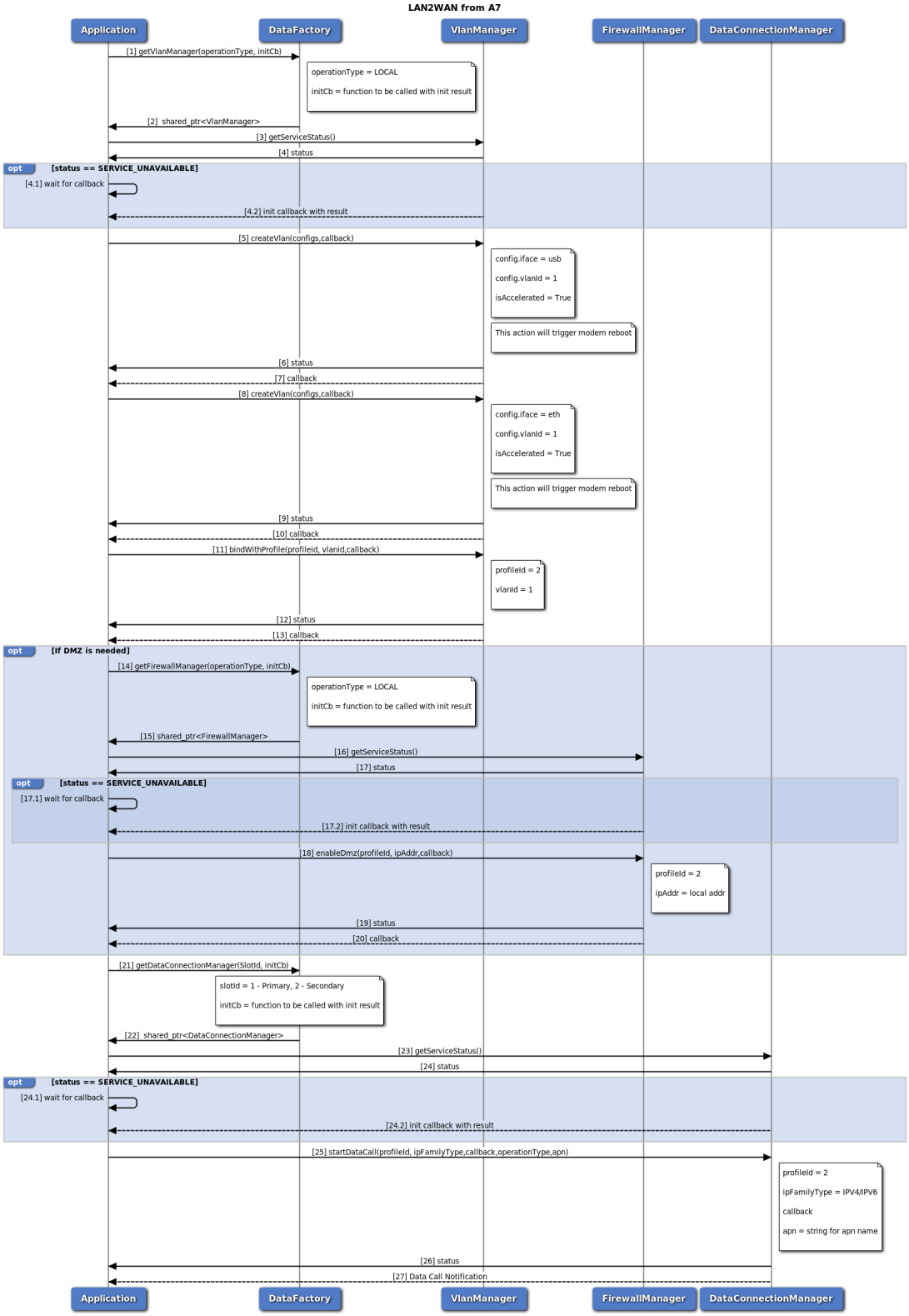

Figure 3-63 LAN-WAN VLAN Configuration Usecase from A7 call flow

1. Application requests data factory for local data vlan manager object.
2. Data factory returns shared pointer to local vlan manager to application.
3. Application request current service status of local vlan manager returned by data factory.
4. Vlan manager returns current service status. 4.1 If status returned is SERVICE_UNAVAILABLE (manager is not ready), application should wait for init callback provided in step 1. 4.2 Vlan manager calls application callback with initialization result (success/failure).
5. On success, application calls IVlanManager::createVlan with USB interface, Vlan id 1 and acceleration.
6. Vlan manager returns synchronous response to application (success/fail).
7. Vlan manager calls application provided callback in step 5 with createVlan results.
8. Application calls IVlanManager::createVlan with ETH interface, Vlan id 1 and acceleration.
9. Vlan manager returns synchronous response to application (success/fail).
10. Vlan manager calls application provided callback in step 8 with createVlan results.
11. Application calls IVlanManager::bindWithProfile with profile id to bind with.
12. Vlan manager returns synchronous response to application (success/fail).
13. Vlan manager calls application provided callback in step 11 with bindWithProfile results. If DMZ is needed:
    (a) Application requests data factory for firewall manager object.
    (b) Data factory returns shared pointer to firewall manager to application.
    (c) Application request current service status of firewall manager returned by data factory.
    (d) Firewall manager returns current service status. 17.1 If status returned is SERVICE_UNAVA↩ ILABLE (manager is not ready), application should wait for init callback provided in step 14. 17.2 Firewall manager calls application callback with initialization result (success/failure).
    (e) Application calls firewall manager enableDmz with profile id and local address to be enable Dmz on.
    (f) Firewall manager returns synchronous response to application (success/fail).
    (g) Firewall manager calls application provided callback in step 25 with enableDmz results.
14. Application requests data factory for data connection manager object.
15. Data factory returns shared pointer to data connection manager to application.
16. Application request current service status of data connection manager returned by data factory.
17. Data connection manager returns current service status. 24.1 If status returned is SERVICE_UNA↩ VAILABLE (manager is not ready), application should wait for init callback provided in step 21. 24.2 Data connection manager calls application callback with initialization result (success/failure).
18. Application can call data Connection manager startDataCall with profile id to start data call on, Ip Family type, operation Type and APN Name.
19. Data connection Manager returns synchronous response to application (success/fail).
20. Data connection Manager returns notification to application with data call details.

3.5.5.6 Create Static NAT entry in data Static NAT manager call flow

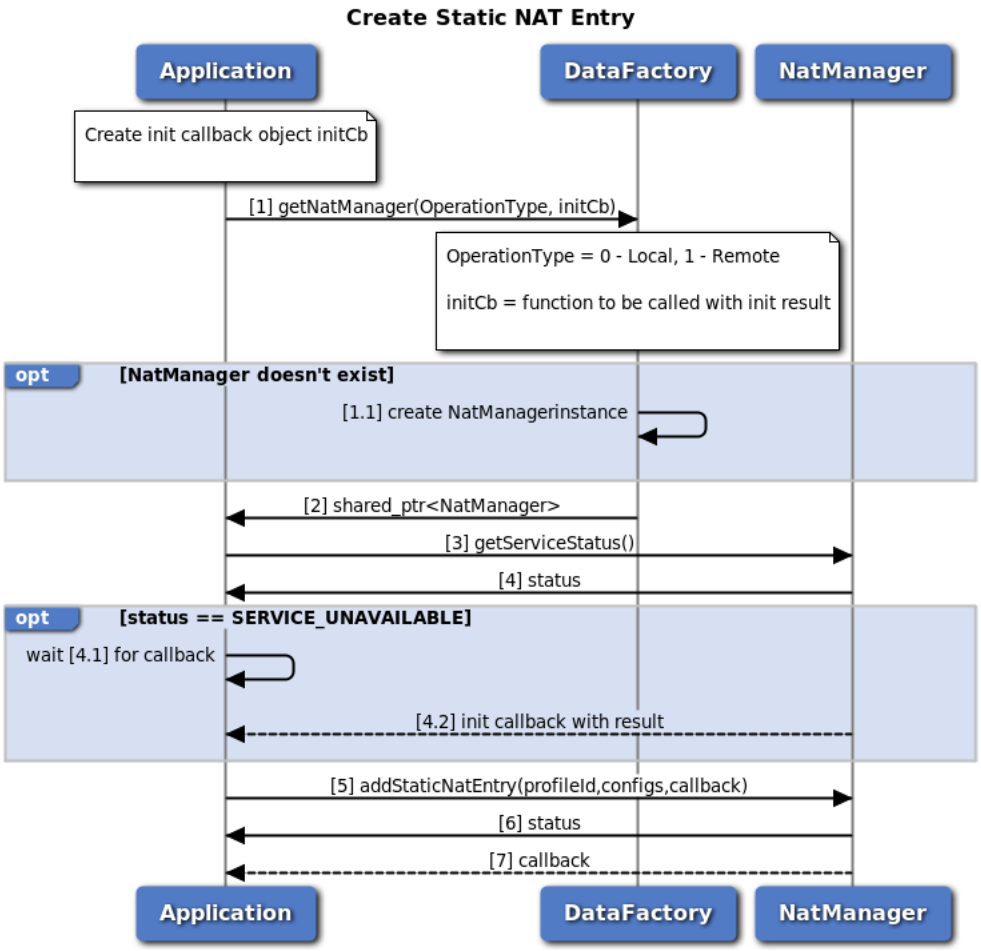

Figure 3-64 Create Static NAT entry for data NAT manager call flow

1. Application requests data factory for data nat manager object. 1.1. If nat manager object does not exist, data factory will create new object.
2. Data factory returns shared pointer to nat manager object to application.
3. Application request current service status of nat manager returned by data factory.
4. Nat manager returns current service status. 4.1 If status returned is SERVICE_UNAVAILABLE (manager is not ready), application should wait for init callback provided in step 1. 4.2 Nat manager calls application callback with initialization result (success/failure).
5. On success, application calls nat manager addStaticNatEntry with profileId, private IP address port, private port, global port and IP Protocol.
6. Application receives synchronous Status which indicates if the nat manager addStaticNatEntry request was sent successfully.
7. Application is notified of the result of the nat manager addStaticNatEntry request (either SUCCESS or FAILED) via the application-supplied callback.

3.5.5.7 Firewall Enablement in data Firewall manager call flow

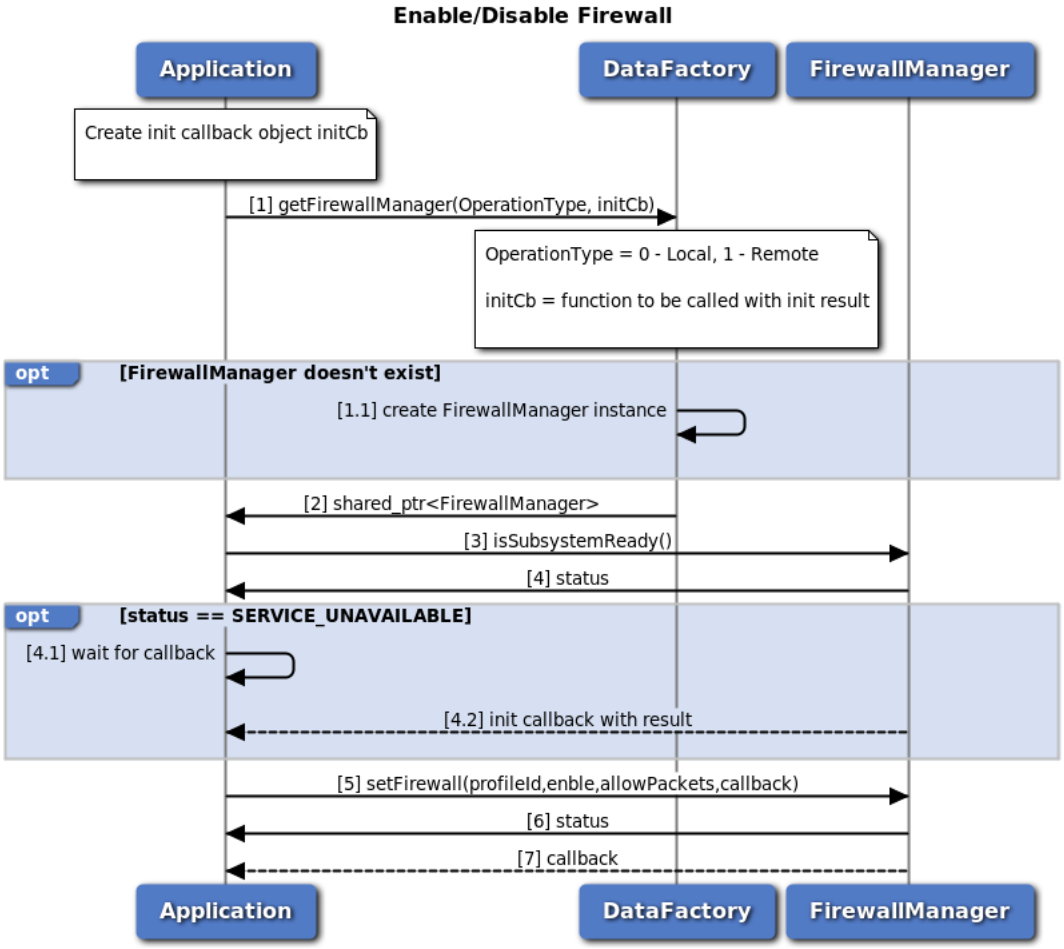

Figure 3-65 Firewall enablement in data Firewall manager call flow

1. Application requests data factory for data firewall manager object. 1.1. If firewall manager object does not exist, data factory will create new object.
2. Data factory returns shared pointer to firewall manager object to application.
3. Application request current service status of firewall manager returned by data factory.
4. Firewall manager returns current service status. 4.1 If status returned is SERVICE_UNAVAILABLE (manager is not ready), application should wait for init callback provided in step 1. 4.2 Firewall manager calls application callback with initialization result (success/failure).
5. On success, application calls firewall manager setFirewall with enable/disable and allow/drop packets.
6. Application receives synchronous Status which indicates if the firewall manager setFirewall request was sent successfully.
7. Application is notified of the Status of the firewall manager setFirewall request (either SUCCESS or FAILED) via the application-supplied callback.

3.5.5.8 Add Firewall Entry in data Firewall manager call flow

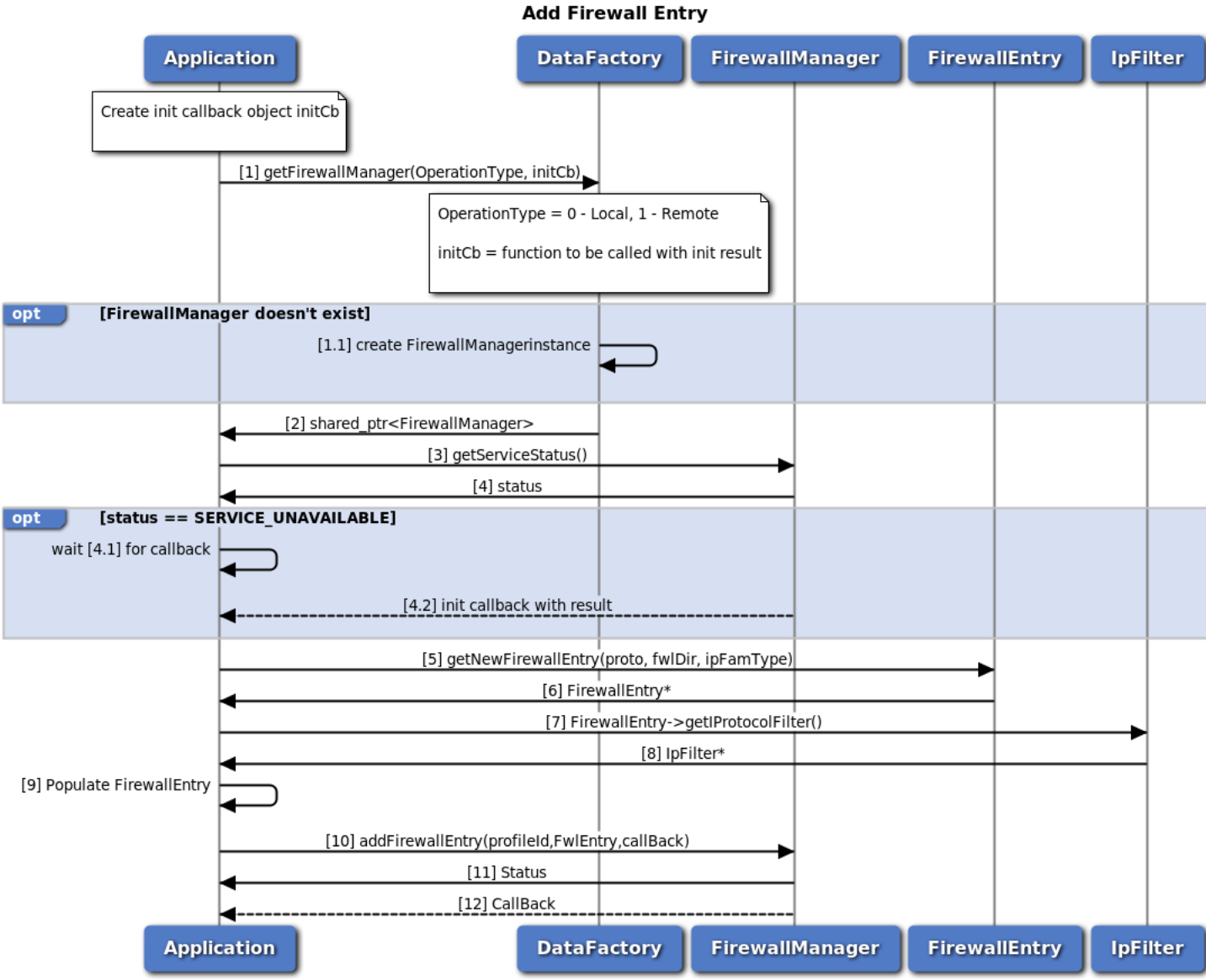

Figure 3-66 Add Firewall entry in data Firewall manager call flow

1. Application requests data factory for data firewall manager object. 1.1. If firewall manager object does not exist, data factory will create new object.
2. Data factory returns shared pointer to firewall manager object to application.
3. Application request current service status of data profile manager returned by data factory.
4. Firewall manager returns current service status. 4.1 If status returned is SERVICE_UNAVAILABLE (manager is not ready), application should wait for init callback provided in step 1. 4.2 Firewall manager calls application callback with initialization result (success/failure).
5. On success, application calls firewall manager getNewFirewallEntry to get FirewallEntry object.

6. Application receives Firewall Entry object.
7. Using Firewall Entry object, application calls IFirewallEntry::getIProtocolFilter to get protocol filter object
8. Application receives IpFilter object.
9. Application populates FirewallEntry and IpFilter objects.
10. Application calls IFirewallManager::addFirewallEntry with profileId and FirewallEntry to add firewall entry
11. Application receives synchronous Status which indicates if addFirewallEntry was sent successfully
12. Application is notified of the Status of the IFirewallManager::addFirewallEntry request (either SUCCESS or FAILED) via the application-supplied callback.

3.5.5.9 Set Firewall DMZ in data Firewall manager call flow

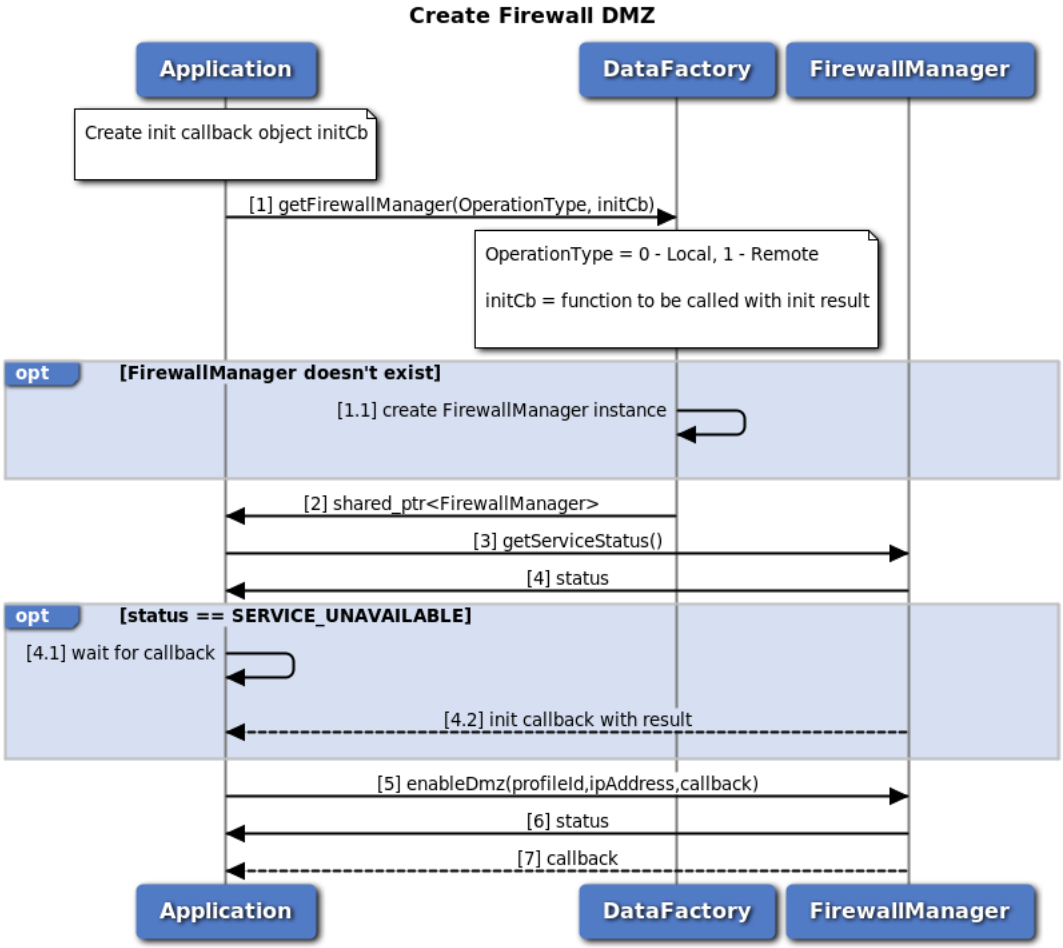

Figure 3-67 Set Firewall DMZ in data Firewall manager call flow

1. Application requests data factory for data firewall manager object. 1.1. If firewall manager object does not exist, data factory will create new object.
2. Data factory returns shared pointer to firewall manager object to application.

3. Application request current service status of firewall manager returned by data factory.
4. Firewall manager returns current service status. 4.1 If status returned is SERVICE_UNAVAILABLE (manager is not ready), application should wait for init callback provided in step 1. 4.2 Firewall manager calls application callback with initialization result (success/failure).
5. On success, application calls firewall manager enableDmz with profileId and IP Address.
6. Application receives synchronous Status which indicates if the firewall manager enableDmz request was sent successfully.
7. Application is notified of the Status of the firewall manager enableDmz request (either SUCCESS or FAILED) via the application-supplied callback.

3.5.5.10 Socks Enablement in data Socks manager call flow

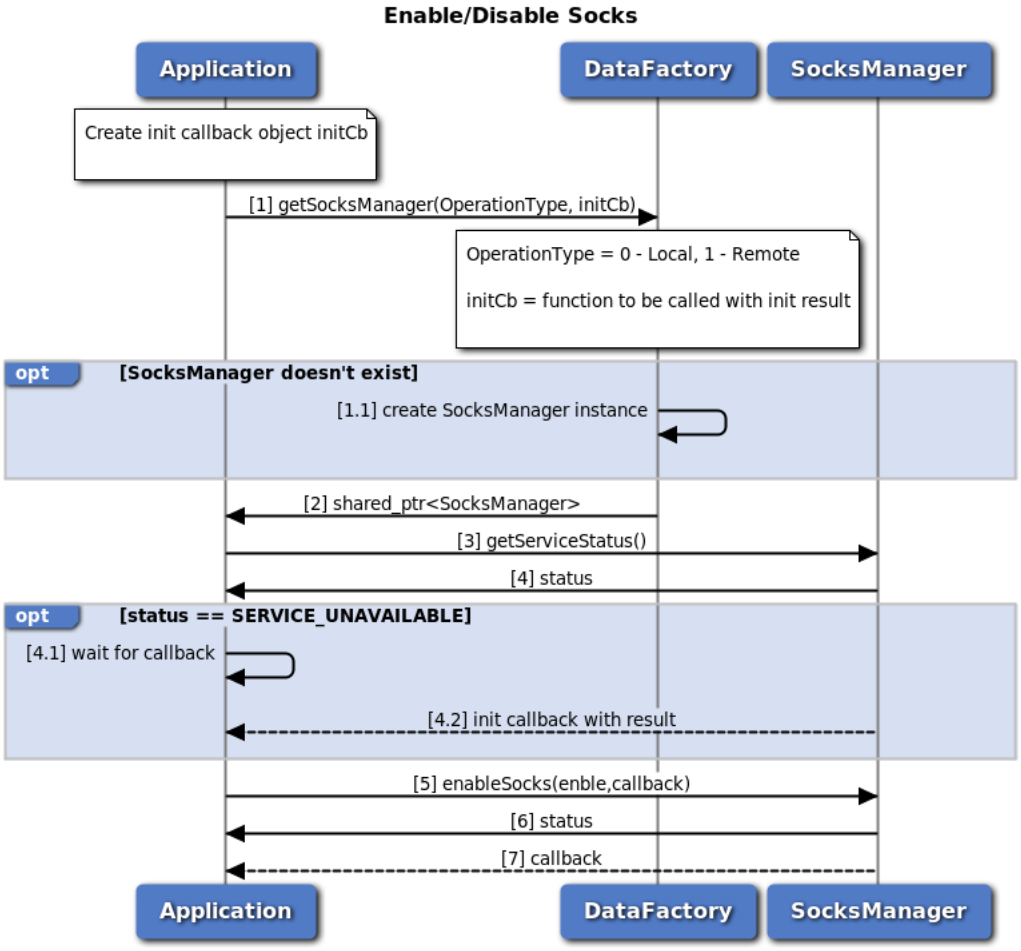

Figure 3-68 Socks enablement in data Socks manager call flow

1. Application requests data factory for data socks manager object. 1.1. If socks manager object does not exist, data factory will create new object.
2. Data factory returns shared pointer to socks manager object to application.
3. Application request current service status of socks manager returned by data factory.

4. Socks manager returns current service status. 4.1 If status returned is SERVICE_UNAVAILABLE (manager is not ready), application should wait for init callback provided in step 1. 4.2 Socks manager calls application callback with initialization result (success/failure).
5. On success, application calls socks manager enableSocks with enable/disable.
6. Application receives synchronous Status which indicates if the socks manager enableSocks request was sent successfully.
7. Application is notified of the Status of the socks manager enableSocks request (either SUCCESS or FAILED) via the application-supplied callback.

3.5.5.11 L2TP Enablement and Configuration in data L2TP manager call flow

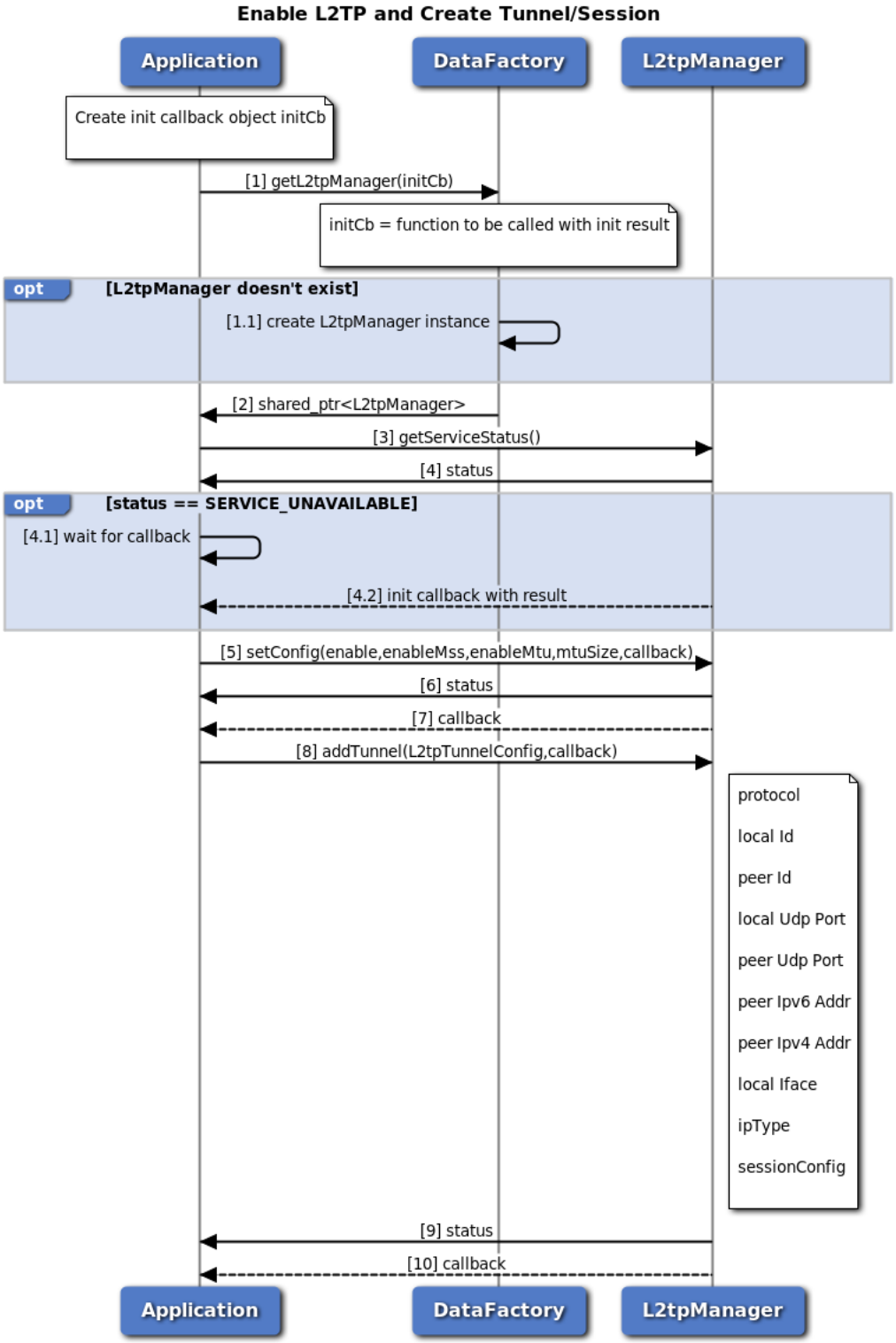

Figure 3-69 L2TP enablement and Configuration in data L2TP manager call flow

1. Application requests data factory for data l2tp manager object. 1.1. If l2tp manager object does not exist, data factory will create new object.

2. Data factory returns shared pointer to l2tp manager object to application.
3. Application request current service status of l2tp manager returned by data factory.
4. L2tp manager returns current service status. 4.1 If status returned is SERVICE_UNAVAILABLE (manager is not ready), application should wait for init callback provided in step 1. 4.2 L2tp manager calls application callback with initialization result (success/failure).
5. On success, application calls l2tp manager setConfig with enable/disable, enable/disable Mss, enable/disable MTU size and MTU size
6. Application receives synchronous Status which indicates if the l2tp manager setConfig request was sent successfully.
7. Application is notified of the Status of the l2tp manager setConfig request (either SUCCESS or FAILED) via the application-supplied callback.
8. Application calls l2tp manager setConfig with all required configurations to setup tunnel and session
9. Application receives synchronous Status which indicates if the l2tp manager setConfig request was sent successfully.
10. Application is notified of the Status of the l2tp manager setConfig request (either SUCCESS or FAILED) via the application-supplied callback.

3.5.5.12 Call flow to add and enable software bridge

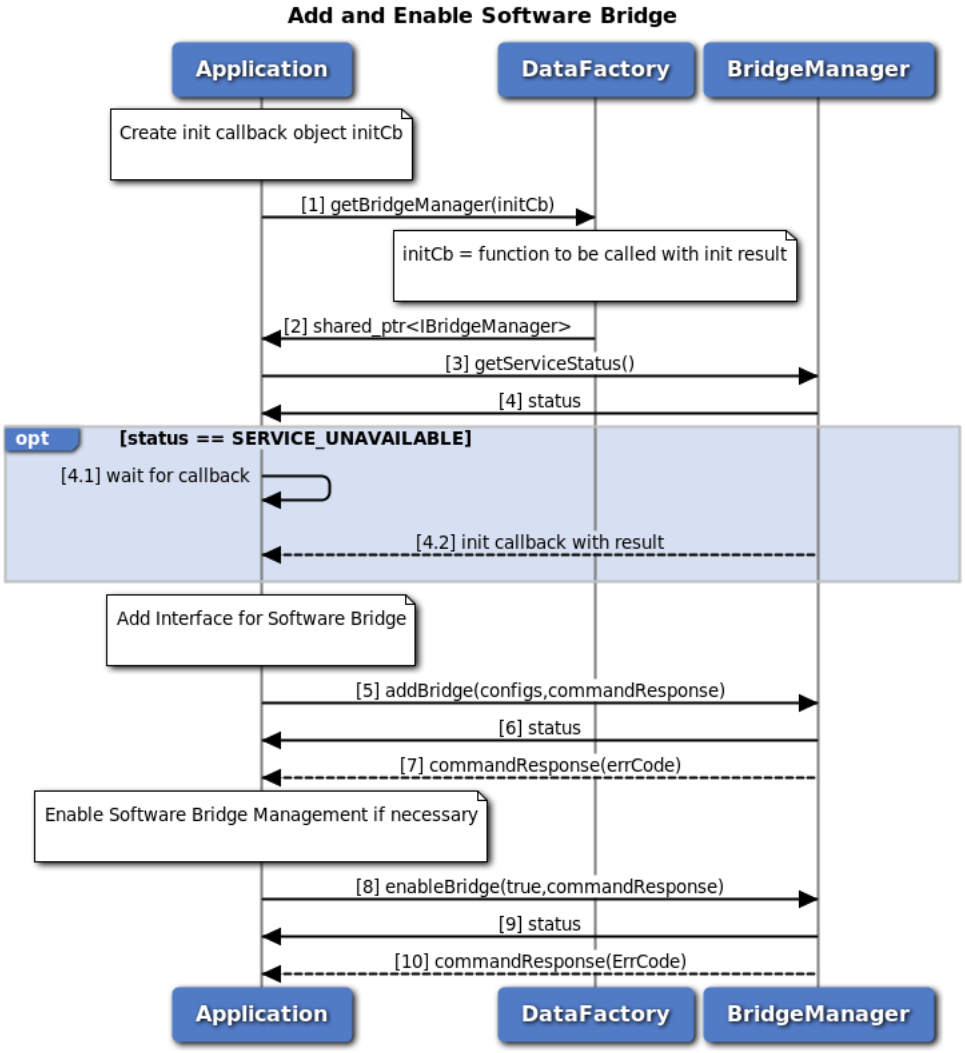

Figure 3-70 Call flow to add and enable a software bridge

1. Application requests data factory for bridge manager object.
2. Data factory returns shared pointer to bridge manager object to application.
3. Application request current service status of bridge manager returned by data factory.
4. Bridge manager returns current service status. 4.1 If status returned is SERVICE_UNAVAILABLE (manager is not ready), application should wait for init callback provided in step 1. 4.2 Bridge manager calls application callback with initialization result (success/failure).
5. On success, application requests to add software bridge configuration for an interface, providing an optional asynchronous response callback using addBridge API.
6. Application receives the synchronous status i.e. either SUCCESS or FAILED which indicates if the request was sent successfully.
7. Optionally, the application gets asynchronous response for addBridge via the application-supplied callback.

8. If the software bridge management is not enabled already, application requests to enable it, providing an optional asynchronous response callback using enableBridge API. Please note that this step affects all the software bridges configured in the system.
9. Application receives the status i.e. either SUCCESS or FAILED which indicates if the request was sent successfully.
10. Optionally, the application gets asynchronous response for enableBridge via the application-supplied callback.

3.5.5.13 Call flow to remove and disable software bridge

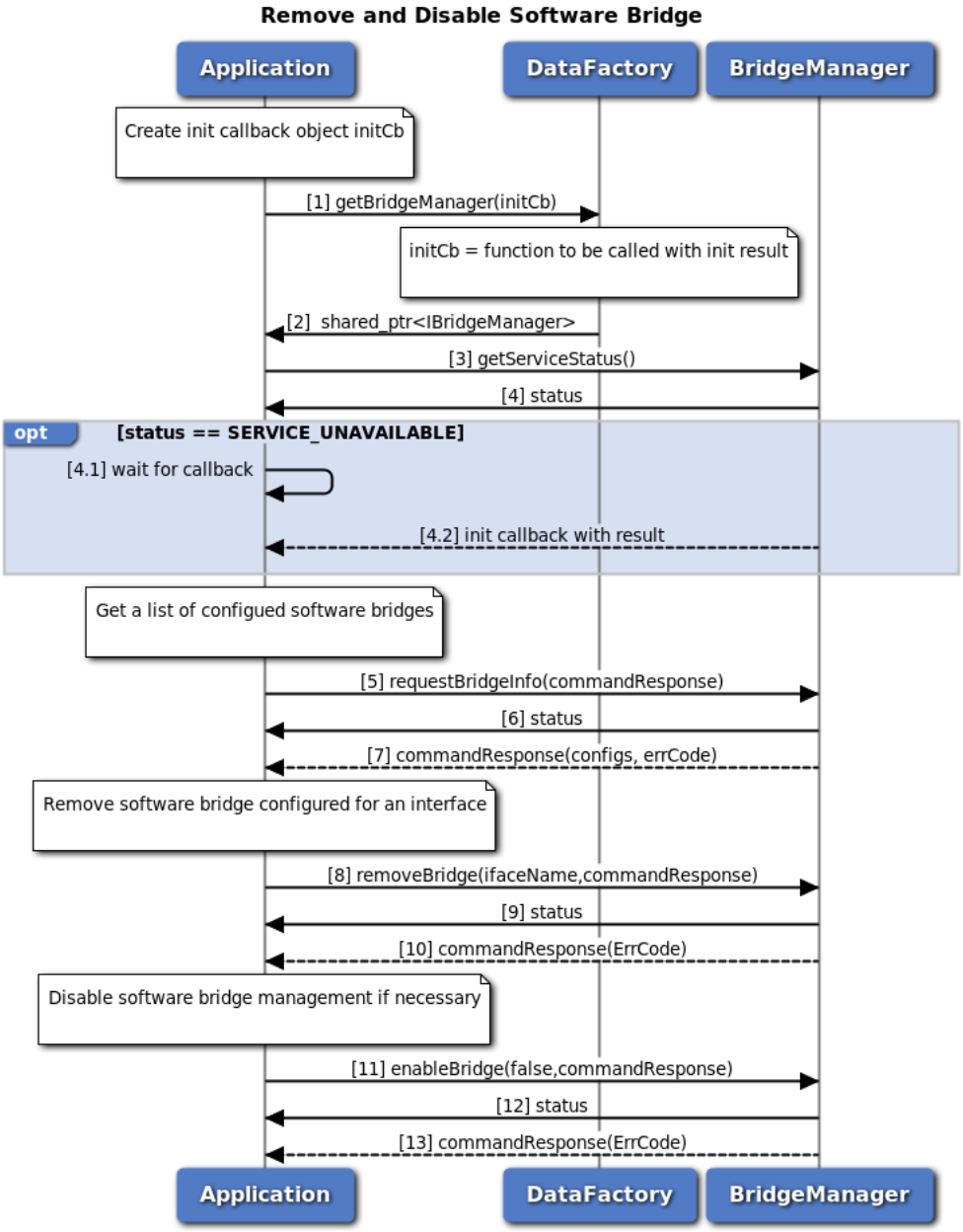

Figure 3-71 Call flow to remove and disable a software bridge

1. Application requests data factory for bridge manager object.
2. Data factory returns shared pointer to bridgeManager object to application.
3. Application request current service status of bridge manager returned by data factory.
4. Bridge manager returns current service status. 4.1 If status returned is SERVICE_UNAVAILABLE (manager is not ready), application should wait for init callback provided in step 1. 4.2 Bridge manager calls application callback with initialization result (success/failure).
5. On success, application requests to get the list of software bridge configurations, providing an asynchronous response callback using requestBridgeInfo API.
6. Application receives the synchronous status i.e. either SUCCESS or FAILED which indicates if the request was sent successfully.
7. The application gets asynchronous response for requestBridgeInfo via the application-supplied callback.
8. Application requests to remove software bridge configuration for an interface, providing an optional asynchronous response callback using removeBridge API.
9. Application receives the synchronous status i.e. either SUCCESS or FAILED which indicates if the request was sent successfully.
10. Optionally, the application gets asynchronous response for removeBridge via the application-supplied callback.
11. If the software bridge management needs to be disabled, application requests to disable it, providing an optional asynchronous response callback using enableBridge API. Please note that this step affects all the software bridges configured in the system.
12. Application receives the status i.e. either SUCCESS or FAILED which indicates if the request was sent successfully.
13. Optionally, the application gets asynchronous response for enableBridge via the application-supplied callback.

3.6 C-V2X

Applications need to have "radio" Linux group permissions to be able to operate successfully with underlying services.

3.6.1 Retrieve/Update C-V2X Configuration

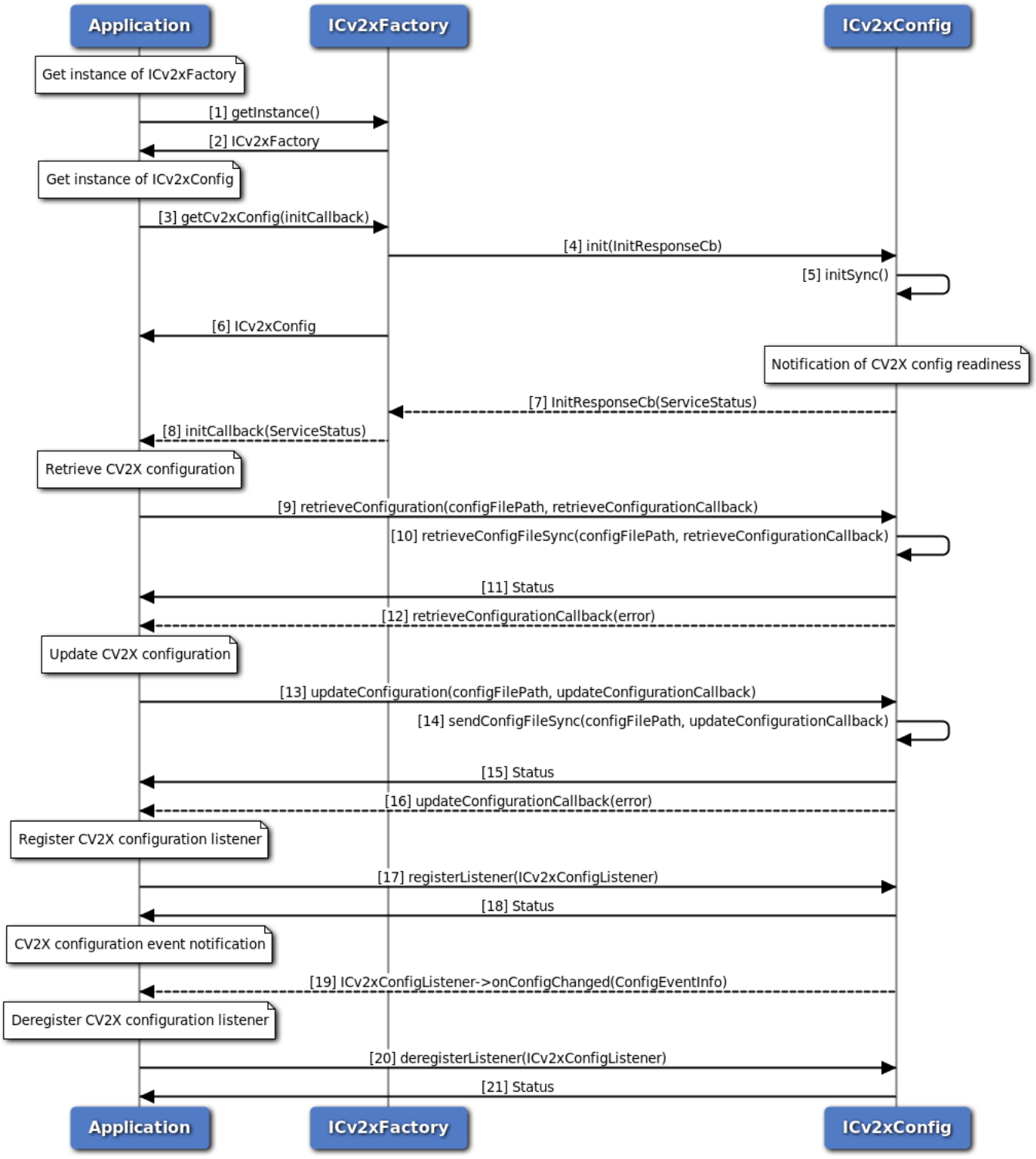

Figure 3-72 Retrieve/Update C-V2X Configuration Call Flow - C++ version

This call flow diagram describes the sequence of steps for retrieving or updating C-V2X configuration file using C++ version APIs.

1. Application requests for a ICv2xFactory instance.
2. Reference to singleton ICv2xFactory is returned to application.
3. Application requests C-V2X factory for a ICv2xConfig instance.
4. C-V2X factory creates ICv2xConfig object and calls init() method of ICv2xConfig.
5. C-V2X config starts initialization asynchronously.
6. C-V2X factory return ICv2xConfig object to application.
7. C-V2X factory is asynchronously notified of the readiness status of the C-V2X config via the initialization callback.
8. C-V2X factory calls application-supplied callback to notify the readiness status of C-V2X config (either SERVICE_AVAILABLE or SERVICE_FAILED). If the status is SERVICE_AVAILABLE, application can then request to retrieve or update C-V2X configuration.
9. Application requests to retrieve C-V2X configuration by calling retrieveConfiguration and supplying it with a path for the storing of config XML file.
10. C-V2X config sends request to modem and waits for response asynchronously.
11. Application receives synchronous status.
12. Application is asynchronously notified of the status of the request (either SUCCESS or FAILED) via the application-supplied callback.
13. Application requests to update C-V2X configuration by calling updateConfiguration and supplying it with a path to the new config XML file.
14. C-V2X config sends request to modem and waits for response asynchronously.
15. Application receives synchronous status.
16. Application is asynchronously notified of the status of the request (either SUCCESS or FAILED) via the application-supplied callback.
17. Application registers ICv2xConfigListener to get notification of C-V2X configuration events if needed.
18. Application receives synchronous status.
19. Application gets notification of C-V2X configuration events if the C-V2X configuration being used in the system has been changed or expired.
20. Application deregisters ICv2xConfigListener to stop listening to C-V2X configuration events.
21. Application receives synchronous status.

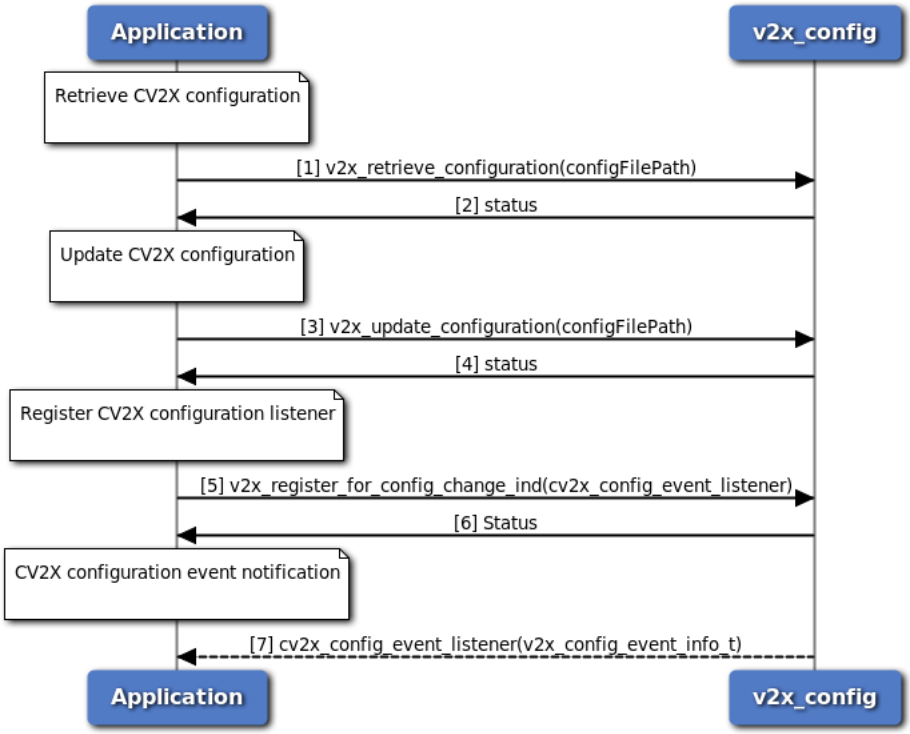

Figure 3-73 Retrieve/Update C-V2X Configuration Call Flow - C Version

This call flow diagram describes the sequence of steps for retrieving or updating C-V2X configuration file using C version APIs.

1. Application requests to retrieve C-V2X configuration by calling v2x_retrieve_configuration and supplying it with a path for the storing of config XML file.
2. Application receives synchronous status.
3. Application requests to update C-V2X configuration by calling v2x_update_configuration and supplying it with a path to the new config file.
4. Application receives synchronous status.
5. Application registers cv2x_config_event_listener to get notification of C-V2X configuration events if needed.
6. Application receives synchronous status.
7. Application gets notification of C-V2X configuration events if the C-V2X configuration being used in the system has been changed or expired.

3.6.2 Start/Stop C-V2X Mode

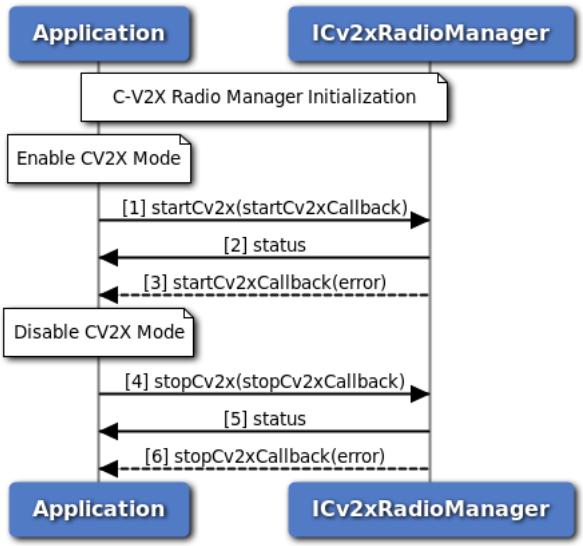

Figure 3-74 Start/Stop C-V2X Mode Call Flow - C++ Version

This call flow diagram describes the sequence of steps for starting or stopping C-V2X mode using C++ version APIs. Application must perform C-V2X radio manager initialization before calling any methods of ICv2xRadioManager. In normal operation, applications do not need to start or stop C-V2X mode. The system is configured by default to start C-V2X mode at boot. We include the call flow below for the sake of completeness.

1. Application requests to put modem into C-V2X mode using startCv2x method.
2. Application receives synchronous status which indicates if the start request was sent successfully.
3. Application is notified of the status of the start request (either SUCCESS or FAILED) via the application-supplied callback.
4. Application requests to disable C-V2X mode using stopCv2x method.
5. Application receives synchronous status which indicates if the stop request was sent successfully.
6. Application is asynchronously notified of the status of the stop request (either SUCCESS or FAILED) via the application-supplied callback.

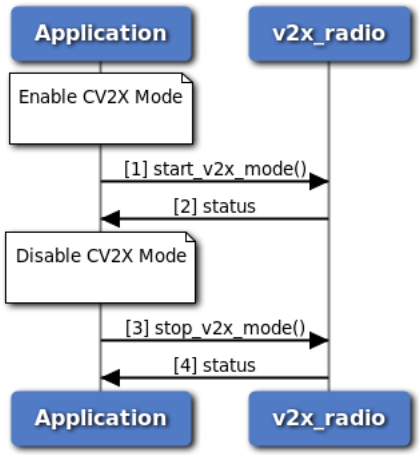

Figure 3-75 Start/Stop C-V2X Mode Call Flow - C Version

This call flow diagram describes the sequence of steps for starting or stopping C-V2X mode using C version APIs.

1. Application requests to put modem into C-V2X mode using start_v2x_mode method.
2. Application receives synchronous status which indicates if the operation was successful.
3. Application requests to disable C-V2X mode using stop_v2x_mode method.
4. Application receives synchronous status which indicates if the operation was successful.

3.6.3 C-V2X Radio Control Flow

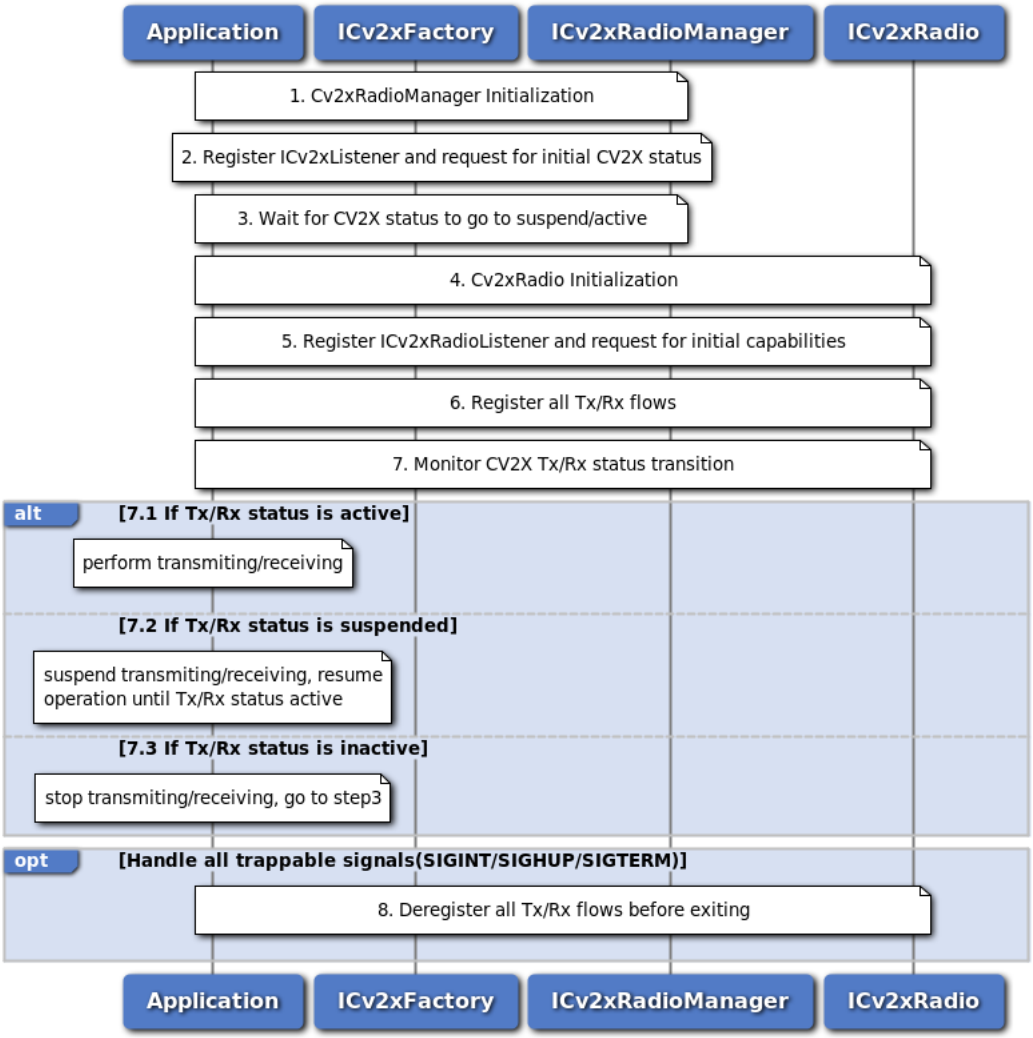

Figure 3-76 C-V2X Radio Control Flow - C++ Version

This call flow diagram describes the sequence of steps for overall C-V2X radio control flow using C++ version APIs.

1. Application performs C-V2X radio manager initialization and waits for the readiness. See steps 1∼8 in fig_cv2x_radio_initialization_callflow.
2. Application registers ICv2xListener to get C-V2X status update notification and requests for the initiate C-V2X status. See steps 1∼6 in fig_cv2x_get_status_callflow.
3. Application waits for C-V2X status to go to SUSPEND/ACTIVE.
4. Application performs C-V2X radio initialization and waits for the readiness. See steps 9∼17 in fig_cv2x_radio_initialization_callflow.
5. Application registers C-V2X radio listener to get C-V2X radio related notifications (L2 address update, SPS offset update, SPS scheduling update, C-V2X radio capabilities update) and requests for the initial C-V2X radio capabilities if needed. See steps 1∼6 in fig_cv2x_get_capabilities_callflow.

6. Application registers all Tx/Rx flows. See fig_cv2x_radio_rx_sub_callflow/fig_cv2x_radio_event_↩ flow_callflow/fig_cv2x_radio_sps_flow_callflow.
7. Application monitors C-V2X status change during operation.
   - 7.1 If C-V2X Tx/Rx status is active, application performs transmitting/receiving.
   - 7.2 If C-V2X Tx/Rx status goes to SUSPEND, application should suspend transmitting/receiving, log the event and resume operation when C-V2X Tx/Rx status goes to ACTIVE again.
   - 7.3 If C-V2X Tx/Rx status goes to INACTIVE, application should stop transmitting/receiving, log the event and go back to step 3 for recovery.
8. Application should handle all trappable signals like SIGINT/SIGHUP/SIGTERM, all Tx/Rx flows must be deregistered before exiting.

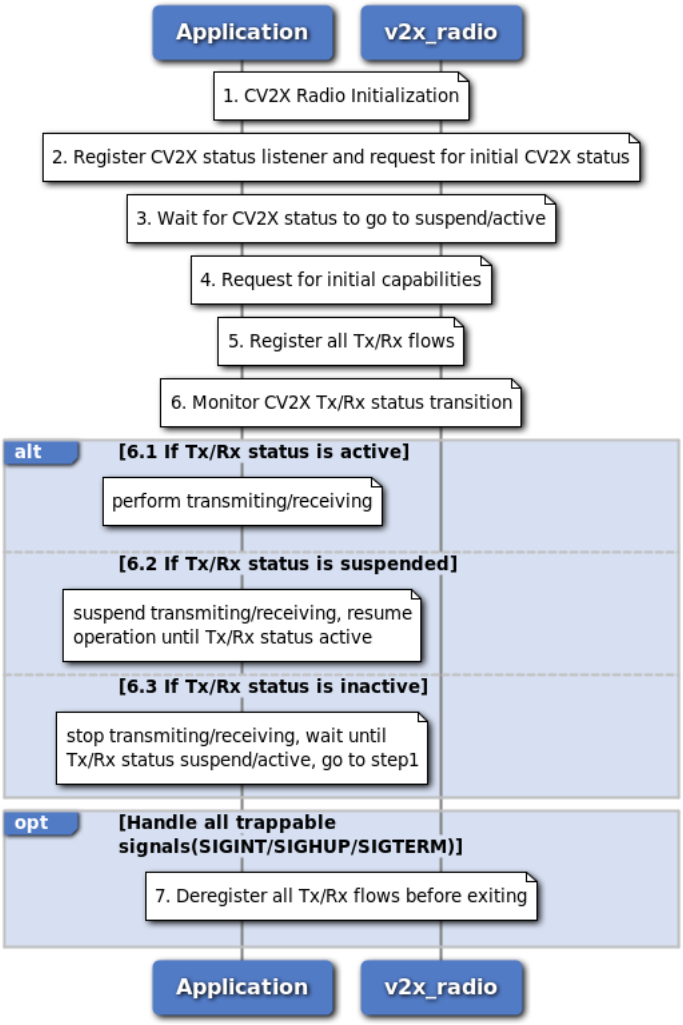

Figure 3-77 C-V2X Radio Control Flow - C Version

This call flow diagram describes the sequence of steps for overall C-V2X radio control flow using C version APIs.

1. Application performs C-V2X radio initialization and waits for the readiness in application provided

callback. See fig_cv2x_radio_initialization_callflow_c. Application should check the C-V2X radio initialization status, retry or exit if failing.

2. Application registers v2x_ext_radio_status_listener to get C-V2X status update notification and requests for the initiate C-V2X status. See steps 1∼5 in fig_cv2x_get_status_callflow_c.
3. Application waits for C-V2X status to go to SUSPEND/ACTIVE.
4. Application requests for the initial C-V2X radio capabilities if needed. See fig_cv2x_get_capabilities_callflow_c.
5. Application registers all Tx/Rx flows. See fig_cv2x_radio_rx_sub_callflow_c/fig_cv2x_radio_↩ event_flow_callflow_c/fig_cv2x_radio_sps_flow_callflow_c.
6. Application monitors C-V2X status change during operation.
   - 6.1 If C-V2X Tx/Rx status is active, application performs transmitting/receiving.
   - 6.2 If C-V2X Tx/Rx status goes to SUSPEND, application should suspend transmitting/receiving, log the event and resume operation when C-V2X Tx/Rx status goes to ACTIVE again.
   - 6.3 If C-V2X Tx/Rx status goes to INACTIVE, application should stop transmitting/receiving, log the event, wait for C-V2X Tx/Rx status to go to SUSPEND or ACTIVE and then go back to step 1 for recovery.
7. Application should handle all trappable signals like SIGINT/SIGHUP/SIGTERM, all Tx/Rx flows must be deregistered before exiting.

3.6.4 C-V2X Radio Initialization

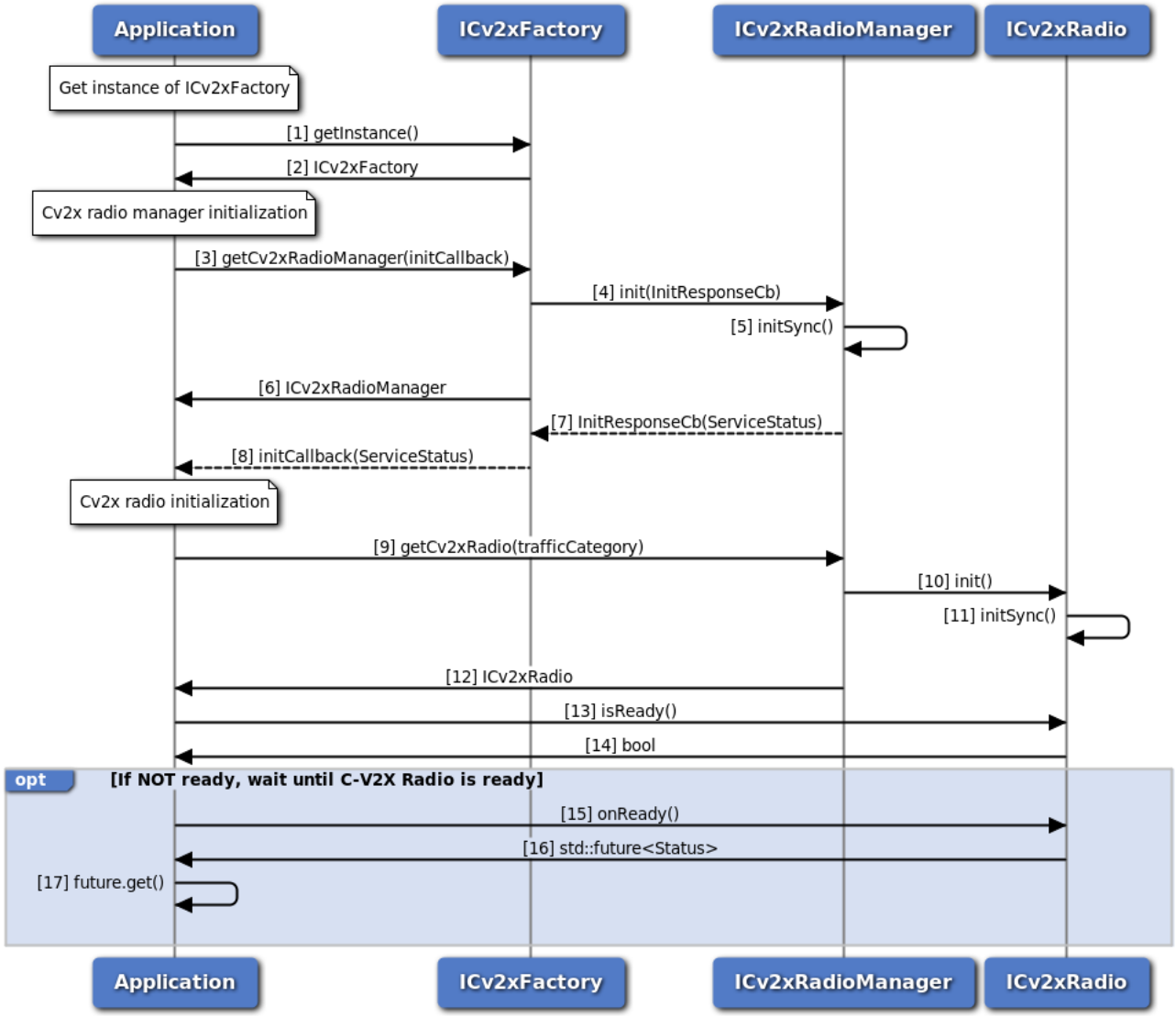

Figure 3-78 C-V2X Radio Initialization Call Flow - C++ Version

This call flow diagram describes the sequence of steps for initializing the ICv2xRadioManager and the ICv2xRadio object using C++ version APIs. Applications must initialize ICv2xRadioManager/ICv2xRadio object and wait for the readiness before calling any other methods on the objects.

1. Application requests for a ICv2xFactory instance.
2. Reference to singleton ICv2xFactory is returned to application.
3. Application requests C-V2X factory for an ICv2xRadioManager instance.
4. C-V2X factory creates ICv2xRadioManager object and calls init() method of ICv2xRadioManager.
5. C-V2X radio manager starts initialization asynchronously.
6. C-V2X factory returns ICv2xRadioManager object to application.

7. C-V2X factory is asynchronously notified of the readiness status of the C-V2X radio manager via the initialization callback.
8. C-V2X factory calls application-supplied callback to notify the readiness status of C-V2X radio manager (either SERVICE_AVAILABLE or SERVICE_FAILED).
9. Application requests C-V2X Radio from ICv2xRadioManager.
10. C-V2X radio manager creates ICv2xRadio object and calls method init().
11. C-V2X radio starts initialization asynchronously.
12. C-V2X radio manager returns ICv2xRadio object to application.
13. Application queries C-V2X radio's readiness state using isReady() method.
14. C-V2X radio returns a bool indicating whether it is ready to be used.
15. If C-V2X radio is not ready, the application calls onReady() method.
16. C-V2X radio returns a future object.
17. Application calls future's get() method and blocks until C-V2X radio has completed its initialization steps. The return value of get() indicates the status of the initialization (either SUCCESS or FAILED).

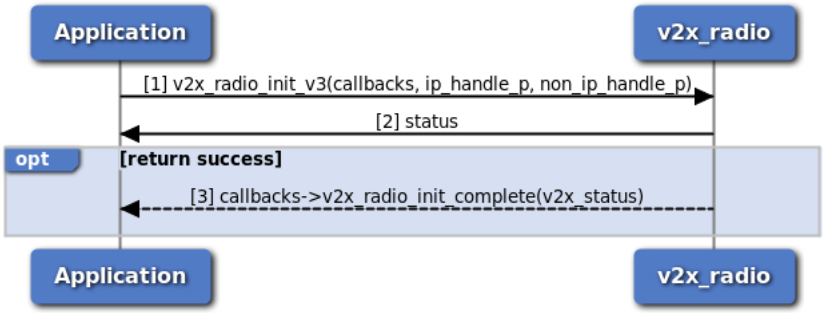

Figure 3-79 C-V2X Radio Initialization Call Flow - C Version

This call flow diagram describes the sequence of steps for initialing C-V2X radio using C version APIs. Applications must initialize C-V2X radio and wait for the readiness before calling any other methods of C-V2X radio.

1. Application calls v2x_radio_init_v3 to initialize C-V2X radio manager and radio, provides callback functions as needed. Callback function v2x_radio_init_complete is mandatory for getting the C-V2X radio initialization status.
2. Application gets return value (0 on success or negative value on error). If the return value is success, the handles of C-V2X IP and non-IP radio interface are provided via the application-supplied pointers ip_handle_p and non_ip_handle_p. The interface handles can be used to specify IP or non-IP traffic type when registering C-V2X radio Tx/Rx flows.
3. Application gets the notification of C-V2X radio initialization status (V2X_STATUS_SUCCESS or V2X_STATUS_FAIL) via the application-supplied v2x_radio_init_complete callback function if the return value of v2x_radio_init_v3 is successful.

3.6.5 Get C-V2X Status

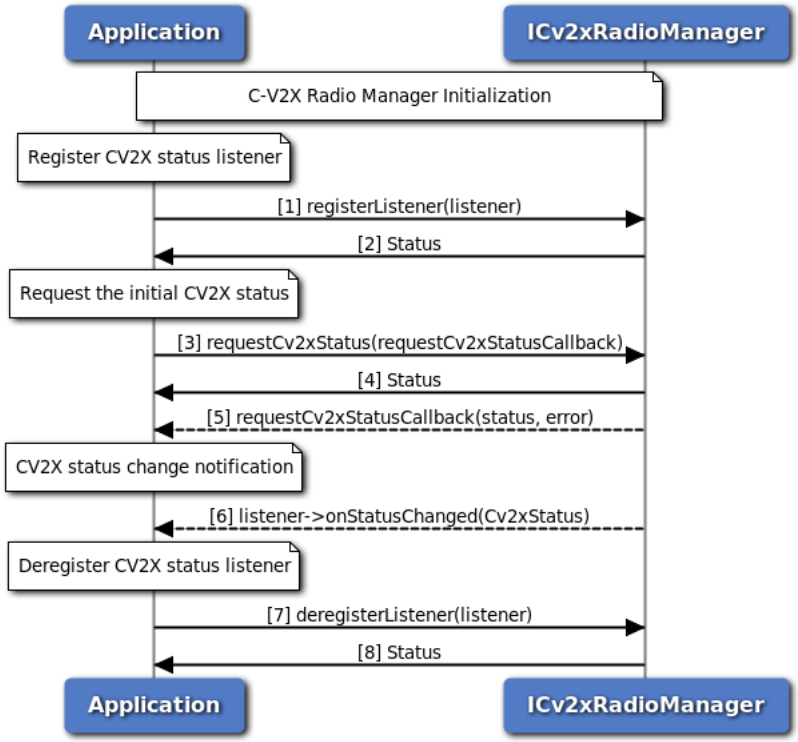

Figure 3-80 Get C-V2X Status Call Flow - C++ Version

This call flow diagram describes the sequence of steps for getting C-V2X radio status using C++ version APIs. Application must perform C-V2X radio manager initialization before calling any methods of ICv2xRadioManager.

1. Application registers a listener for getting notifications of C-V2X status update.
2. Status of register listener (either SUCCESS or FAILED) will be returned to the application.
3. Application requests the initial C-V2X status using requestCv2xStatus method.
4. Application receives synchronous status (either SUCCESS or FAILED) which indicates if the request was sent successfully.
5. Application is asynchronously notified of the status of the request (either SUCCESS or FAILED) via the application-supplied callback. If success, current C-V2X status is supplied via callback.
6. Application gets notification of C-V2X status update via method onStatusChanged of ICv2xRadioListener.
7. Application deregister the listener.
8. Status of deregistering listener (either SUCCESS or FAILED) will be returned to the application.

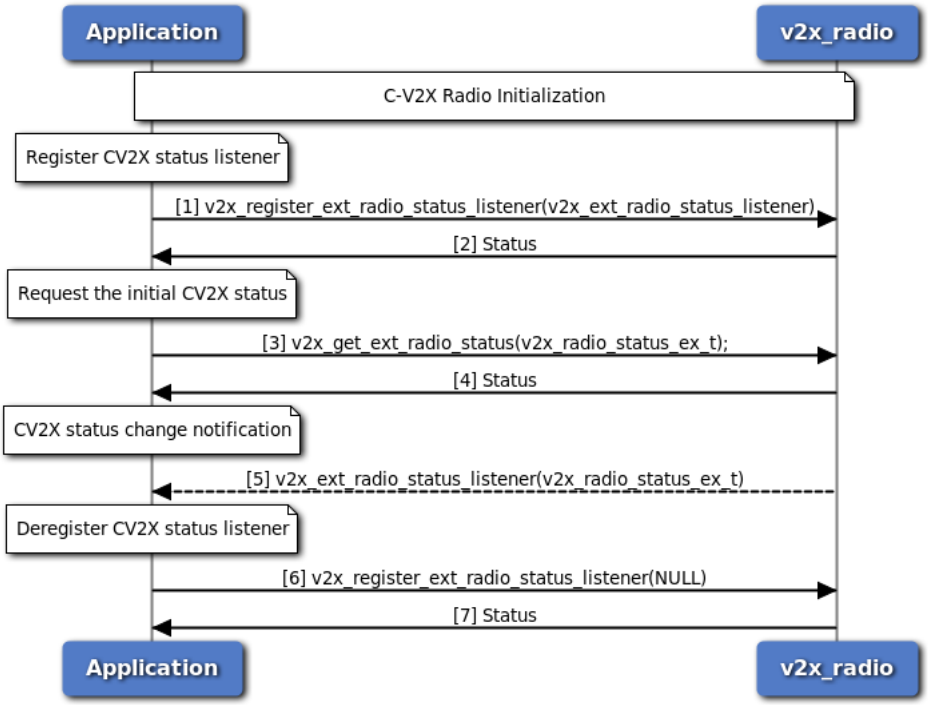

Figure 3-81 Get C-V2X Status Call Flow - C Version

This call flow diagram describes the sequence of steps for getting C-V2X radio status using C version APIs. Application must perform C-V2X radio initialization before calling any methods of C-V2X radio.

1. Application registers a listener for getting notifications of C-V2X status update.
2. Application gets the return value (V2X_STATUS_SUCCESS or V2X_STATUS_FAIL) which indicates if the operation was successfully performed. If it succeeded, the C-V2X overall status and per pool status are provided via the application-supplied pointer.
3. Application requests the initial C-V2X status using v2x_get_ext_radio_status method.
4. Application gets the return value (V2X_STATUS_SUCCESS or V2X_STATUS_FAIL) which indicates if the operation was successfully performed.
5. Application gets notification of C-V2X status update.
6. Application deregister the listener via setting a NULL listener.
7. Application gets the return value (V2X_STATUS_SUCCESS or V2X_STATUS_FAIL) which indicates if the operation was successfully performed.

3.6.6 Get C-V2X Capabilities

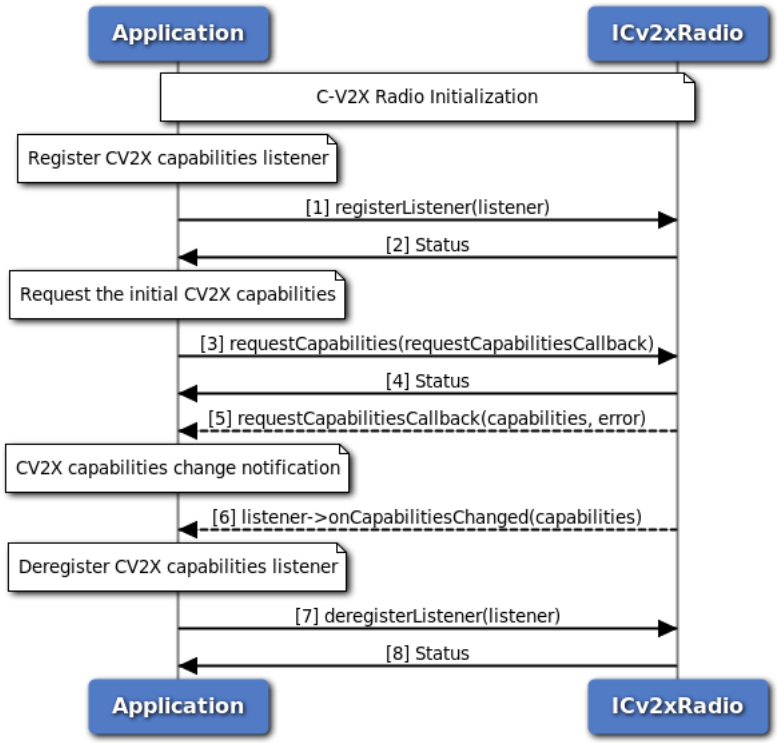

Figure 3-82 Get C-V2X Capabilities Call Flow - C++ Version

This call flow diagram describes the sequence of steps for getting C-V2X radio capabilities using C++ version APIs. Application must perform C-V2X radio initialization before calling any methods of ICv2xRadio.

1. Application registers a listener for getting notifications of C-V2X capabilities update.
2. Status of register listener (either SUCCESS or FAILED) will be returned to the application.
3. Application requests the initial C-V2X capabilities using requestCapabilities method.
4. Application receives synchronous status (either SUCCESS or FAILED) which indicates if the request was sent successfully.
5. Application is asynchronously notified of the status of the request (either SUCCESS or FAILED) via the application-supplied callback. If success, current C-V2X capabilities is supplied via callback.
6. Application gets notification of C-V2X capabilities update via method onCapabilitiesChanged of ICv2xRadioListener.
7. Application deregister the listener.
8. Status of deregistering listener (either SUCCESS or FAILED) will be returned to the application.

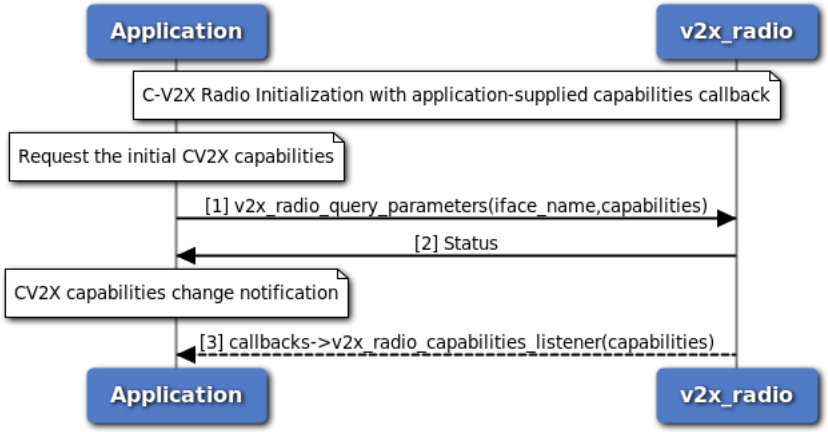

Figure 3-83 Get C-V2X Capabilities Call Flow - C Version

This call flow diagram describes the sequence of steps for getting C-V2X radio capabilities using C version APIs. Application must perform C-V2X radio initialization and provide C-V2X capabilities callback to get C-V2X radio capabilities update notification.

1. Application requests the initial C-V2X capabilities using v2x_radio_query_parameters method.
2. Application gets the return value (V2X_STATUS_SUCCESS or V2X_STATUS_FAIL) which indicates if the operation was successfully performed. If it succeeded, C-V2X capabilities are provided via the application-supplied pointer.
3. Application gets notification of C-V2X capabilities update via application-supplied callback v2x_radio_capabilities_listener.

3.6.7 C-V2X Radio RX Subscription

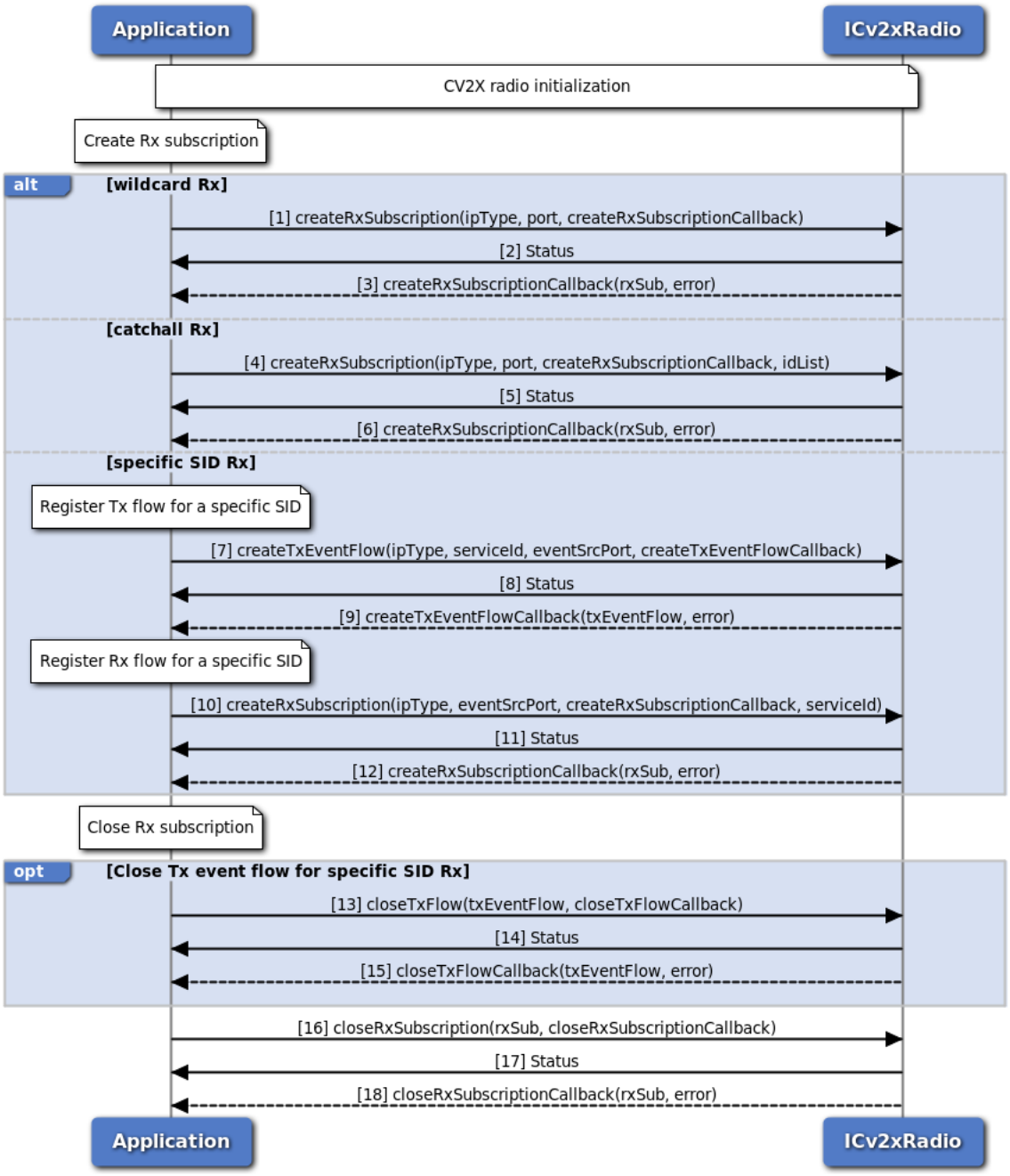

Figure 3-84 C-V2X Radio RX Subscription Call Flow - C++ Version

This call flow diagram describes the sequence of steps for registering or deregistering C-V2X Rx flows using C++ version APIs. Application must perform C-V2X radio initialization before calling any methods of ICv2xRadio.

There are three Rx modes supported for non-IP traffic,

- Wildcard Rx: Receive all service IDs on a single port. This mode could break catchall Rx and specific service ID Rx.

- Catchall Rx: Receive the specified service IDs on a single port. This mode can be used for Rx-only devices for which no Tx flow need to be created.
- Specific service ID Rx: Receive a specified service ID on a single port. Application can receive different service IDs on different ports by registering a pair of Tx and Rx flows for each service ID.

NOTE: For catchall or specific service ID Rx mode, filtering for received broadcast packets is implemented in low layer based on the service ID and its mapped destination L2 address, assuming that each service ID is mapped to a unique destination L2 address for all Tx and Rx devices. If this assumption is not valid, for example, in ETSI standards all broadcast service IDs are using same destination L2 address 0xFFFFFFFF, then application should choose wildcard Rx method and do filtering in application layer.

1. Application requests to enable wildcard Rx mode by specifying no service ID when creating Rx flow.
2. Application receives synchronous status (either SUCCESS or FAILED) indicating whether the request for creating Rx flow was sent successfully.
3. C-V2X radio sends asynchronous notification via the callback function on the status of the request for creating Rx flow. If SUCCESS, the RX flow is returned in the callback.
4. Application requests to enable catchall Rx mode by specifying a valid service ID list when creating Rx flow.
5. Application receives synchronous status (either SUCCESS or FAILED) indicating whether the request for creating Rx flow was sent successfully.
6. C-V2X radio sends asynchronous notification via the callback function on the status of the request for creating Rx flow. If SUCCESS, the RX flow is returned in the callback.
7. Application requests to create a Tx event flow before creating Rx flow to enable specific service ID Rx mode. The service ID and the port number of Tx and Rx flow should be the same.
8. Application receives synchronous status (either SUCCESS or FAILED) indicating whether the request for creating Tx flow was sent successfully.
9. C-V2X radio sends asynchronous notification via the callback function on the status of creating Tx flow. If SUCCESS, the TX event flow is returned in the callback.
10. Application requests to enable specific service ID Rx mode if the corresponding Tx flow has been registered successfully.
11. Application receives synchronous status (either SUCCESS or FAILED) indicating whether the request for creating Rx flow was sent successfully.
12. C-V2X radio sends asynchronous notification via the callback function on the status of the request for creating Rx flow. If SUCCESS, the RX flow is returned in the callback.
13. Application requests to closes the Tx event flow registered for specific service ID Rx mode.
14. Application receives synchronous status (either SUCCESS or FAILED) indicating whether the request for closing Tx flow was sent successfully.
15. C-V2X radio sends asynchronous notification via the callback function on the status of closing Tx flow.
16. Application requests to close the RX flow.
17. Application receives synchronous status (either SUCCESS or FAILED) indicating whether the request for closing Rx flow was sent successfully.
18. C-V2X radio sends asynchronous notification via the callback (if a callback was specified) indicating

the status of closing Rx flow.

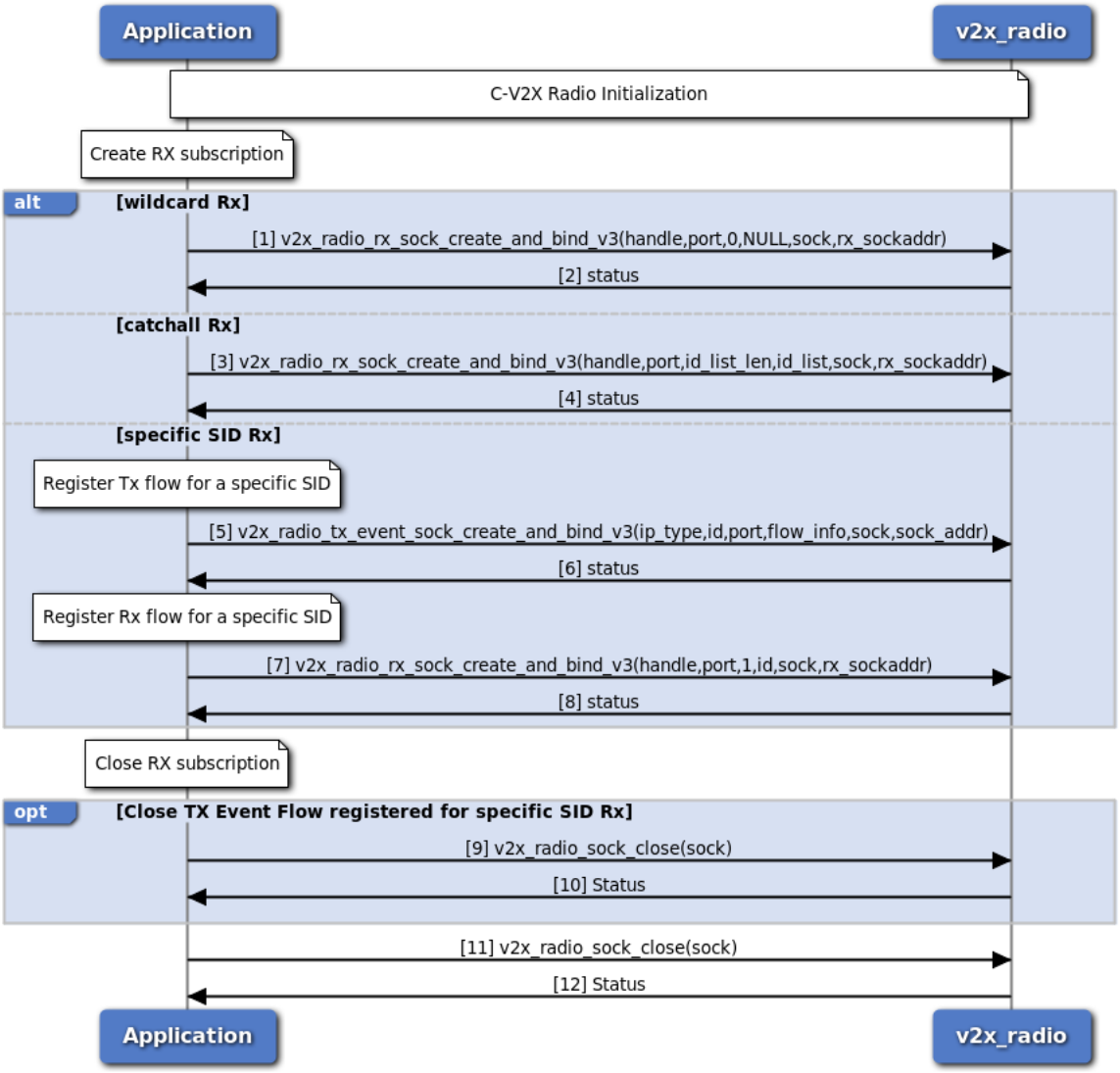

Figure 3-85 C-V2X Radio RX Subscription Call Flow - C Version

This call flow diagram describes the sequence of steps for registering or deregistering C-V2X Rx flows using C version APIs. Application must perform C-V2X radio initialization before calling any methods of C-V2X radio.

1. Application requests to enable wildcard Rx mode by specifying no service ID when creating Rx flow.
2. Application receives synchronous status (either 0 on SUCCESS or negative values if FAILED) indicating whether the Rx flow creation was successful.
3. Application requests to enable catchall Rx mode by specifying a valid service ID list when creating Rx flow.
4. Application receives synchronous status (either 0 on SUCCESS or negative values if FAILED) indicating whether the Rx flow creation was successful.
5. Application registers a Tx event flow before creating Rx flow to enable specific service ID Rx mode. The service ID and the port number of Tx and Rx flow should be the same.

6. Application receives synchronous status (either 0 on SUCCESS or negative values if FAILED) indicating whether the Tx event flow registration was successful.
7. Application requests to enable specific service ID Rx mode if the corresponding Tx flow has been registered successfully.
8. Application receives synchronous status (either 0 on SUCCESS or negative values if FAILED) indicating whether the Rx flow creation was successful.
9. Application requests to closes the Tx event flow registered for specific service ID Rx mode.
10. Application receives synchronous status (either 0 on SUCCESS or negative values if FAILED) indicating whether the Tx flow has been closed successfully.
11. Application requests to close the Rx flow.
12. Application receives synchronous status (either 0 on SUCCESS or negative values if FAILED) indicating whether the Rx flow has been closed successfully.

3.6.8 C-V2X Radio TX Event Flow

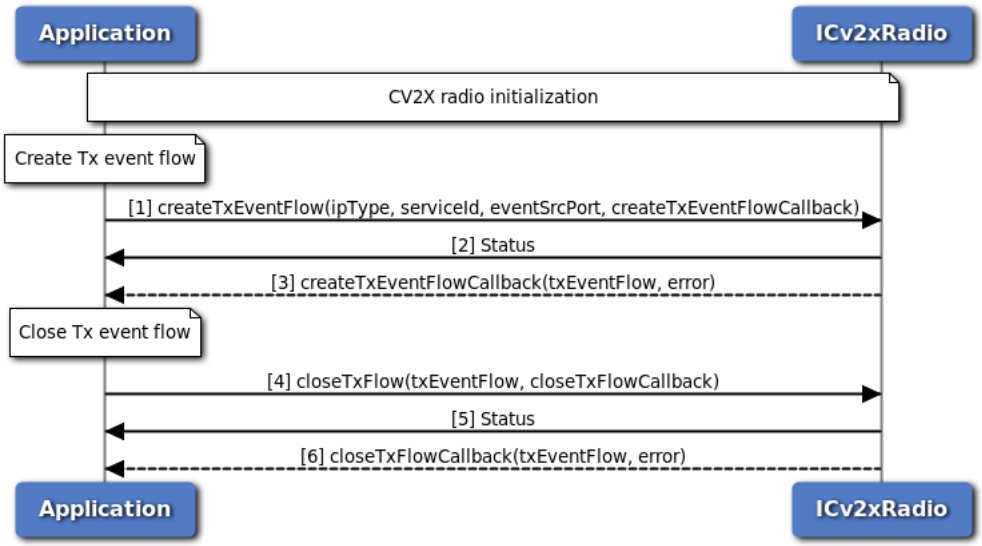

Figure 3-86 C-V2X Radio TX Event Flow Call Flow - C++ Version

This call flow diagram describes the sequence of steps for registering or deregistering C-V2X Tx event flows using C++ version APIs. Application must perform C-V2X radio initialization before calling any methods of ICv2xRadio.

1. Application requests a new TX event flow from the C-V2X radio using createTxEventFlow method.
2. Application receives synchronous status (either SUCCESS or FAILED) indicating whether the request was sent successfully.
3. C-V2X radio sends asynchronous notification via the callback function on the status of the request. If SUCCESS, the TX event flow is returned in the callback.
4. Application requests to close the TX event flow.
5. Application receives synchronous status (either SUCCESS or FAILED) indicating whether the request was sent successfully.

6. C-V2X radio sends asynchronous notification via the callback (if a callback was specified) indicating the status of the request.

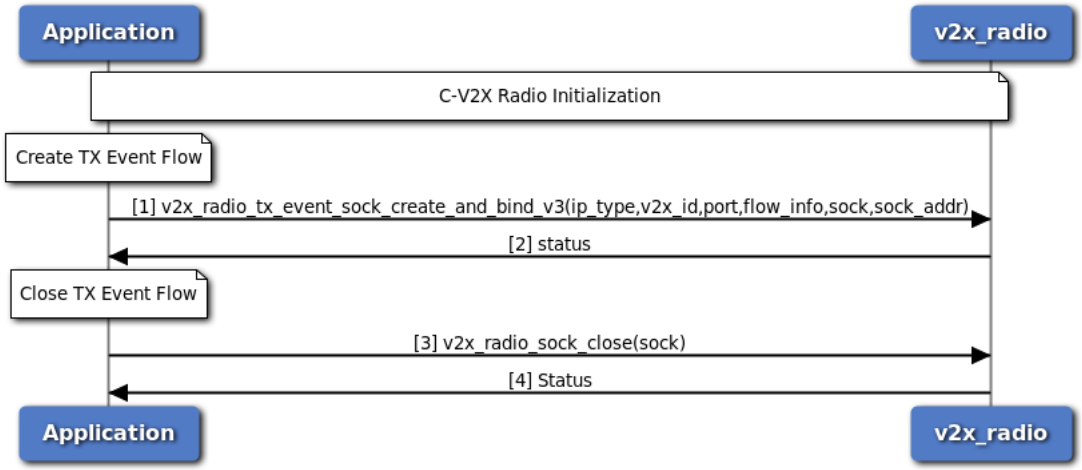

Figure 3-87 C-V2X Radio TX Event Flow Call Flow - C++ Version

This call flow diagram describes the sequence of steps for registering or deregistering C-V2X Tx event flows using C version APIs. Application must perform C-V2X radio initialization before calling any methods of C-V2X radio.

1. Application requests a new TX event flow from the C-V2X radio using v2x_radio_tx_event_sock_create_and_bind_v3 method and specifies the IP or non-IP traffic type, C-V2X service ID, port number, event flow information and pointers that that point to the created socket and socket address on success.
2. Application receives synchronous status (either 0 on SUCCESS or negative values if FAILED) indicating whether the operation was successfully performed. If the return value is 0, application gets the socket value from the application-supplied pointer and then starts sending C-V2X data packets using the socket.
3. Application requests to close the RX subscription by calling method v2x_radio_sock_close and supplying with a pointer that points to the TX socket created.
4. Application receives synchronous status (either 0 on SUCCESS or negative values if FAILED) indicating whether the operation was successfully performed

3.6.9 C-V2X Radio TX SPS flow

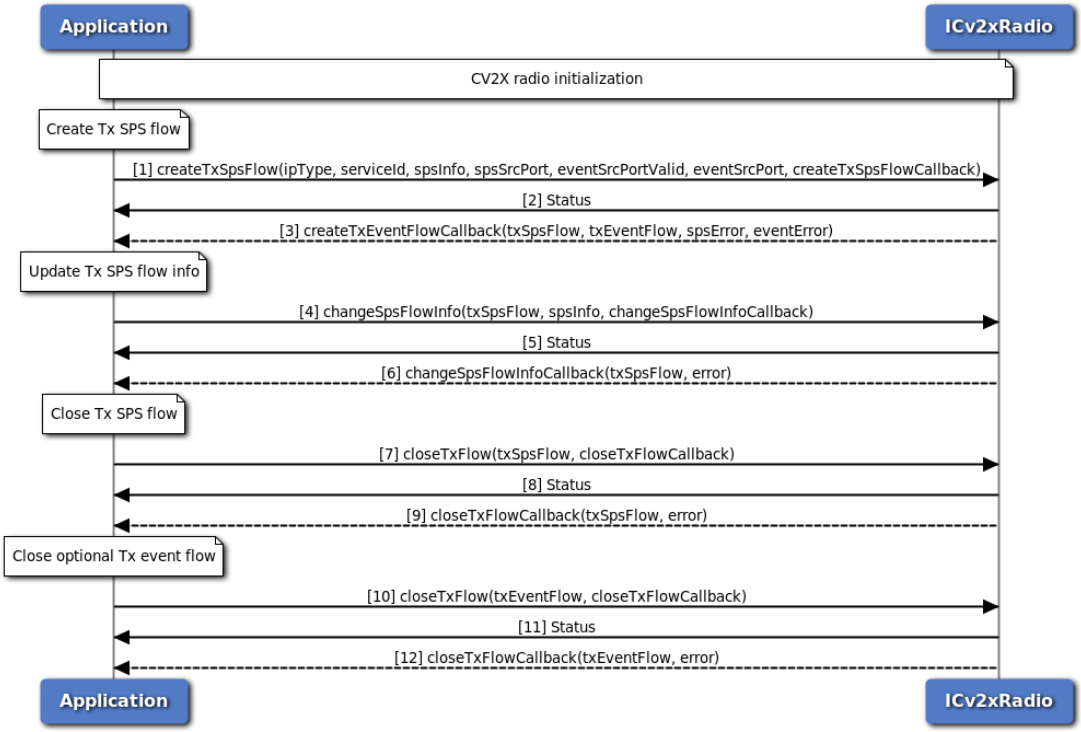

Figure 3-88 C-V2X Radio SPS Flow Call Flow - C++ Version

This call flow diagram describes the sequence of steps for registering or deregistering C-V2X Tx SPS flows using C++ version APIs. Application must perform C-V2X radio initialization before calling any methods of ICv2xRadio. Only 2 Tx SPS flows are allowed in maximum in the system.

1. Application requests a new TX SPS flow from the C-V2X radio using createTxSPSFlow method. The application can also specify an optional Tx event flow.
2. Application receives synchronous status (either SUCCESS or FAILED) indicating whether the request was sent successfully.
3. C-V2X radio sends asynchronous notification via the callback function. The callback will return the Tx SPS flow and its status as well as the optional Tx event flow and its status.
4. Application requests to change the SPS parameters using the changeSpsFlowInfo method.
5. Application received synchronous status (either SUCCESS or FAILED) indicating whether the request was sent successfully.
6. C-V2X radio sends asynchronous notification via the callback (if callback was specified) indicating the status of the request.
7. Application requests to close the SPS flow.
8. Application receives synchronous status (either SUCCESS or FAILED) indicating whether the request was sent successfully.
9. C-V2X radio sends asynchronous notification via the callback (if a callback was specified) indicating the status of the request.

10. Application requests to close optional Tx event flow (if one was created).
11. Application receives synchronous status (either SUCCESS or FAILED) indicating whether the request was sent successfully.
12. C-V2X radio sends asynchronous notification via the callback (if a callback was specified) indicating the status of the request.

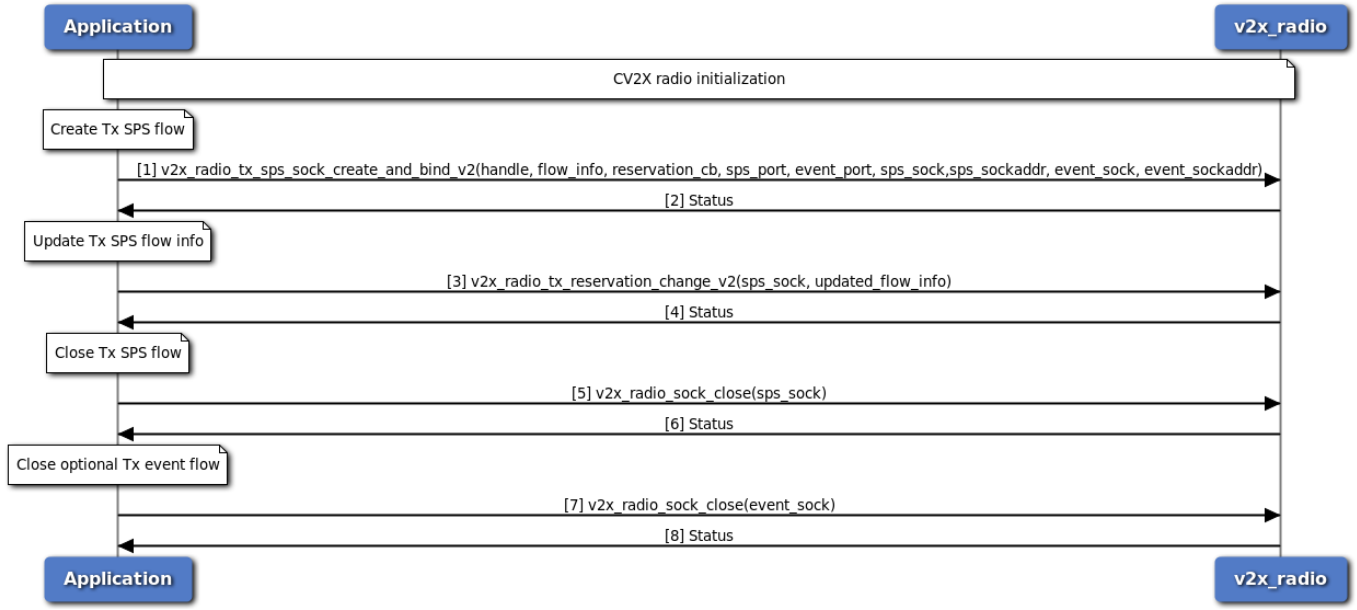

Figure 3-89 C-V2X Radio SPS Flow Call Flow - C Version

This call flow diagram describes the sequence of steps for registering or deregistering C-V2X Tx SPS flows using C version APIs. Application must perform C-V2X radio initialization before calling any methods of C-V2X radio. Only 2 Tx SPS flows are allowed in maximum in the system.

1. Application requests a new TX SPS flow from the C-V2X radio using v2x_radio_tx_sps_sock_create_and_bind_v2 method. The application can also specify an optional Tx event flow.
2. Application receives synchronous status (either 0 on SUCCESS or negative values if FAILED) indicating whether the sps flow was created successfully.
3. Application requests to change the SPS parameters using the v2x_radio_tx_reservation_change_v2 method.
4. Application received synchronous status (either 0 on SUCCESS or negative values if FAILED) indicating whether the sps flow was updated successfully.
5. Application requests to close the SPS flow.
6. Application receives synchronous status (either 0 on SUCCESS or negative values if FAILED) indicating whether the sps flow was closed successfully.
7. Application requests to close optional Tx event flow (if one was created).
8. Application receives synchronous status (either 0 on SUCCESS or negative values if FAILED) indicating whether the event flow was closed successfully.

3.6.10 C-V2X Throttle Manager Filer rate adjustment notification flow

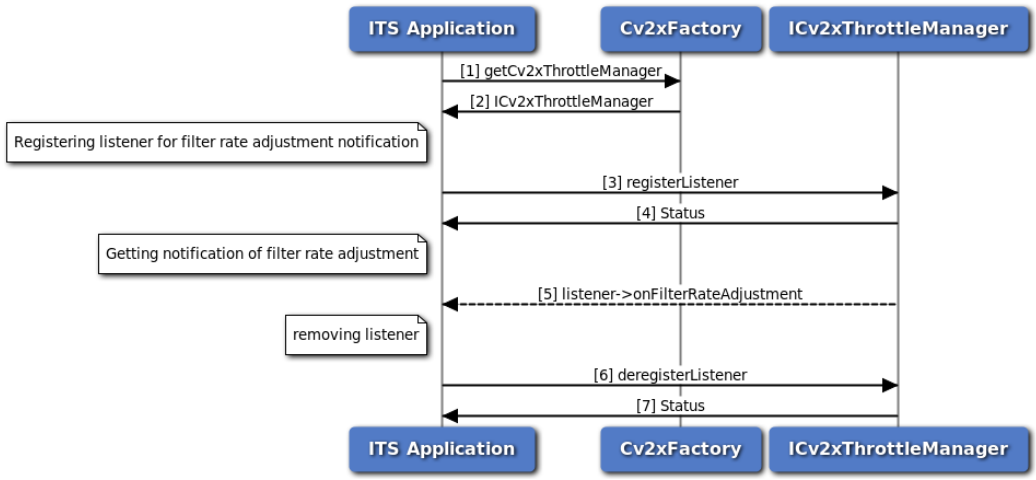

Figure 3-90 C-V2X Throttle Manager Filter Rate Adjustment Notification call flow

1. Application requests C-V2X factory for a C-V2X Throttle Manager.
2. C-V2X factory return ICv2xThrottleManager object to application.
3. Application register a listener for getting notifications of filter rate update.
4. Status of register listener i.e. either SUCCESS or FAILED will be returned to the application.
5. Application will get filter rate updates, positive value indicates to the application to filter more messages, and negative value indicates to the application to filter less messages
6. Application deregister the listener.
7. Status of deregistering listener, i.e. either SUCCESS or FAILED will be returned.

3.6.11 C-V2X Throttle Manager set verification load flow

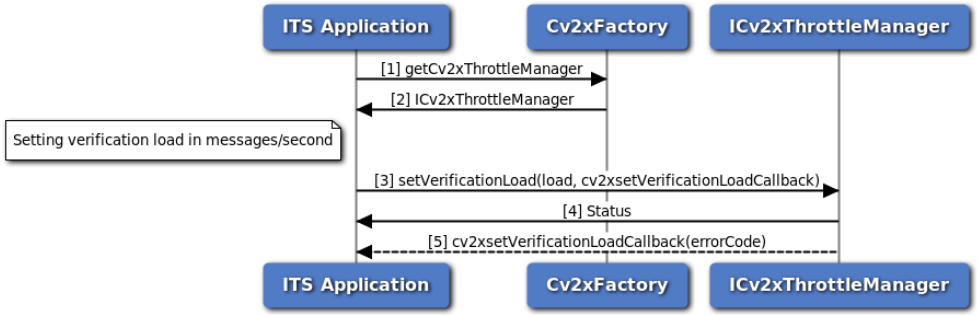

Figure 3-91 C-V2X Throttle Manager set verification call flow

1. Application requests C-V2X factory for a C-V2X Throttle Manager.
2. C-V2X factory return ICv2xThrottleManager object to application.
3. Application set verification load using setVerificationLoad method.

4. Application receives synchronous status which indicates if the request was sent successfully.
5. Application is notified of the status of the request (either SUCCESS or FAILED) via the application-supplied callback.

3.6.12 C-V2X TX Status Report

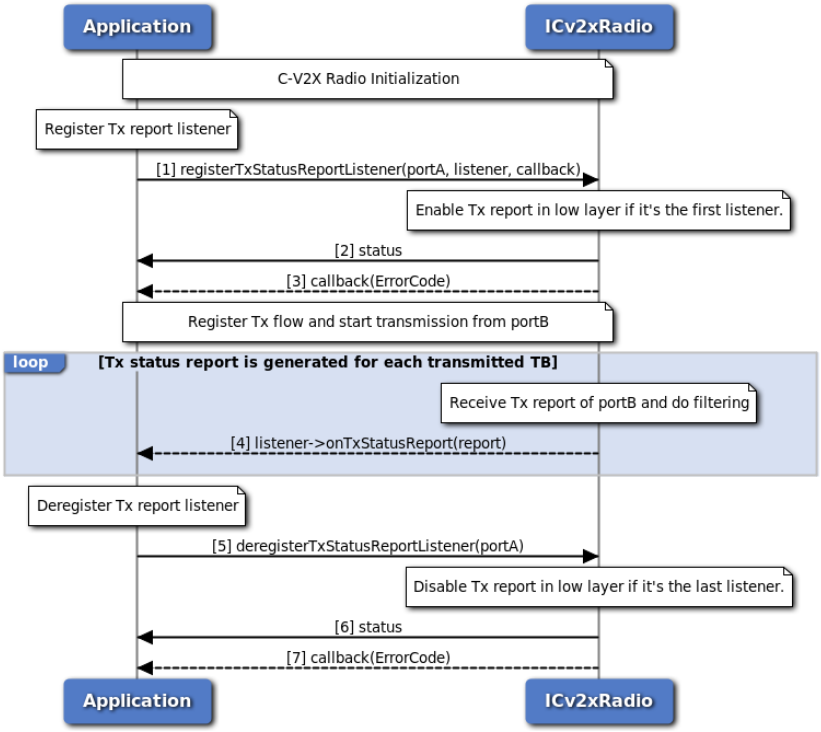

Figure 3-92 C-V2X TX Status Report Call Flow - C++ Version

This call flow diagram describes the sequence of steps for getting C-V2X Tx Status report per transport block using C++ version APIs. Application must perform C-V2X radio initialization before calling any methods of ICv2xRadio.

1. Application requests to register a listener for Tx status report, providing the interested port number(portA) of Tx status reports, the listener for receiving the reports and the callback function that used for notification of the result. C-V2X radio enables Tx status report in low layer if its the first request for registering Tx status report.
2. Application receives synchronous status (either SUCCESS or FAILED) indicating whether the request was sent successfully.
3. C-V2X radio sends asynchronous notification via the callback function indication the status of the request.
4. Tx status report is generated for each transport block transmitted, the portB included in Tx status report indicates which source port the specific transport block is sent from, it can be used to associate the Tx status report with a Tx SPS/Event flow. C-V2X radio filters received Tx status reports based on the port of the listener (portA) and the port included in Tx status report (portB):
    - If portA is 0, no filtering to the Tx status reports, application gets Tx status reports for all transport blocks.

- If portA is not 0 and it equals to portB, application only gets Tx status reports for transport blocks being sent from the specified port.
- In other cases, Tx status reports are not sent to application.

5. Application deregister listener for Tx status report. If it is the last listener for Tx status report in the system, C-V2X radio disables Tx status report in low layer.
6. Application receives synchronous status (either SUCCESS or FAILED) indicating whether the request was sent successfully.
7. C-V2X radio sends asynchronous notification via the callback (if callback was specified) indicating the status of the request.

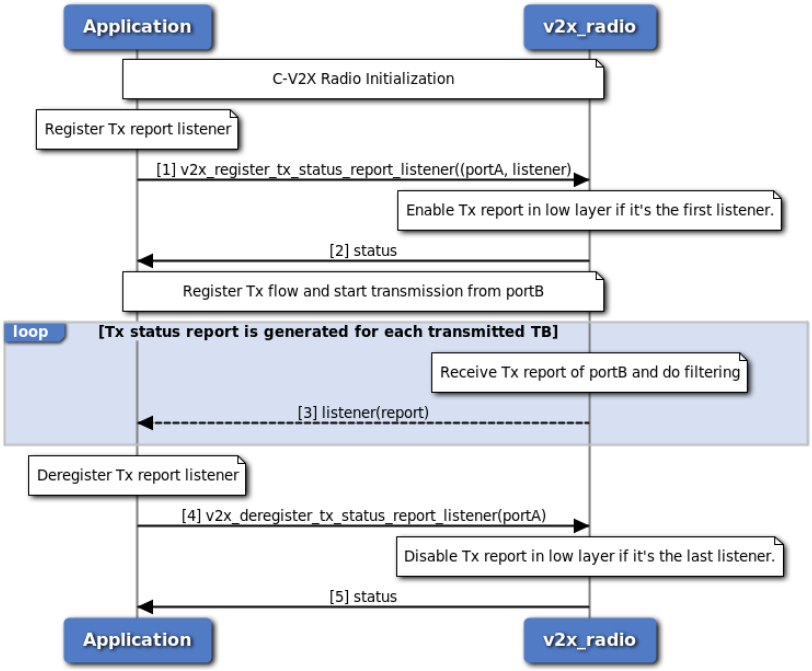

Figure 3-93 C-V2X TX Status Report Call Flow - C Version

This call flow diagram describes the sequence of steps for getting C-V2X Tx Status report per transport block using C version APIs. Application must perform C-V2X radio initialization before calling any methods of C-V2X radio.

1. Application requests to register a listener for Tx status report, providing the interested port number(portA) of Tx status reports and the listener for receiving the reports. C-V2X radio enables Tx status report in low layer if it is the first request for registering Tx status report.
2. Application receives synchronous status (either SUCCESS or FAILED) indicating whether the request was performed successfully.
3. Tx status report is generated for each transport block transmitted, the portB included in Tx status report indicates which source port the specific transport block is sent from, it can be used to associate the Tx status report with a Tx SPS/Event flow. C-V2X radio filters received Tx status reports based on the port of the listener (portA) and the port included in Tx status report (portB):
   - If portA is 0, no filtering to the Tx status reports, application gets Tx status reports for all transport blocks.

- If portA is not 0 and it equals to portB, application only gets Tx status reports for transport blocks being sent from the specified port.
- In other cases, Tx status reports are not sent to application.

4. Application deregister listener for Tx status report. If it is the last listener for Tx status report in the system, C-V2X radio disables Tx status report in low layer.
5. Application receives synchronous status (either SUCCESS or FAILED) indicating whether the request was performed successfully.

3.6.13 C-V2X RX Meta Data

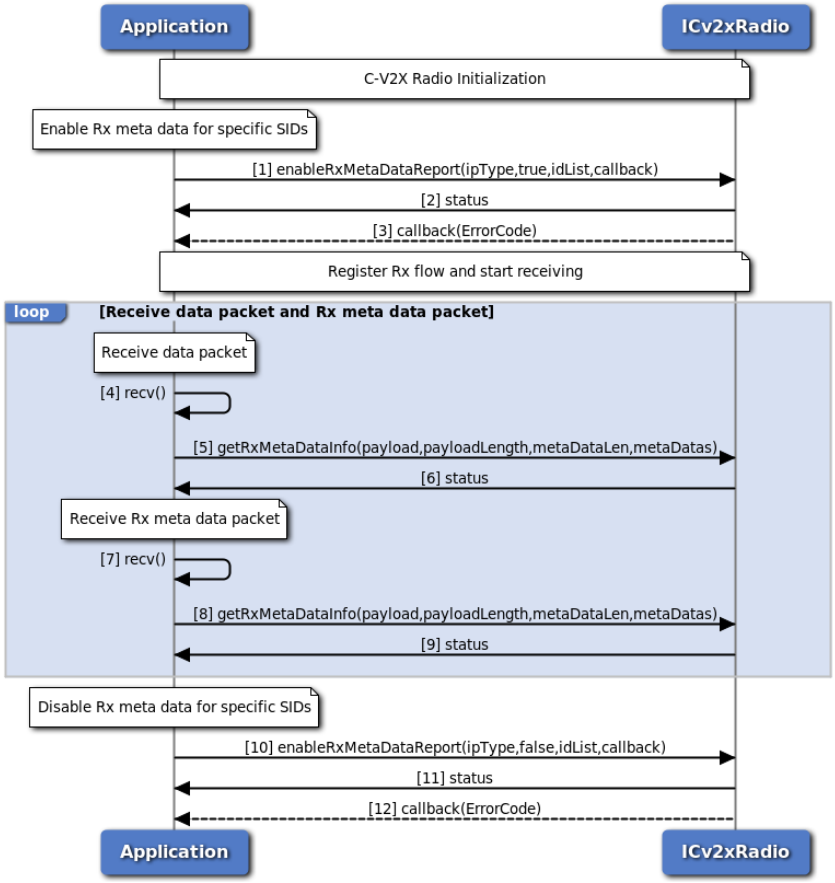

Figure 3-94 C-V2X RX Meta Data Call Flow - C++ Version

This call flow diagram describes the sequence of steps for enabling C-V2X Rx meta data per packet for non-IP traffic using C++ version APIs. Application must perform C-V2X radio initialization before calling any methods of ICv2xRadio.

1. Application requests to enable Rx meta data, providing the traffic type (only Non-IP is supported for Rx meta data), the interested service IDs and the callback function that used for notification of the result.
2. Application receives synchronous status (either SUCCESS or FAILED) indicating whether the request was sent successfully.
3. C-V2X radio sends asynchronous notification via the callback function indication the status of the

request.

4. After the enable of Rx meta data, application receives data packet which includes raw data along with SFN and subchannel index information via the Rx socket returned from Rx flow registration.
5. Application calls method getRxMetaDataInfo to get the raw data, SFN and subchannel index information from received data packet.
6. Application receives synchronous status (either SUCCESS or FAILED) indicating whether the data packet was parsed successfully.
7. Application receives Rx meta data packet via the Rx socket returned from Rx flow registration, one Rx meta data packet may include Rx meta data for multiple received data packets.
8. Application calls method getRxMetaDataInfo to get Rx meta data information from received Rx meta data packet. The information of SFN and subchannel index included in Rx meta data can be used to associate the Rx meta data with the received data packet.
9. Application receives synchronous status (either SUCCESS or FAILED) indicating whether the Rx meta data packet was parsed successfully.
10. Application requests to disable Rx meta data, providing the traffic type (only Non-IP is supported for Rx meta data), the registered service IDs and the callback function that used for notification of the result.
11. Application receives synchronous status (either SUCCESS or FAILED) indicating whether the request was sent successfully.
12. C-V2X radio sends asynchronous notification via the callback function indication the status of the request.

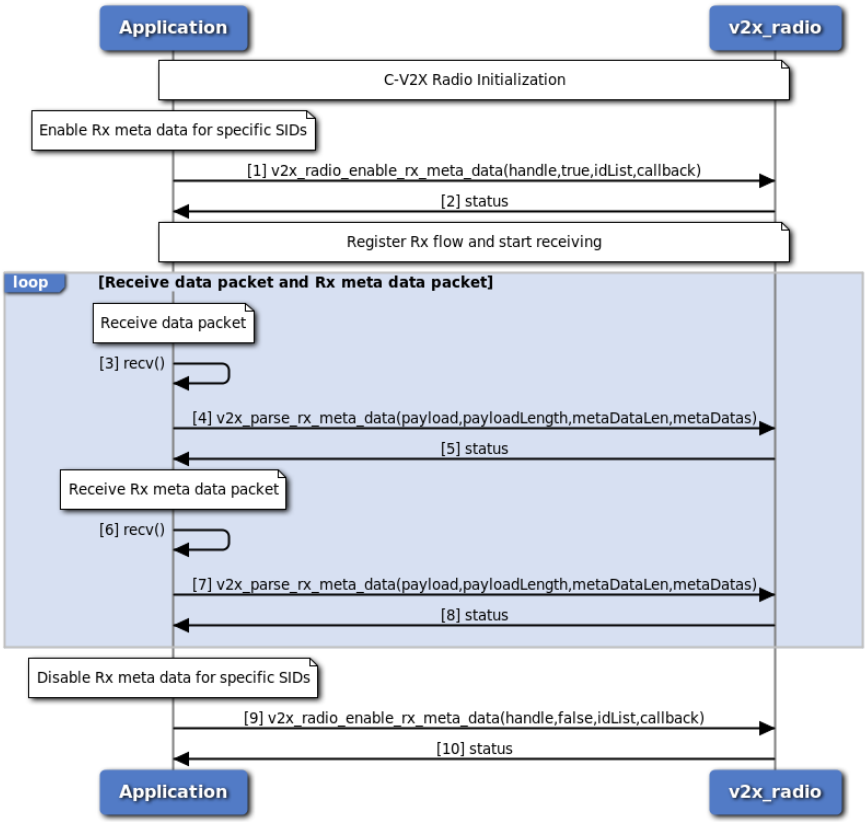

Figure 3-95 C-V2X RX Meta Data Call Flow - C Version

This call flow diagram describes the sequence of steps for enabling C-V2X Rx meta data per packet for non-IP traffic using C version APIs. Application must perform C-V2X radio initialization before calling any methods of C-V2X radio.

1. Application requests to enable Rx meta data, providing the traffic type (only Non-IP is supported for Rx meta data) and the interested service IDs.
2. Application receives synchronous status (either SUCCESS or FAILED) indicating whether the request was performed successfully.
3. After the enable of Rx meta data, application receives data packet which includes raw data along with SFN and subchannel index information via the Rx socket returned from Rx flow registration.
4. Application calls method v2x_parse_rx_meta_data to get the raw data, SFN and subchannel index information from received data packet.
5. Application receives synchronous status (either SUCCESS or FAILED) indicating whether the data packet was parsed successfully.
6. Application receives Rx meta data packet via the Rx socket returned from Rx flow registration, one Rx meta data packet may include Rx meta data for multiple received data packets.
7. Application calls method v2x_parse_rx_meta_data to get Rx meta data information from received Rx meta data packet. The information of SFN and subchannel index included in Rx meta data can be used to associate the Rx meta data with the received data packet.
8. Application receives synchronous status (either SUCCESS or FAILED) indicating whether the Rx meta data packet was parsed successfully.

9. Application requests to disable Rx meta data, providing the traffic type (only Non-IP is supported for Rx meta data), the registered service IDs and the callback function that used for notification of the result.
10. Application receives synchronous status (either SUCCESS or FAILED) indicating whether the request was performed successfully.

3.7 Audio

3.7.1 Audio Manager API call flow

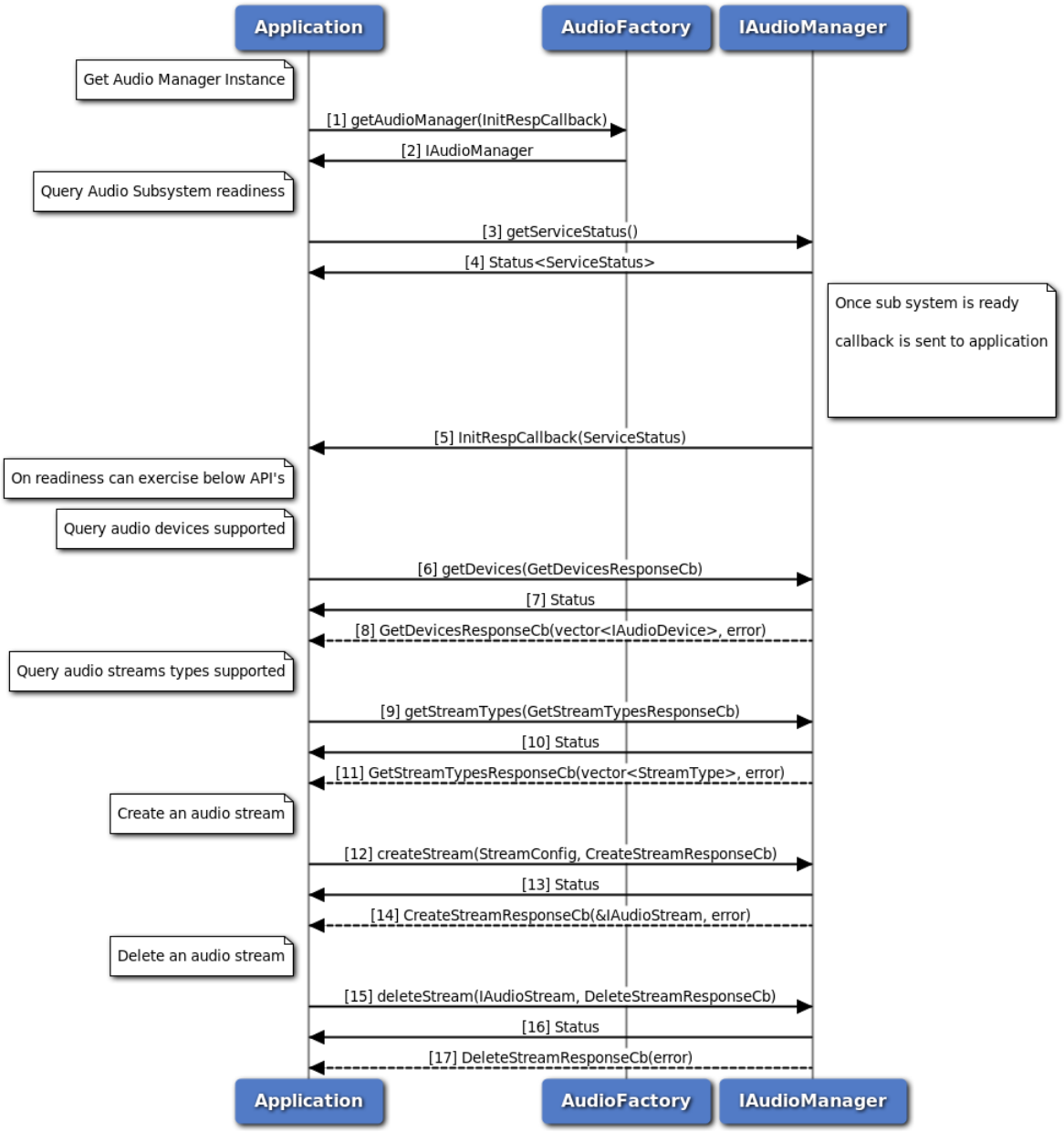

Figure 3-96 Audio Manager API call flow

1. Application requests Audio factory for an Audio Manager and passes callback pointer.
2. Audio factory return IAudioManager object to application.
3. Application can use IAudioManager::getServiceStatus to determine the state of sub system.
4. The application receives the ServiceStatus of sub system which indicates the state of service.
5. AudioManager notifies the application when the subsystem is ready through the callback mechanism.
6. On Readiness, Application requests supported device types using getDevices method.
7. Application receives synchronous Status which indicates if the getDevices request was sent successfully.
8. Application is notified of the Status of the getDevices request (either SUCCESS or FAILED) via the application-supplied callback, with array of supported device types.
9. Application requests supported stream types using getStreamTypes method.
10. Application receives synchronous Status which indicates if the getStreamTypes request was sent successfully.
11. Application is notified of the Status of the getStreamTypes request (either SUCCESS or FAILED) via the application-supplied callback, with array of supported stream types.
12. Application requests create audio stream using createStream method.
13. Application receives synchronous Status which indicates if the createStream request was sent successfully.
14. Application is notified of the Status of the createStream request (either SUCCESS or FAILED) via the application-supplied callback, with pointer to stream interface.
15. Application requests delete audio stream using deleteStream method.
16. Application receives synchronous Status which indicates if the deleteStream request was sent successfully.
17. Application is notified of the Status of the deleteStream request (either SUCCESS or FAILED) via the application-supplied callback.

3.7.2 Audio Voice Call Start/Stop call flow

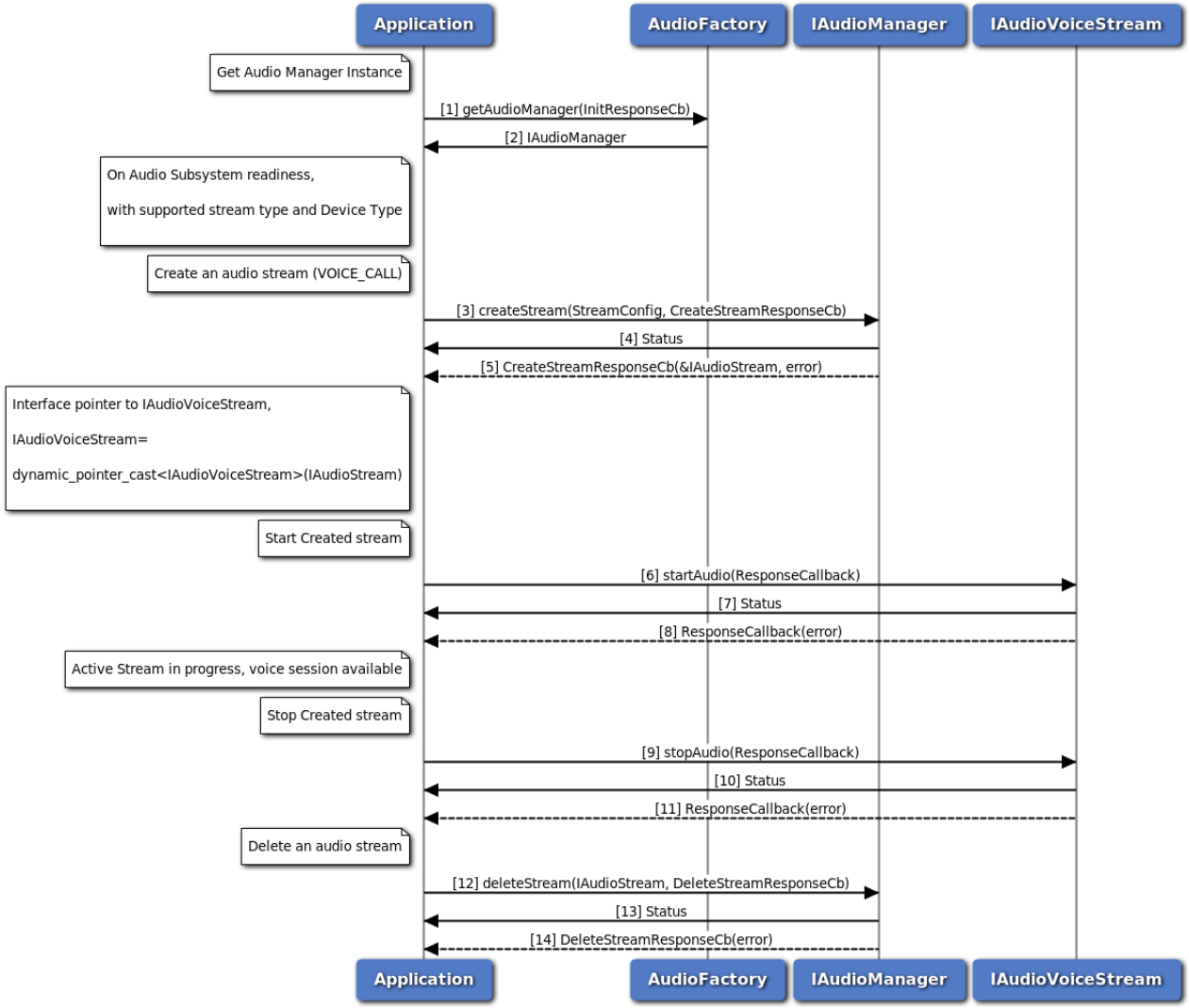

Figure 3-97 Audio Voice Call Start/Stop call flow

1. Application requests Audio factory for an Audio Manager.
2. Audio factory return IAudioManager object to application.
3. On Readiness, Application requests create audio voice stream using createStream method with streamType as VOICE_CALL.
4. Application receives synchronous Status which indicates if the createStream request was sent successfully.
5. Application is notified of the Status of the createStream request (either SUCCESS or FAILED) via the application-supplied callback, with pointer to stream interface refering to IAudioVoiceStream.
6. Application requests start audio stream using startAudio method on IAudioVoiceStream.
7. Application receives synchronous Status which indicates if the startAudio request was sent successfully.

8. Application is notified of the Status of the startAudio request (either SUCCESS or FAILED) via the application-supplied callback.
9. Application requests stop audio stream using stopAudio method on IAudioVoiceStream.
10. Application receives synchronous Status which indicates if the stopAudio request was sent successfully.
11. Application is notified of the Status of the stopAudio request (either SUCCESS or FAILED) via the application-supplied callback.
12. Application requests delete audio stream using deleteStream method.
13. Application receives synchronous Status which indicates if the deleteStream request was sent successfully.
14. Application is notified of the Status of the deleteStream request (either SUCCESS or FAILED) via the application-supplied callback.

3.7.3 Audio Voice Call Device Switch call flow

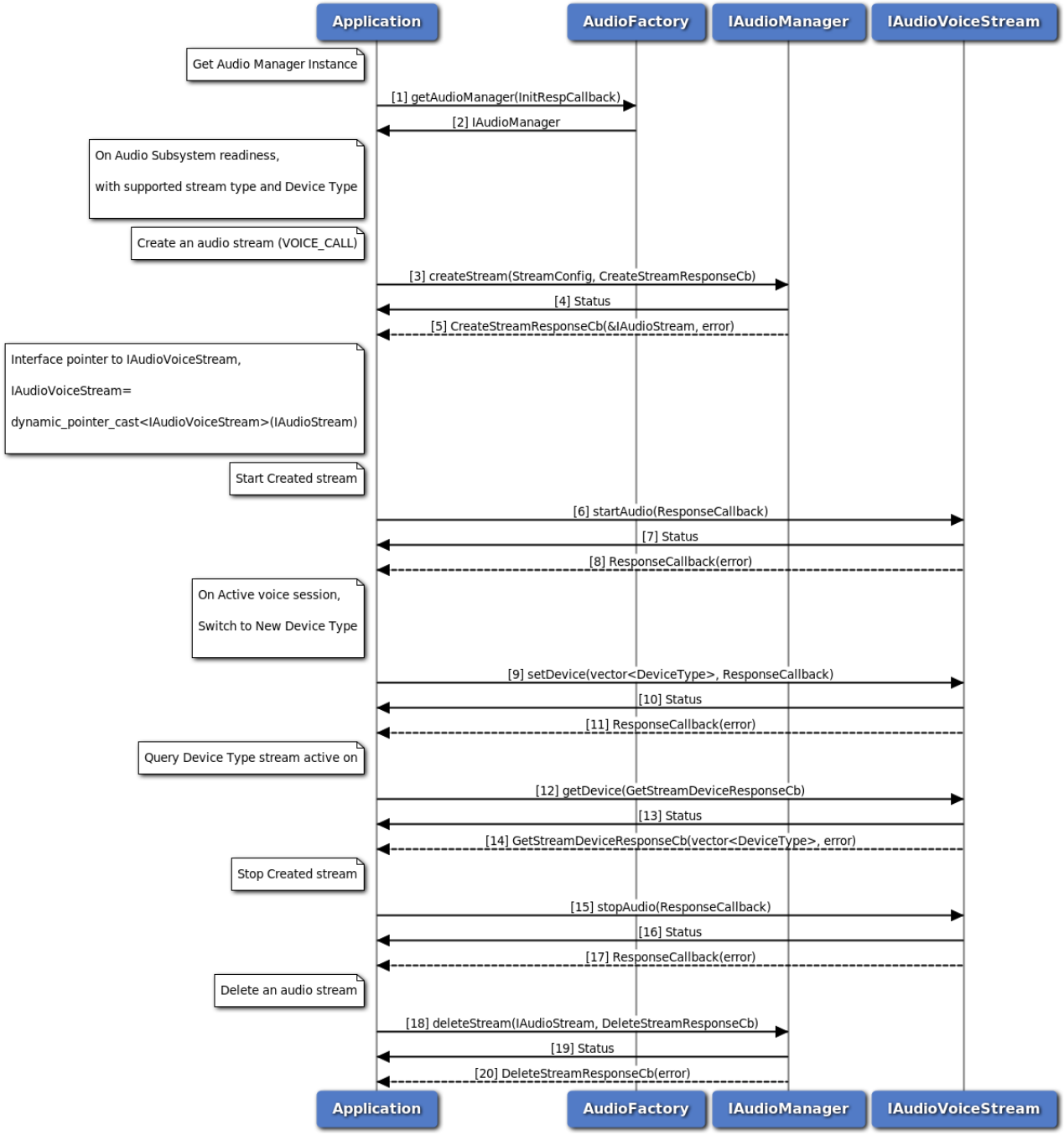

Figure 3-98 Audio Voice Call Device Switch call flow

1. Application requests Audio factory for an Audio Manager.
2. Audio factory return IAudioManager object to application.
3. On Readiness, Application requests create audio voice stream using createStream method with streamType as VOICE_CALL.

4. Application receives synchronous Status which indicates if the createStream request was sent successfully.
5. Application is notified of the Status of the createStream request (either SUCCESS or FAILED) via the application-supplied callback, with pointer to stream interface refering to IAudioVoiceStream.
6. Application requests start audio stream using startAudio method on IAudioVoiceStream.
7. Application receives synchronous Status which indicates if the startAudio request was sent successfully.
8. Application is notified of the Status of the startAudio request (either SUCCESS or FAILED) via the application-supplied callback.
9. Application requests new device routing of stream using setDevice method on IAudioVoiceStream.
10. Application receives synchronous Status which indicates if the setDevice request was sent successfully.
11. Application is notified of the Status of the setDevice request (either SUCCESS or FAILED) via the application-supplied callback.
12. Application query device stream routed to using getDevice method on IAudioVoiceStream.
13. Application receives synchronous Status which indicates if the getDevice request was sent successfully.
14. Application is notified of the Status of the getDevice request (either SUCCESS or FAILED) via the application-supplied callback, along with device types.
15. Application requests stop audio stream using stopAudio method on IAudioVoiceStream.
16. Application receives synchronous Status which indicates if the stopAudio request was sent successfully.
17. Application is notified of the Status of the stopAudio request (either SUCCESS or FAILED) via the application-supplied callback.
18. Application requests delete audio stream using deleteStream method.
19. Application receives synchronous Status which indicates if the deleteStream request was sent successfully.
20. Application is notified of the Status of the deleteStream request (either SUCCESS or FAILED) via the application-supplied callback.

3.7.4 Audio Voice Call Volume/Mute control call flow

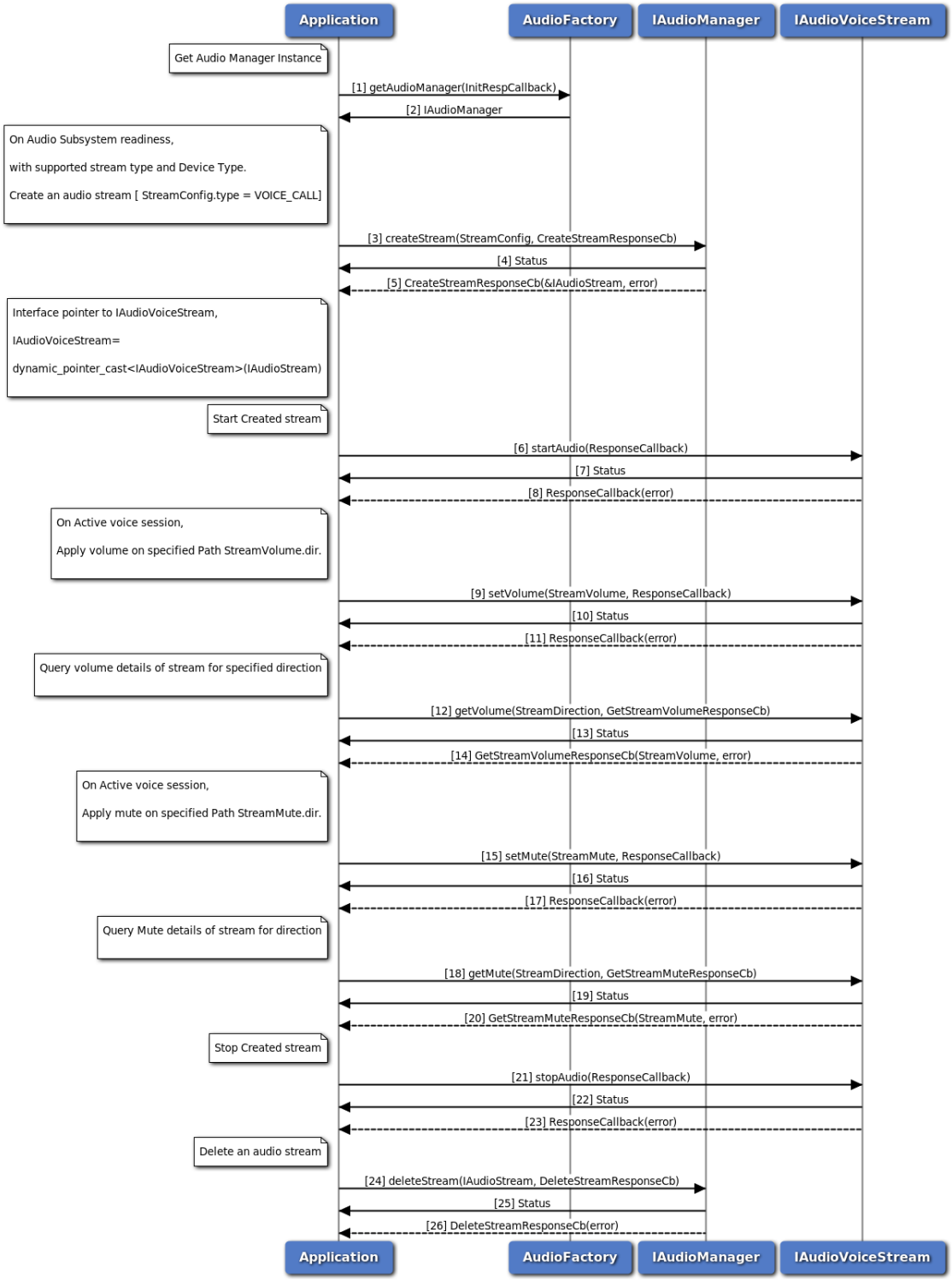

Figure 3-99 Audio Voice Call Volume/Mute control call flow

1. Application requests Audio factory for an Audio Manager.
2. Audio factory return IAudioManager object to application.

3. On Readiness, Application requests create audio voice stream using createStream method with streamType as VOICE_CALL.
4. Application receives synchronous Status which indicates if the createStream request was sent successfully.
5. Application is notified of the Status of the createStream request (either SUCCESS or FAILED) via the application-supplied callback, with pointer to stream interface refering to IAudioVoiceStream.
6. Application requests start audio stream using startAudio method on IAudioVoiceStream.
7. Application receives synchronous Status which indicates if the startAudio request was sent successfully.
8. Application is notified of the Status of the startAudio request (either SUCCESS or FAILED) via the application-supplied callback.
9. Application requests new volume on stream using setVolume method on IAudioVoiceStream for specified direction.
10. Application receives synchronous Status which indicates if the setVolume request was sent successfully.
11. Application is notified of the Status of the setVolume request (either SUCCESS or FAILED) via the application-supplied callback.
12. Application query volume on stream using getVolume method on IAudioVoiceStream for specified direction.
13. Application receives synchronous Status which indicates if the getVolume request was sent successfully.
14. Application is notified of the Status of the getVolume request (either SUCCESS or FAILED) via the application-supplied callback for specified direction with volume details.
15. Application requests new mute on stream using setMute method on IAudioVoiceStream for specified direction.
16. Application receives synchronous Status which indicates if the setMute request was sent successfully.
17. Application is notified of the Status of the setMute request (either SUCCESS or FAILED) via the application-supplied callback.
18. Application query mute details on stream using getMute method on IAudioVoiceStream for specified direction.
19. Application receives synchronous Status which indicates if the getMute request was sent successfully.
20. Application is notified of the Status of the getMute request (either SUCCESS or FAILED) via the application-supplied callback for specified direction with mute details.
21. Application requests stop audio stream using stopAudio method on IAudioVoiceStream.
22. Application receives synchronous Status which indicates if the stopAudio request was sent successfully.
23. Application is notified of the Status of the stopAudio request (either SUCCESS or FAILED) via the application-supplied callback.
24. Application requests delete audio stream using deleteStream method.
25. Application receives synchronous Status which indicates if the deleteStream request was sent

successfully.

26. Application is notified of the Status of the deleteStream request (either SUCCESS or FAILED) via the application-supplied callback.

3.7.5 Call flow to play DTMF tone

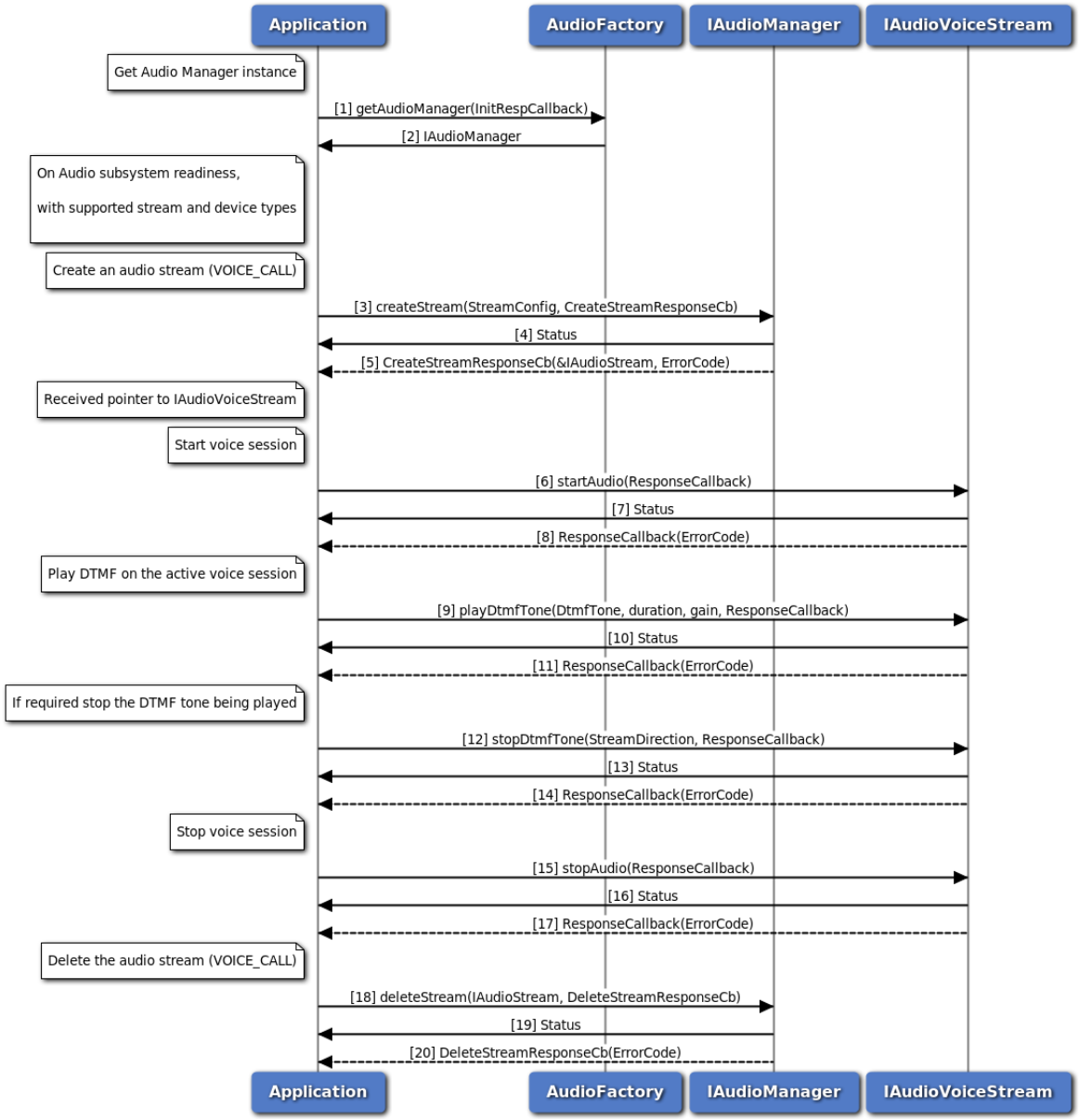

Figure 3-100 Call flow to play DTMF tone

1. Application requests Audio factory for an Audio Manager.
2. Audio factory return IAudioManager object to application.
3. On Readiness, Application requests to create a voice stream with streamType as VOICE_CALL.
4. Application receives synchronous status which indicates if the createStream request was sent

successfully.

5. Application is notified of the createStream request status (either SUCCESS or FAILED) via the application-supplied callback, with pointer to stream interface refering to IAudioVoiceStream.
6. Application requests to start voice session using startAudio method on IAudioVoiceStream.
7. Application receives synchronous status which indicates if the startAudio request was sent successfully.
8. Application is notified of the startAudio request status (either SUCCESS or FAILED) via the application-supplied callback.
9. Application requests to play a DTMF tone associated with the voice session
10. Application receives synchronous status which indicates if the playDtmfTone request was sent successfully.
11. Application is notified of the playDtmfTone request status (either SUCCESS or FAILED) via the application-supplied callback.
12. Application can optionally stop the DTMF tone being played, before its duration expires.
13. Application receives synchronous status which indicates if the stopDtmfTone request was sent successfully.
14. Application is notified of the stopDtmfTone request status (either SUCCESS or FAILED) via the application-supplied callback.
15. Application requests to stop the voice session using stopAudio method on IAudioVoiceStream.
16. Application receives synchronous Status which indicates if the stopAudio request was sent successfully.
17. Application is notified of the stopAudio request status(either SUCCESS or FAILED) via the application-supplied callback.
18. Application requests delete audio stream using deleteStream method.
19. Application receives synchronous Status which indicates if the deleteStream request was sent successfully.
20. Application is notified of the deleteStream request status(either SUCCESS or FAILED) via the application-supplied callback.

3.7.6 Call flow to detect DTMF tones

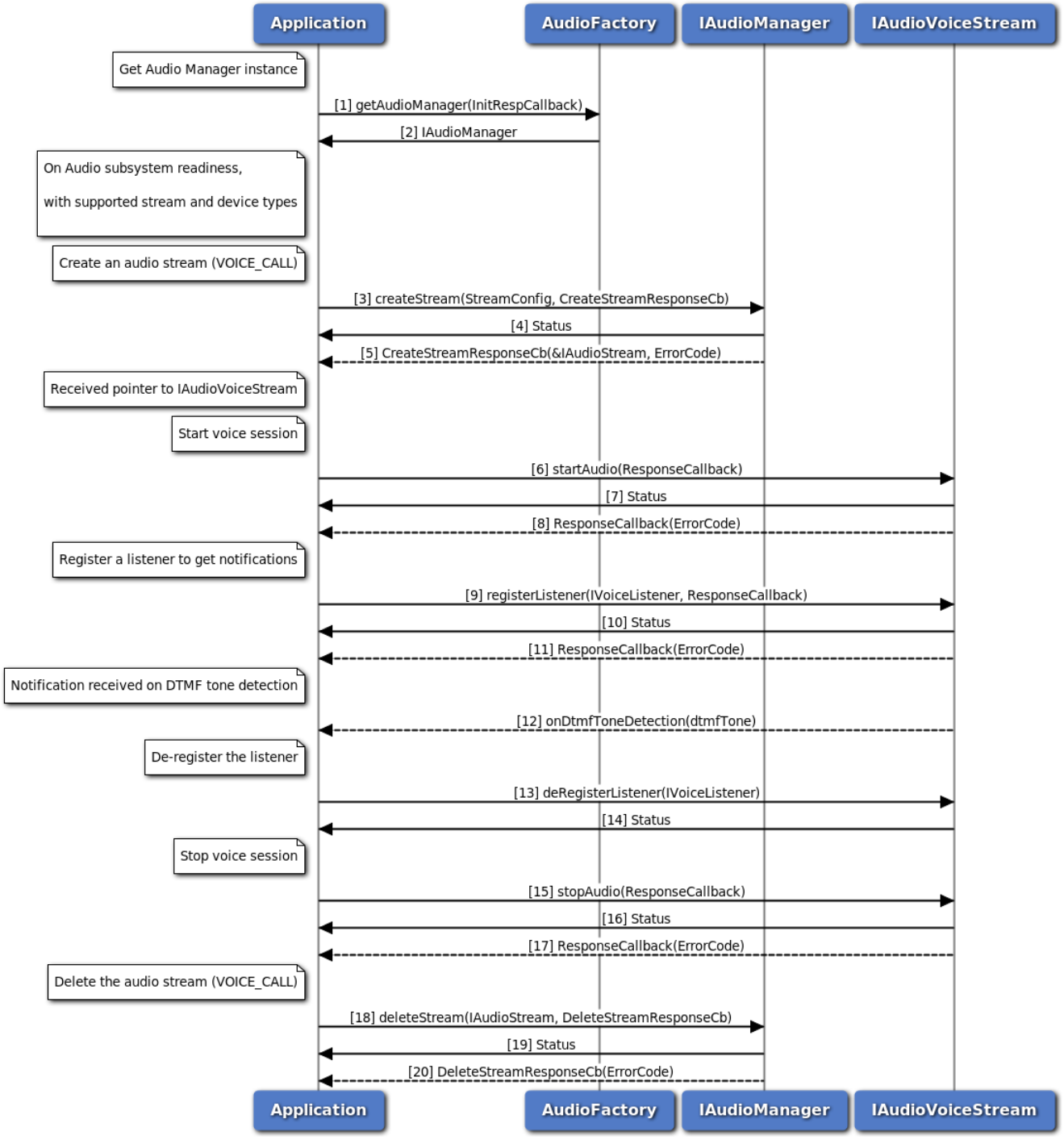

Figure 3-101 Call flow to detect DTMF tone

1. Application requests Audio factory for an Audio Manager.
2. Audio factory return IAudioManager object to application.

3. On Readiness, Application requests to create a voice stream with streamType as VOICE_CALL.
4. Application receives synchronous status which indicates if the createStream request was sent successfully.
5. Application is notified of the createStream request status (either SUCCESS or FAILED) via the application-supplied callback, with pointer to stream interface refering to IAudioVoiceStream.
6. Application requests to start voice session using startAudio method on IAudioVoiceStream.
7. Application receives synchronous status which indicates if the startAudio request was sent successfully.
8. Application is notified of the startAudio request status (either SUCCESS or FAILED) via the application-supplied callback.
9. Application registers a listener for getting notifications when DTMF tones are detected
10. Application receives synchronous status which indicates if the registerListener request was sent successfully.
11. Application is notified of the registerListener request status (either SUCCESS or FAILED) via the application-supplied callback.
12. Application receives onDtmfToneDetection notification when a DTMF tone is detected in the active voice call session
13. Application deregisters a listener to stop getting notifications
14. Application receives synchronous status which indicates if the deRegisterListener request was sent successfully.
15. Application requests to stop the voice session using stopAudio method on IAudioVoiceStream.
16. Application receives synchronous Status which indicates if the stopAudio request was sent successfully.
17. Application is notified of the stopAudio request status(either SUCCESS or FAILED) via the application-supplied callback.
18. Application requests delete audio stream using deleteStream method.
19. Application receives synchronous Status which indicates if the deleteStream request was sent successfully.
20. Application is notified of the deleteStream request status(either SUCCESS or FAILED) via the application-supplied callback.

3.7.7 Audio Playback call flow

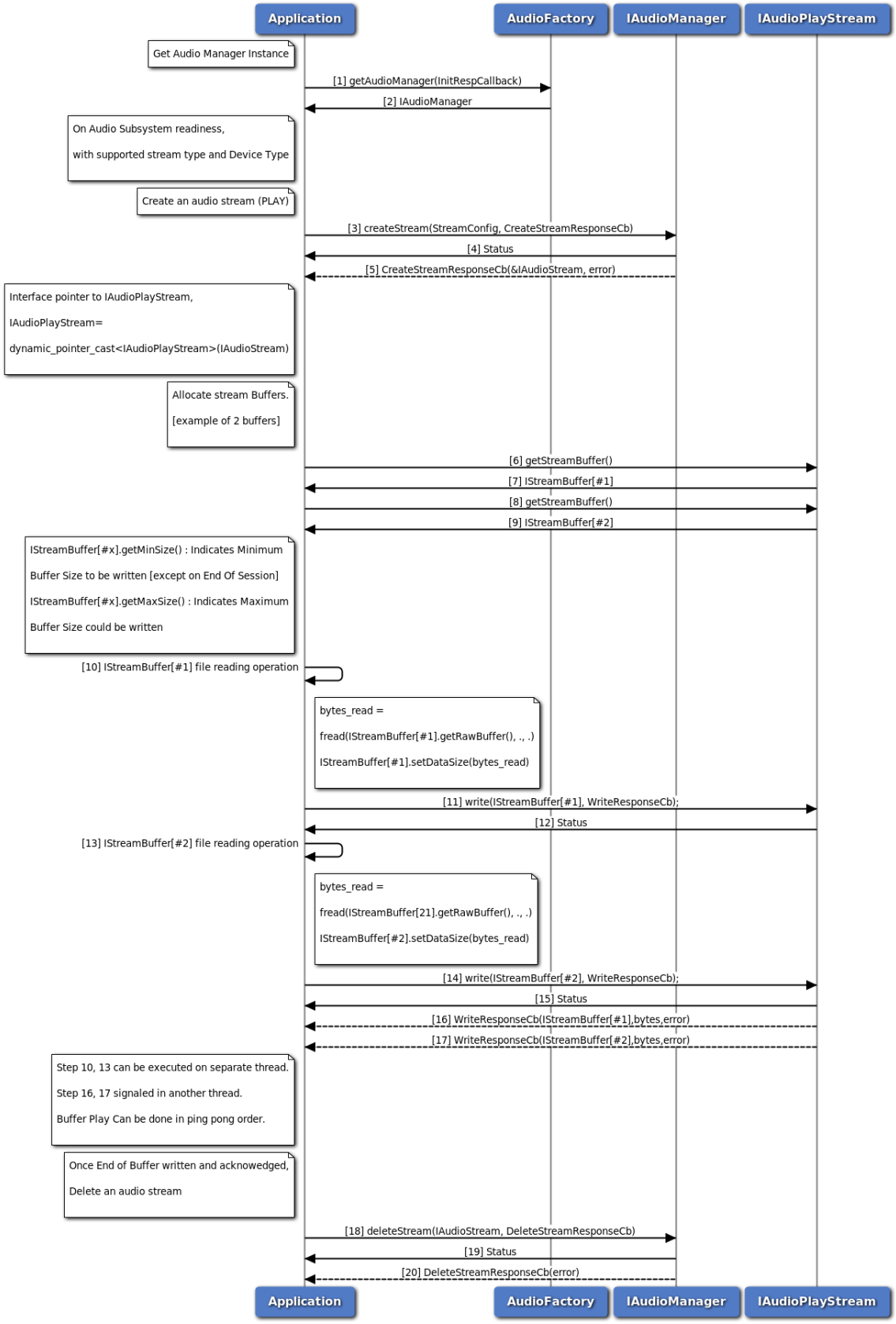

Figure 3-102 Audio Playback call flow

1. Application requests Audio factory for an Audio Manager.
2. Audio factory return IAudioManager object to application.
3. On Readiness, Application requests create audio playback stream using createStream method with streamType as PLAY.
4. Application receives synchronous Status which indicates if the createStream request was sent successfully.
5. Application is notified of the Status of the createStream request (either SUCCESS or FAILED) via the application-supplied callback, with pointer to stream interface refering to IAudioPlayStream.
6. Application requests stream buffer#1 using getStreamBuffer method on IAudioPlayStream.
7. Application receives IStreamBuffer if Success.
8. Application requests stream buffer#2 using getStreamBuffer method on IAudioPlayStream.
9. Application receives IStreamBuffer if Success.
10. Application writes audio samples on buffer#1 using getRawBuffer method on IStreamBuffer.
11. Application writes buffer#1 on Playback session using write method on IAudioPlayStream.
12. Application receives synchronous Status which indicates if the write request was sent successfully.
13. Application writes audio samples on buffer#2 using getRawBuffer method on IStreamBuffer.
14. Application writes buffer#2 on Playback session using write method on IAudioPlayStream.
15. Application receives synchronous Status which indicates if the write request was sent successfully.
16. Application is notified of the buffer#1 write Status (either SUCCESS or FAILED) via the application-supplied write callback with successful bytes written.
17. Application is notified of the buffer#2 write Status (either SUCCESS or FAILED) via the application-supplied write callback with successful bytes written.
18. Application requests delete audio stream using deleteStream method.
19. Application receives synchronous Status which indicates if the deleteStream request was sent successfully.
20. Application is notified of the Status of the deleteStream request (either SUCCESS or FAILED) via the application-supplied callback.

3.7.8 Audio Capture call flow

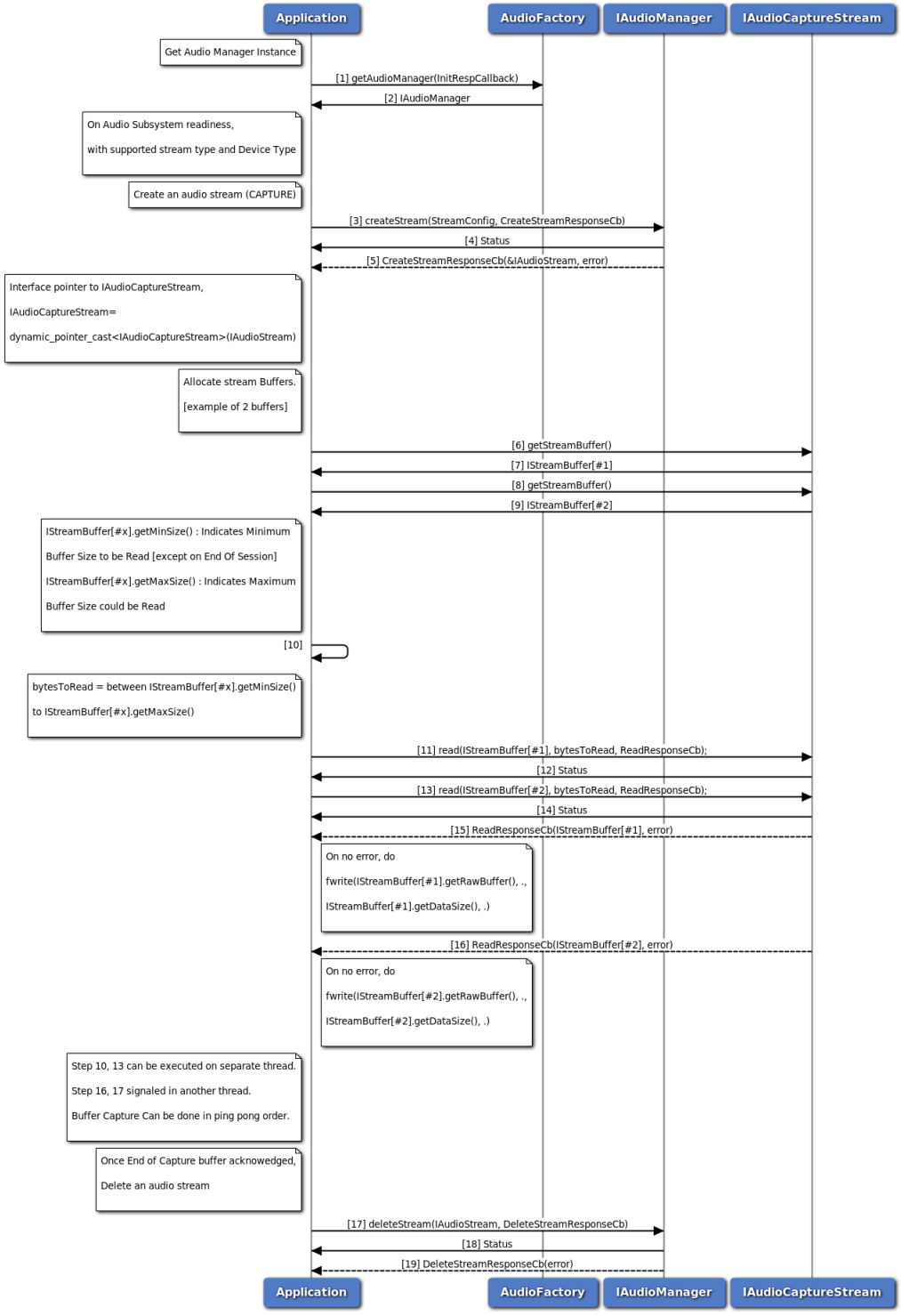

Figure 3-103 Audio Capture call flow

1. Application requests Audio factory for an Audio Manager.
2. Audio factory return IAudioManager object to application.

3. On Readiness, Application requests create audio capture stream using createStream method with streamType as CAPTURE.
4. Application receives synchronous Status which indicates if the createStream request was sent successfully.
5. Application is notified of the Status of the createStream request (either SUCCESS or FAILED) via the application-supplied callback, with pointer to stream interface refering to IAudioCaptureStream.
6. Application requests stream buffer#1 using getStreamBuffer method on IAudioCaptureStream.
7. Application receives IStreamBuffer if Success.
8. Application requests stream buffer#2 using getStreamBuffer method on IAudioCaptureStream.
9. Application receives IStreamBuffer if Success.
10. Application decides read sample size.
11. Application issue read audio samples on buffer#1 using read method on IAudioCaptureStream.
12. Application receives synchronous Status which indicates if the read request was sent successfully.
13. Application issue read audio samples on buffer#2 using read method on IAudioCaptureStream.
14. Application receives synchronous Status which indicates if the read request was sent successfully.
15. Application is notified of the buffer#1 write Status (either SUCCESS or FAILED) via the application-supplied read callback with successful bytes read.
16. Application is notified of the buffer#2 write Status (either SUCCESS or FAILED) via the application-supplied read callback with successful bytes read.
17. Application requests delete audio stream using deleteStream method.
18. Application receives synchronous Status which indicates if the deleteStream request was sent successfully.
19. Application is notified of the Status of the deleteStream request (either SUCCESS or FAILED) via the application-supplied callback.

3.7.9 Audio Tone Generator call flow

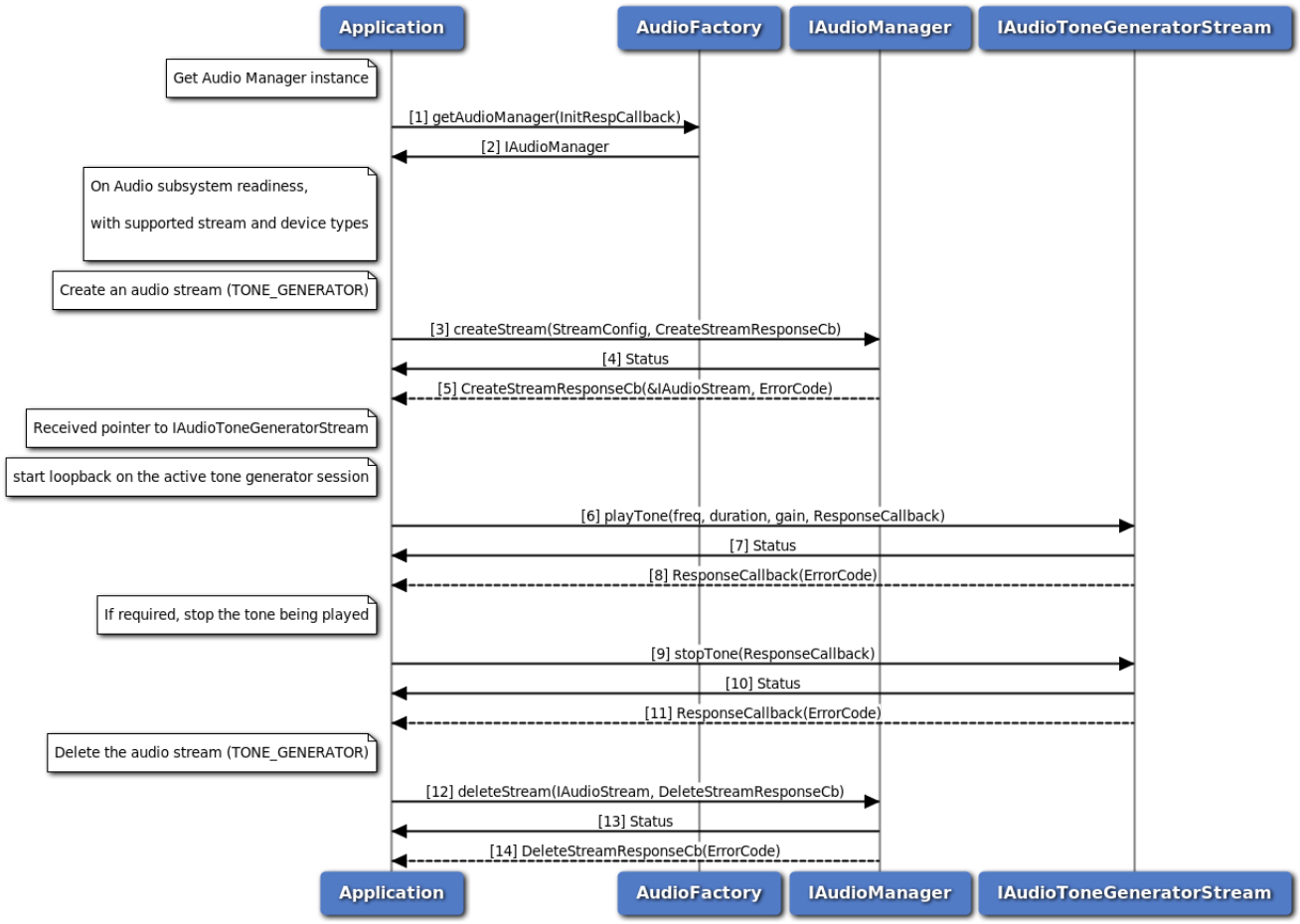

Figure 3-104 Call flow to play/stop tone on a sink device

1. Application requests Audio factory for an Audio Manager.
2. Audio factory return IAudioManager object to application.
3. On Readiness, Application requests to create a tone generator stream with streamType as TONE_GENERATOR.
4. Application receives synchronous status which indicates if the createStream request was sent successfully.
5. Application is notified of the createStream request status (either SUCCESS or FAILED) via the application-supplied callback, with pointer to stream interface refering to IAudioToneGeneratorStream.
6. Application requests to play tone using playTone method on IAudioToneGeneratorStream.
7. Application receives synchronous status which indicates if the playTone request was sent successfully.
8. Application is notified of the playTone request status (either SUCCESS or FAILED) via the application-supplied callback.

9. Application can optionally stop the tone being played, before its duration expires.
10. Application receives synchronous status which indicates if the stopTone request was sent successfully.
11. Application is notified of the stopTone request status (either SUCCESS or FAILED) via the application-supplied callback.
12. Application requests delete audio stream using deleteStream method.
13. Application receives synchronous Status which indicates if the deleteStream request was sent successfully.
14. Application is notified of the deleteStream request status(either SUCCESS or FAILED) via the application-supplied callback.

3.7.10 Audio Loopback call flow

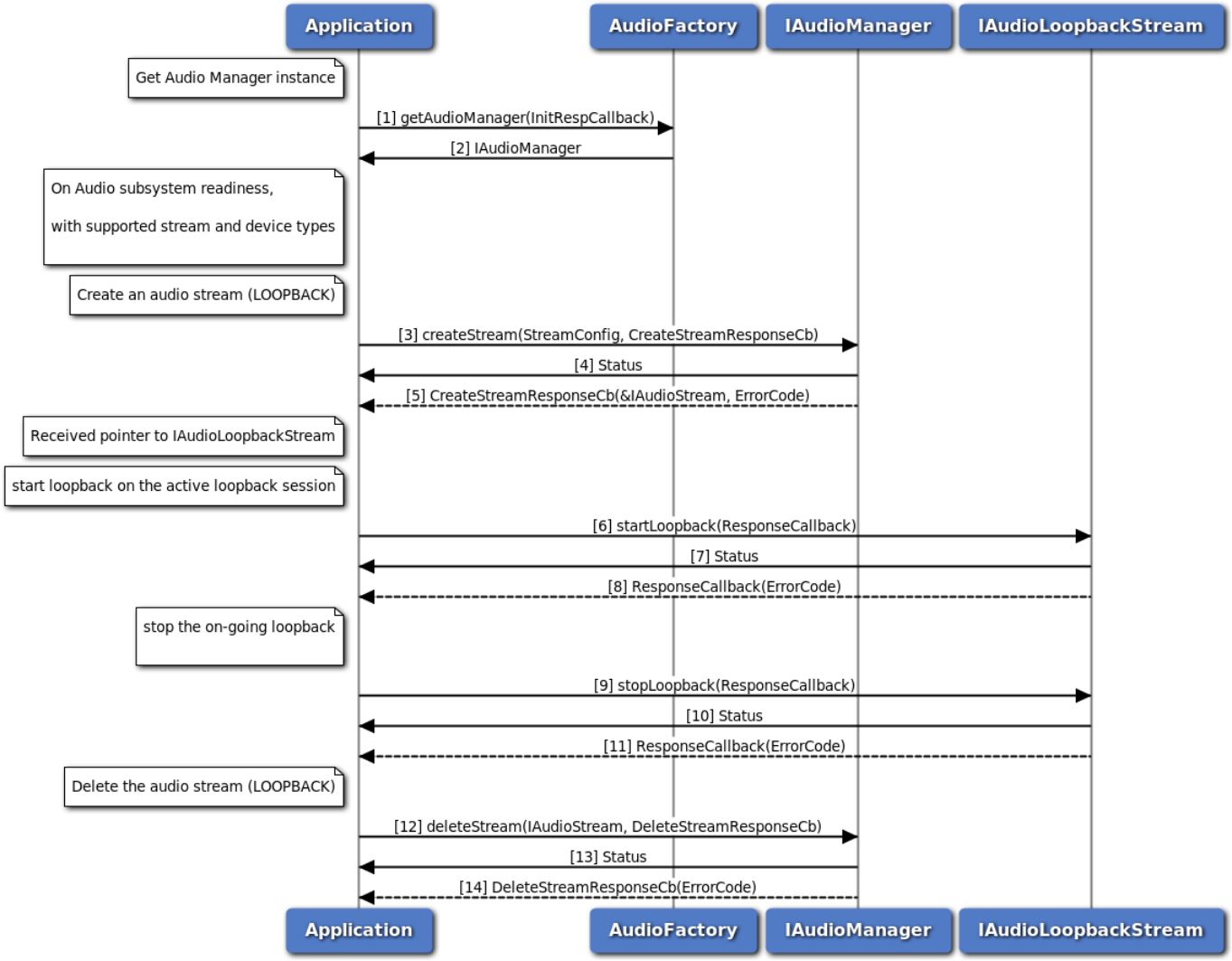

Figure 3-105 Call flow to start/stop loopback between source and sink devices

1. Application requests Audio factory for an Audio Manager.
2. Audio factory return IAudioManager object to application.
3. On Readiness, Application requests to create a loopback stream with streamType as LOOPBACK.
4. Application receives synchronous status which indicates if the createStream request was sent successfully.
5. Application is notified of the createStream request status (either SUCCESS or FAILED) via the application-supplied callback, with pointer to stream interface refering to IAudioLoopbackStream.
6. Application requests to start loopback using startLoopback method on IAudioLoopbackStream.
7. Application receives synchronous status which indicates if the startLoopback request was sent successfully.
8. Application is notified of the startLoopback request status (either SUCCESS or FAILED) via the application-supplied callback.
9. Application requests to stop loopback using stopLoopback method on IAudioLoopbackStream.
10. Application receives synchronous status which indicates if the stopLoopback request was sent successfully.
11. Application is notified of the stopLoopback request status (either SUCCESS or FAILED) via the application-supplied callback.
12. Application requests delete audio stream using deleteStream method.
13. Application receives synchronous Status which indicates if the deleteStream request was sent successfully.
14. Application is notified of the deleteStream request status(either SUCCESS or FAILED) via the application-supplied callback.

3.7.11 Compressed audio format playback call flow

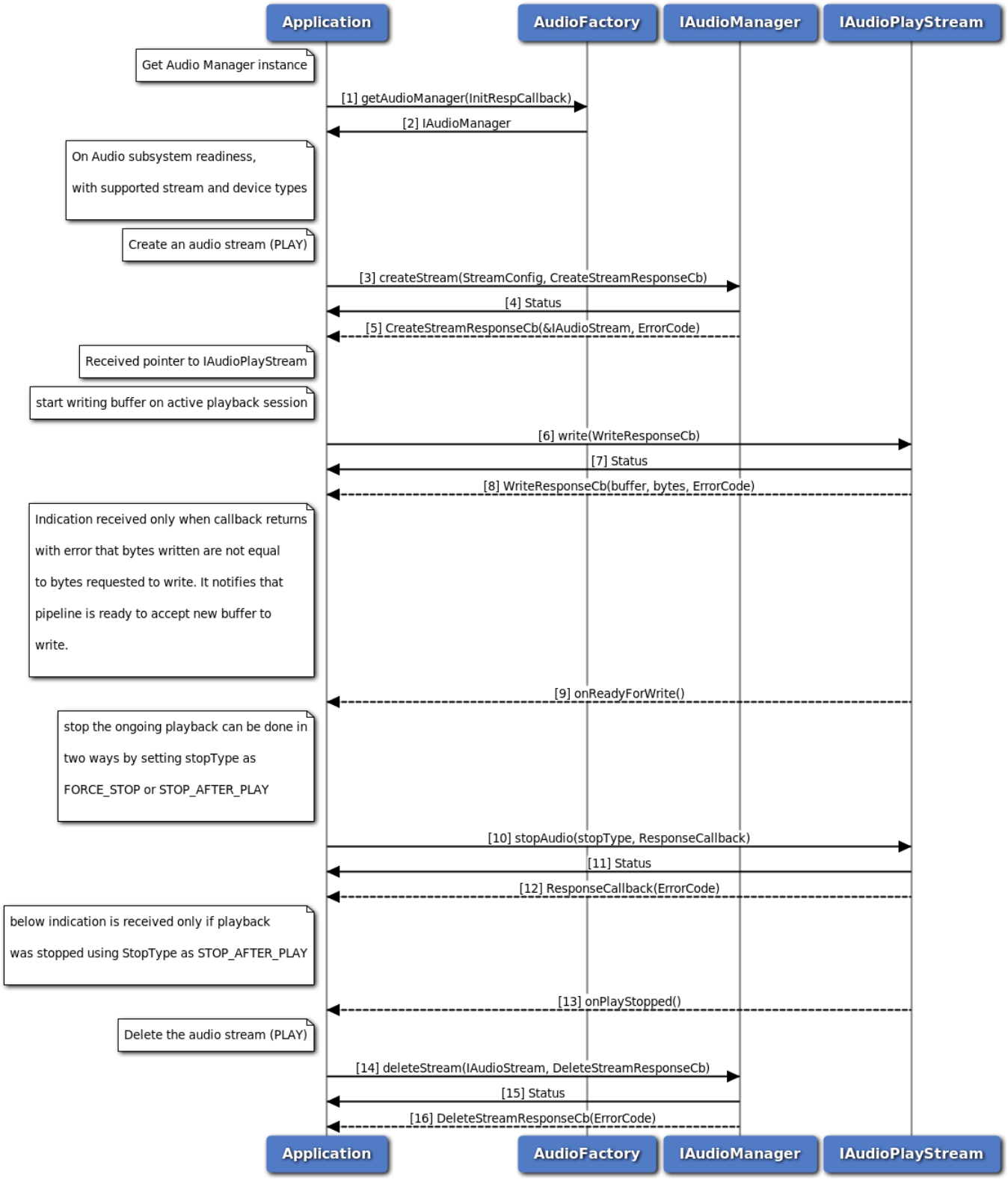

Figure 3-106 Call flow to play Compressed audio format

1. Application requests Audio factory for an Audio Manager.
2. Audio factory return IAudioManager object to application.
3. On Readiness, Application requests to create a play stream with streamType as PLAY.
4. Application receives synchronous status which indicates if the createStream request was sent successfully.
5. Application is notified of the createStream request status (either SUCCESS or FAILED) via the application-supplied callback, with pointer to stream interface refering to IAudioPlayStream.
6. Application requests to write buffer using write method on IAudioPlayStream.
7. Application receives synchronous status which indicates if the write request was sent successfully.
8. Application is notified of the write request status (either SUCCESS or FAILED) via the application-supplied callback along with number of bytes written.
9. Application is notified of when pipeline is ready to accept new buffer if callback returns with error that number of bytes written are not equal to bytes requested.
10. Application send request to stop playback using stopAudio method of IAudioPlayStream.
11. Application receives synchronous status which indicates if the stopAudio request was sent successfully.
12. Application is notified of the stopAudio request status (either SUCCESS or FAILED) via the application-supplied callback.
13. Appication is notified via indication that playback is stopped if StopType is STOP_AFTER_PLAY.
14. Application requests delete audio stream using deleteStream method.
15. Application receives synchronous Status which indicates if the deleteStream request was sent successfully.
16. Application is notified of the deleteStream request status(either SUCCESS or FAILED) via the application-supplied callback.

3.7.12 Audio Transcoding Operation Callflow

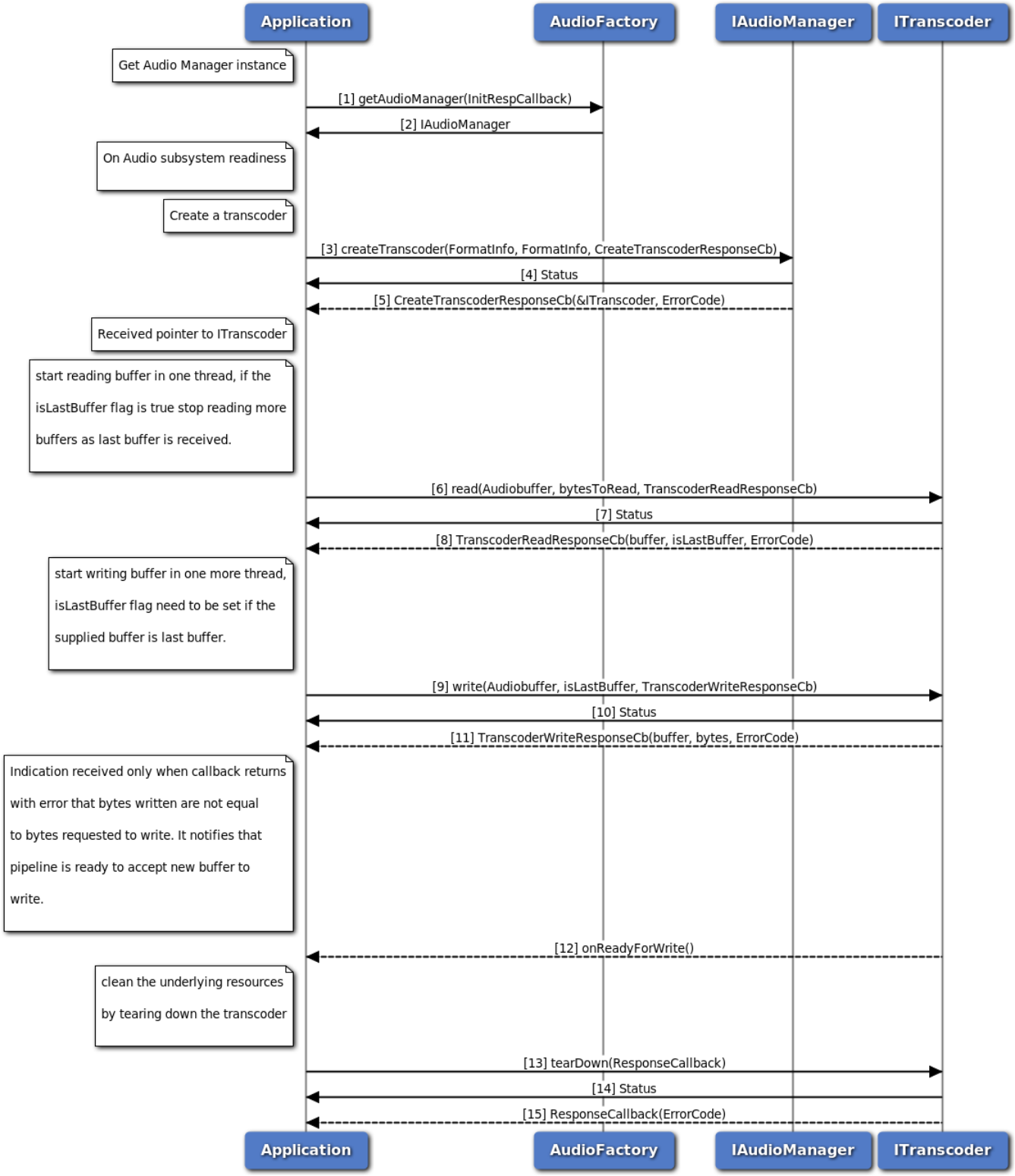

Figure 3-107 Audio Transcoding Operation Callflow

1. Application requests Audio factory for an Audio Manager.
2. Audio factory return IAudioManager object to application.
3. On Readiness, Application requests to create a transcoder.
4. Application receives synchronous status which indicates if the createTranscoder request was sent successfully.
5. Application is notified of the createTranscoder request status (either SUCCESS or FAILED) via the application-supplied callback, with pointer to transcoder interface refering to ITranscoder.
6. Application requests to read buffer using read method on ITranscoder.
7. Application receives synchronous status which indicates if the read request was sent successfully.
8. Application is notified of the read request status (either SUCCESS or FAILED) via the application-supplied callback along with isLastBuffer flag which indicates whether the buffer is last buffer to read or not.
9. Application requests to write buffer using write method on ITranscoder.
10. Application receives synchronous status which indicates if the write request was sent successfully.Application need to mark the isLastBuffer flag, whenever it is providing the last buffer to be write.
11. Application is notified of the write request status (either SUCCESS or FAILED) via the application-supplied callback along with number of bytes written.
12. Application is notified of when pipeline is ready to accept new buffer if callback returns with error that number of bytes written are not equal to bytes requested.
13. Once transcoding done, Application requests to tearDown transcoder as transcoder can not be used for multiple transcoding operations.
14. Application receives synchronous status which indicates if the tearDown request was sent successfully.
15. Application is notified of the tearDown request status (either SUCCESS or FAILED) via the application-supplied callback.

3.7.13 Compressed audio format playback on Voice Paths Callflow

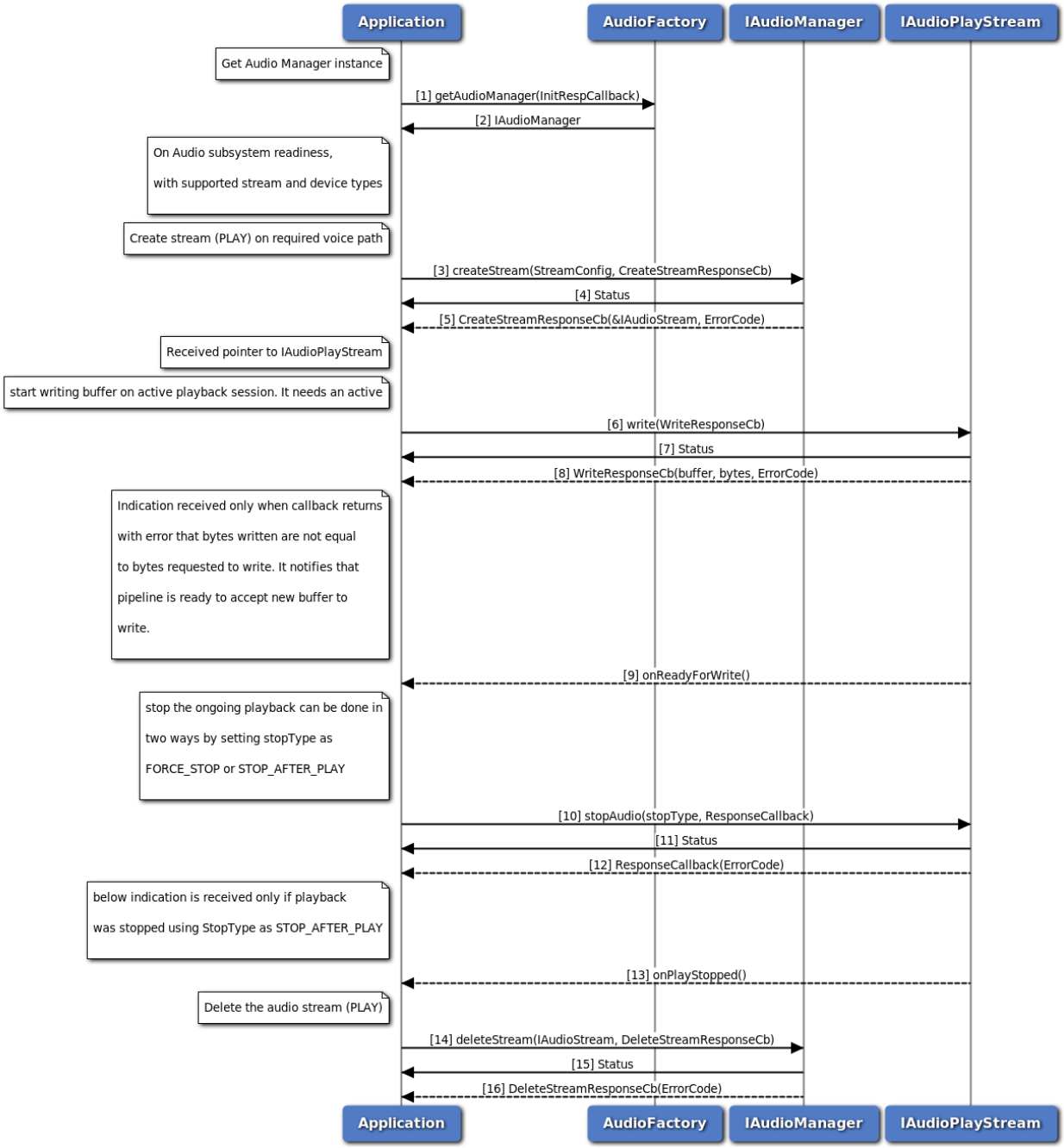

Figure 3-108 Compressed audio format playback on Voice Paths Callflow

1. Application requests Audio factory for an Audio Manager.
2. Audio factory return IAudioManager object to application.
3. On Readiness, Application requests to create a play stream with streamType as PLAY, voicePaths direction as TX or RX and no device is selected.

4. Application receives synchronous status which indicates if the createStream request was sent successfully.
5. Application is notified of the createStream request status (either SUCCESS or FAILED) via the application-supplied callback, with pointer to stream interface refering to IAudioPlayStream.
6. Application requests to write buffer using write method on IAudioPlayStream. It needs an active voice session to play over voice paths, refer IAudioVoiceStream for more details on how to create voice stream.
7. Application receives synchronous status which indicates if the write request was sent successfully.
8. Application is notified of the write request status (either SUCCESS or FAILED) via the application-supplied callback along with number of bytes written.
9. Application is notified of when pipeline is ready to accept new buffer if callback returns with error that number of bytes written are not equal to bytes requested.
10. Application send request to stop playback using stopAudio method of IAudioPlayStream.
11. Application receives synchronous status which indicates if the stopAudio request was sent successfully.
12. Application is notified of the stopAudio request status (either SUCCESS or FAILED) via the application-supplied callback.
13. Appication is notified via indication that playback is stopped if StopType is STOP_AFTER_PLAY.
14. Application requests delete audio play stream using deleteStream method.
15. Application receives synchronous Status which indicates if the deleteStream request was sent successfully.
16. Application is notified of the deleteStream request status(either SUCCESS or FAILED) via the application-supplied callback.

3.7.14 Audio Subsystem Restart Callflow

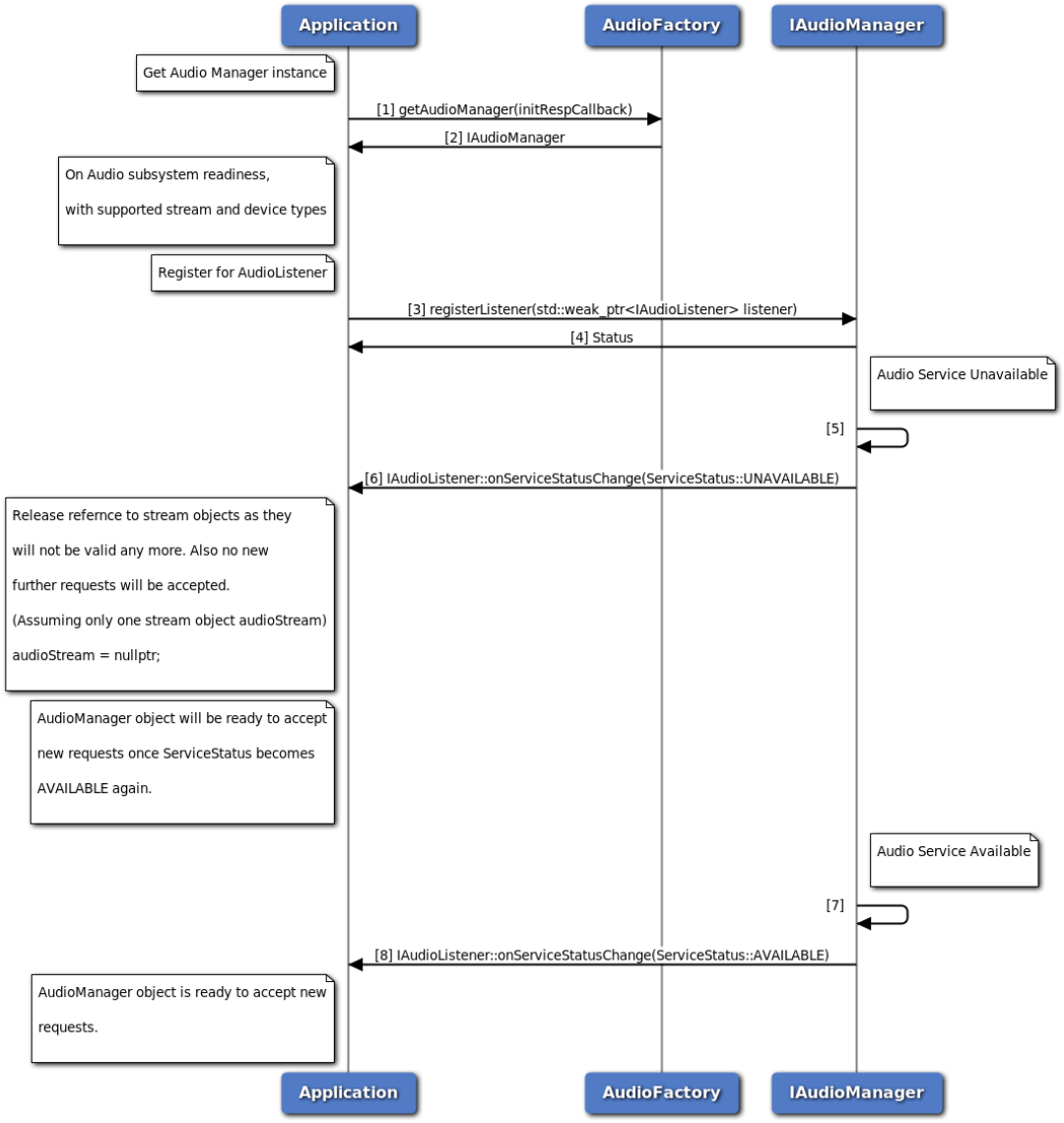

Figure 3-109 Audio Subsystem Restart Callflow

1. Application requests Audio factory for an Audio Manager.
2. Audio factory return IAudioManager object to application.
3. On Readiness, Application requests to register listener to IAudioManager object.
4. Application receives synchronous status which indicates registerListener request was sent successfully.
5. IAudioManager is notified that audio service is unavailable.
6. Application receives a notification from the IAudioManager regarding service status as unavailable. Application is supposed to relealse references to all the IAudioStream/ITranscoder objects and related resources. Application should not send any new request to IAudioManager or

IAudioStream/ITranscoder objects.

7. IAudioManager is notified that audio service is available.
8. Application receives a notification from the IAudioManager regarding service status as available. IAudioManager object is now ready to accept new requests from application.

3.7.15 Audio calibration configuration status

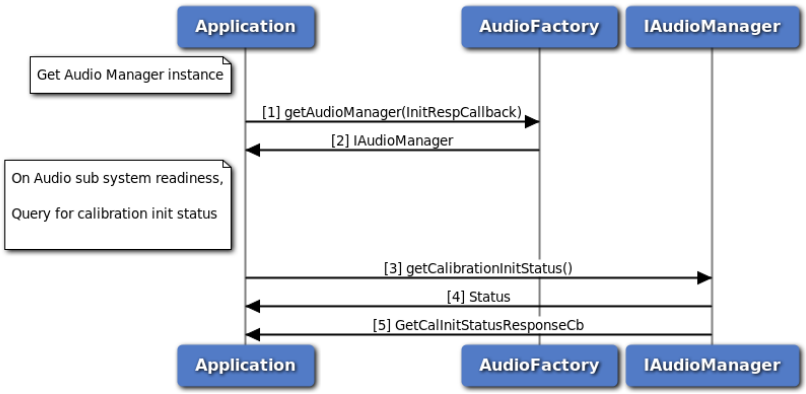

Figure 3-110 Audio calibration configuration status

1. Application requests Audio factory for an Audio Manager.
2. Audio factory return IAudioManager object to application.
3. On sub system Readiness, application requests for calibration init status using getCalibrationInitStatus API to IAudioManager.
4. Application receives synchronous status which indicates if the getCalibrationInitStatus request was sent successfully.
5. Application is notified with CalibrationInitStatus and suitable error code via the application supplied callback.

3.8 Thermal

3.8.1 Thermal Manager API call flow

Thermal manager provides APIs to get list of thermal zones and cooling devices. It also contains APIs to get a particular thermal zone and a particular cooling device details with the given Id.

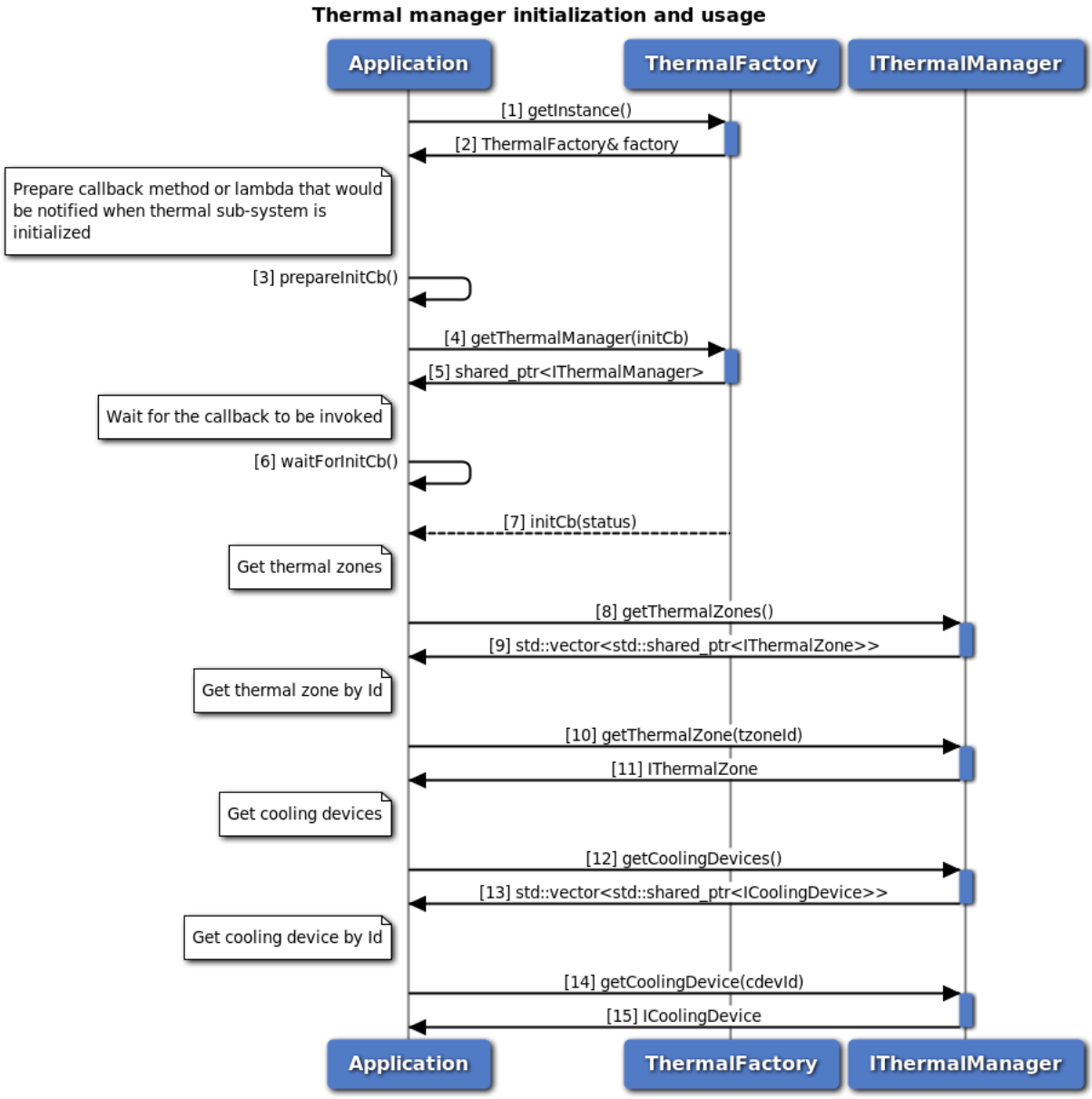

Figure 3-111 Thermal manager API call flow

1. Application requests for an instance of thermal factory.
2. Thermal factory instance is received by the application.
3. Application prepares the callback which would be called on initialization of thermal manager.
4. Application request for thermal manager from thermal factory.
5. Thermal factory returns an instance of the thermal manager object.
6. Application waits for initalization callback to be called.

7. Thermal factory invokes the callback once initialization completes.
8. Application sends request to get all thermal zones using IThermalManager object.
9. Thermal manager returns the list of thermal zones to the application.
10. Application requests for a particular thermal zone details by mentioning the thermal zone Id.
11. Application receives thermal zone details with the given Id from thermal manager.
12. Application sends request to get all cooling devices using IThermalManager object.
13. Thermal manager returns the list of cooling devices to the application.
14. Application requests for a particular cooling device details by passing the cooling device Id.
15. Thermal Manager sends cooling device details with the given Id to the application.

3.8.2 Call flow to register/remove listener for Thermal manager notifications.

Thermal listeners provide a set of listeners on which the app is notified when certain events occur. It also allows an application to register some or all types of notifications.

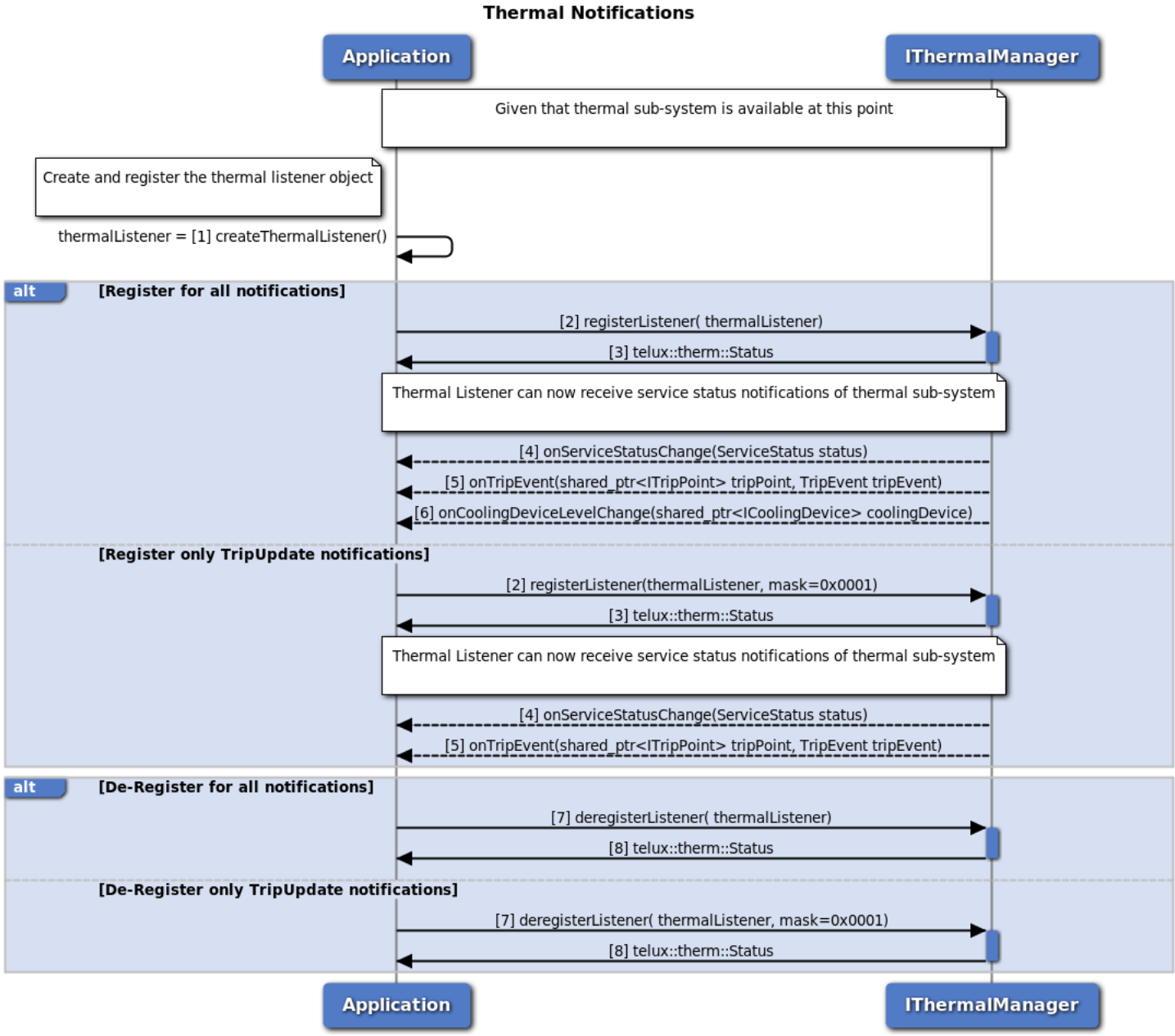

Figure 3-112 Call flow to register/remove listener for Thermal manager notifications

The thermal sub-system should have been initialized successfully with SERVICE_AVAILABLE as a pre-requisite for any thermal notifications and a valid ThermalManager object is available.

1. Application should create a thermal listening object.

2. Application can register a listener for specific or all type of notifications by providing a different mask value. The default value of mask is 0xFFFF which indicates for all type of notifications. The SSR notification is registered by default, whether it provides mask values or not.
3. Status of register listener i.e. either SUCCESS or FAILED will be returned to the application.
4. Application is notified when service status changes.
5. Application registered for this notification will be notified when a trip event occurs for any thermal zone.
6. Application registered for this notification will be notified when a cooling device changes its states.
7. Application can de-register a listener for specific or all type of notifications by providing a different mask value. The default value of mask is 0xFFFF which indicates for all type of notifications including SSR. The SSR notification will not be deregistered if the request provided a mask value.
8. Status of de-register listener i.e. either SUCCESS or FAILED will be returned to the application.

3.8.3 Thermal shutdown management

Thermal shutdown manager provides APIs to set/get auto thermal shutdown modes. It also has listener interface for notifications. Application will get the Thermal-shutdown manager object from thermal factory. The application can register a listener for updates in thermal auto shutdown modes and its management service status. Also there is provision to set the desired thermal auto-shutdown mode.

When application is notifed of service being unavailable, the thermal auto-shutdown mode updates are inactive. After service becomes available, the existing listener registrations will be maintained.

As a reference, auto-shutdown management in an eCall application is described in the below sections.

3.8.3.1 Call flow to register/remove listener for Thermal auto-shutdown mode updates.

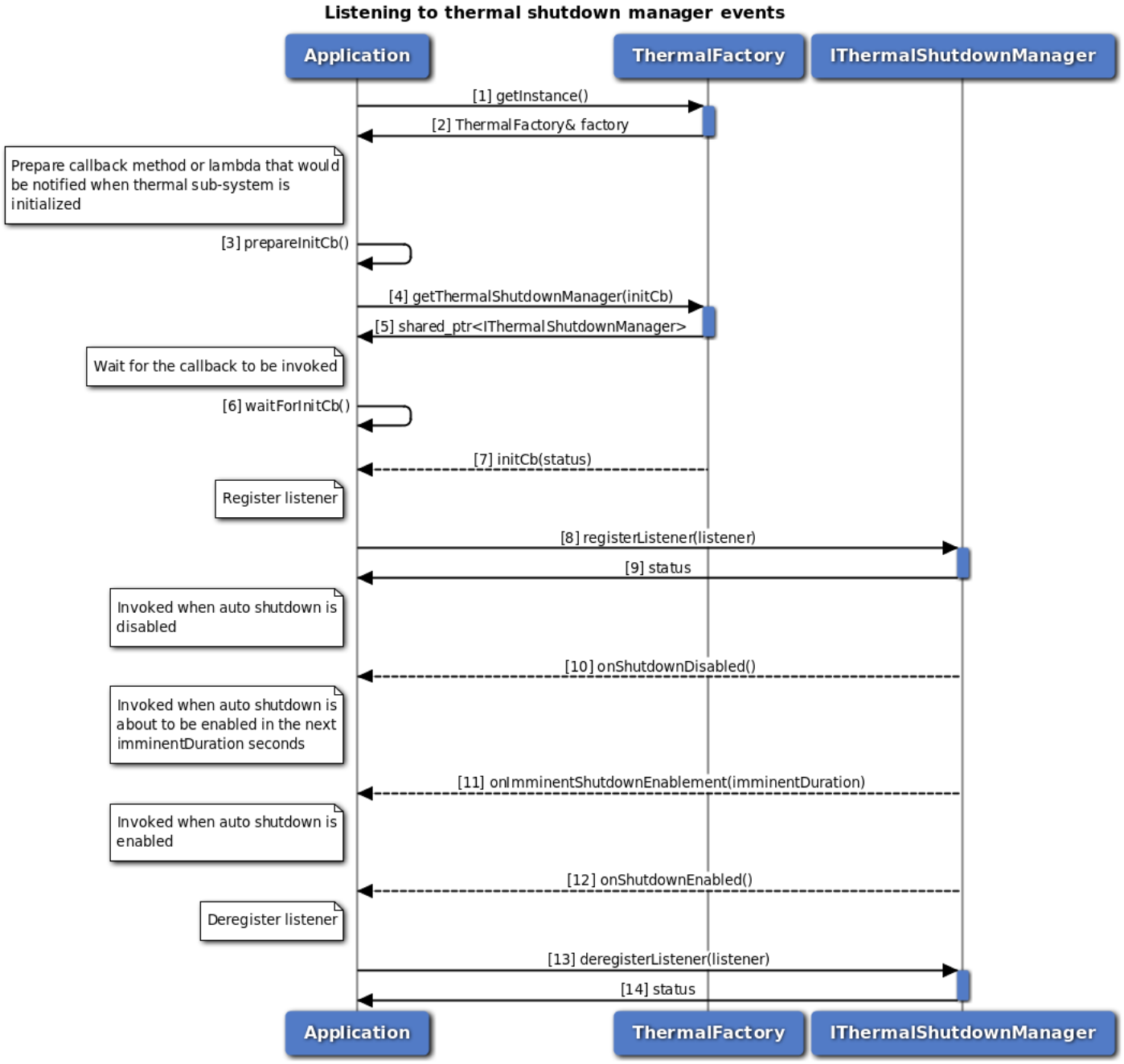

Figure 3-113 Call flow to register/remove listener for Thermal shutdown manager

1. Application requests for an instance of thermal factory.
2. Thermal factory instance is received by the application.
3. Application prepares the callback which would be called on initialization of thermal shutdown manager.
4. Application request for thermal shutdown manager from thermal factory.

5. Thermal factory returns an instance of the thermal shutdown manager object.
6. Application waits for initalization callback to be called.
7. Thermal factory invokes the callback once initialization completes.
8. Application can register a listener for getting notifications on Thermal auto-shutdown mode updates.
9. Status of register listener i.e. either SUCCESS or FAILED will be returned to the application.
10. Application receives a notification that thermal auto-shutdown mode is disabled.
11. Application receives a notification that thermal auto-shutdown mode is going to enabled soon. The exact duration is also recieved as part of notification.
12. Application receives a notification that thermal auto-shutdown mode is enabled.
13. Application can remove listener.
14. Status of remove listener i.e. either SUCCESS or FAILED will be returned to the application.

3.8.3.2 Call flow to set/get the Thermal auto-shutdown mode

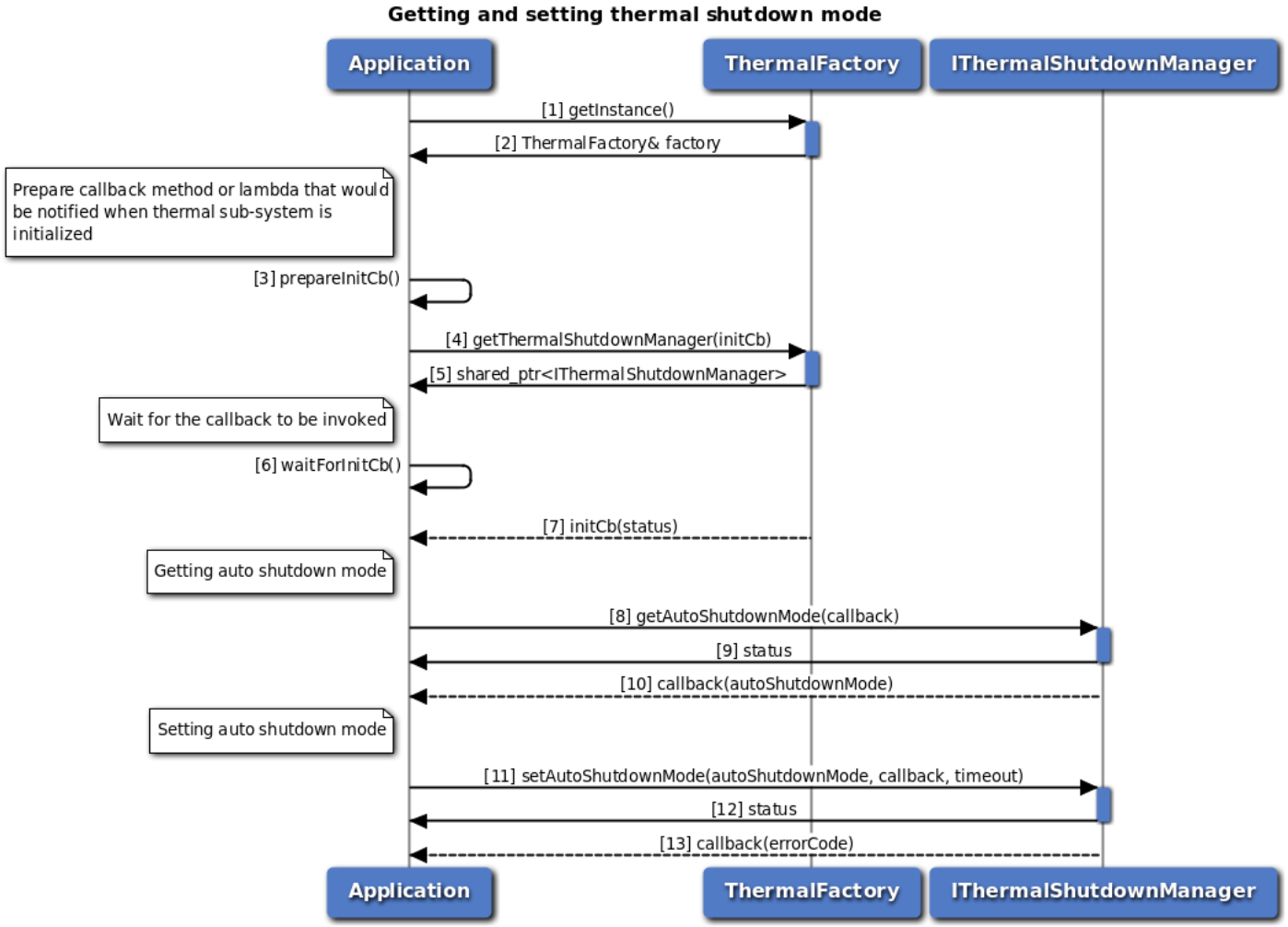

Figure 3-114 Call flow to set/get the Thermal auto-shutdown mode

1. Application requests for an instance of thermal factory.
2. Thermal factory instance is received by the application.
3. Application prepares the callback which would be called on initialization of thermal shutdown manager.
4. Application request for thermal shutdown manager from thermal factory.
5. Thermal factory returns an instance of the thermal shutdown manager object.
6. Application waits for initalization callback to be called.
7. Thermal factory invokes the callback once initialization completes.
8. Application can query the thermal auto-shutdown mode.
9. Application receives synchronous status which indicates if the request was sent successfully.
10. Application receives the auto-shutdown mode asynchronously.
11. Application can set the thermal auto-shutdown mode to ENABLE or DISABLE.
12. Application receives synchronous status which indicates if the request was sent successfully.
13. Optionally, the response to setAutoShutdownMode request can be received by the application.

3.8.3.3 Call flow to manage thermal auto-shutdown from an eCall application.

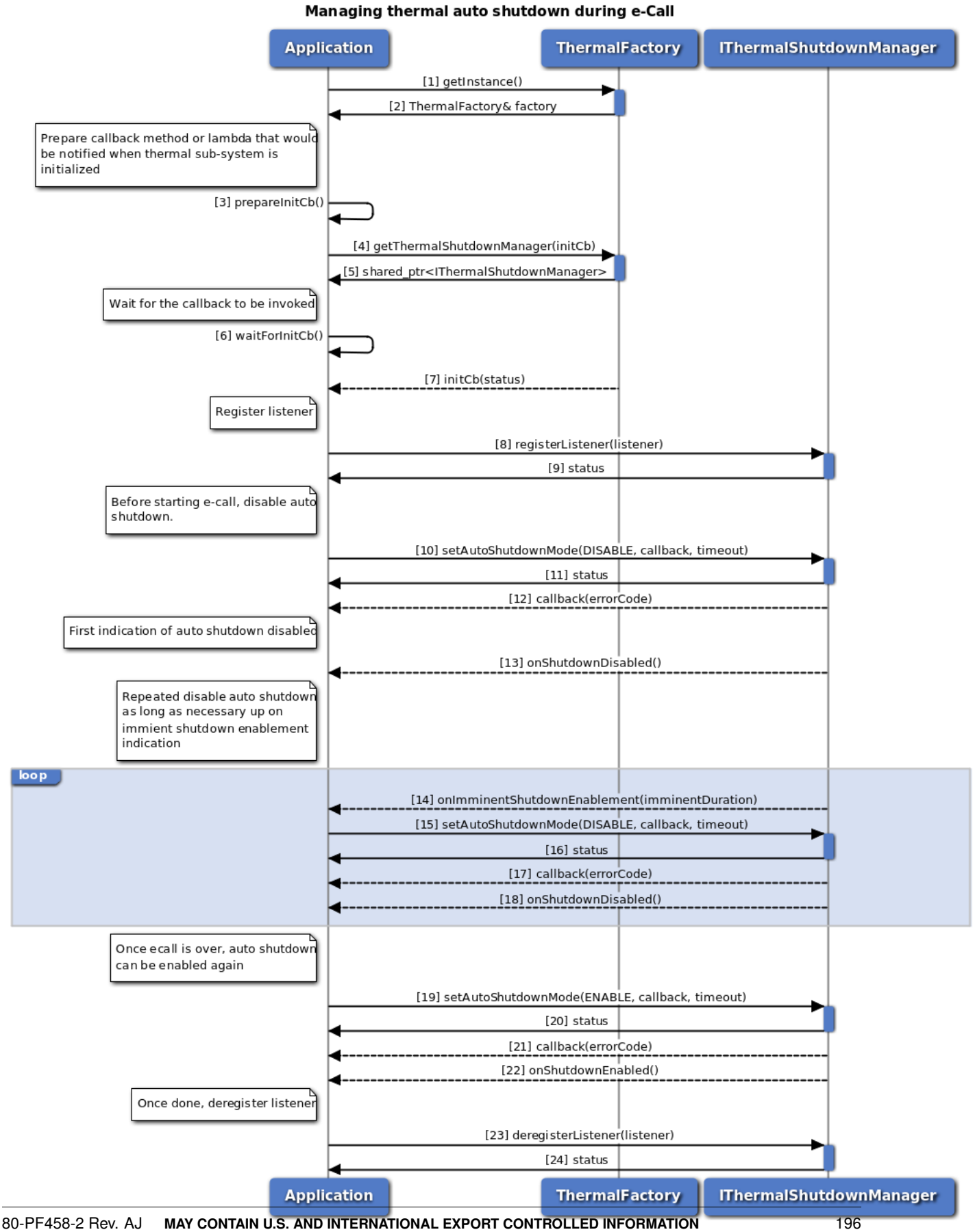

Figure 3-115 Call flow to manage thermal auto-shutdown from an eCall application

1. Application requests for an instance of thermal factory.
2. Thermal factory instance is received by the application.
3. Application prepares the callback which would be called on initialization of thermal shutdown manager.
4. Application request for thermal shutdown manager from thermal factory.
5. Thermal factory returns an instance of the thermal shutdown manager object.
6. Application waits for initalization callback to be called.
7. Thermal factory invokes the callback once initialization completes.
8. Application can register a listener for getting notifications on Thermal auto-shutdown mode updates.
9. Status of register listener i.e. either SUCCESS or FAILED will be returned to the application.
10. Application disables auto-shutdown using setAutoShutdownMode API, to prevent a possible thermal auto-shutdown during eCall.
11. Application receives synchronous status which indicates if the request was sent successfully.
12. Optionally, the response to setAutoShutdownMode request can be received by the application.
13. Application receives a notification that thermal auto-shutdown mode is disabled.
14. Application receives an imminent auto-shutdown enable notification and system will attempt to enable auto-shutdown after a certain period. This notification is received if application does not enable auto-shutdown due to an active eCall.
15. If the eCall is still active, the application disables auto-shutdown before it gets enabled automatically.
16. Application receives synchronous status which indicates if the request was sent successfully.
17. Optionally, the response to setAutoShutdownMode request can be received by the application.
18. Application receives a notification that thermal auto-shutdown mode is disabled. Steps 14 to 18 are repeated as long as the eCall is active.
19. When the eCall is completed, the application immediately enables auto-shutdown using setAutoShutdownMode API.
20. Application receives synchronous status which indicates if the request was sent successfully.
21. Optionally, the response to setAutoShutdownMode request can be received by the application.
22. Application receives a notification that thermal auto-shutdown mode is enabled.
23. Application can remove listener.
24. Status of remove listener i.e. either SUCCESS or FAILED will be returned to the application.

3.9 TCU Activity Management

An application can get the appropriate TCU-activity manager (i.e. slave or master) object from the power factory. The TCU-activity manager configured as the master is responsible for triggering state transitions. TCU-activity manager configured as a slave is responsible for listening to state change indications and acknowledging when it performs necessary tasks and prepares for the state transition. A machine in this power management framework represents an application processor subsystem or a host/guest virtual machine on hypervisor based platforms.

- Only one master is allowed in the system, and currently we only support allowing the master on the primary/host machine and not on the guest virtual machine.
- It is expected that all processes interested in a TCU-activity state change should register as slaves.
- When the master changes the TCU-activate state, slaves connected to the impacted machine are notified.
- Master can trigger the TCU-activity state change of a specific machine or all machines at once.
- If the slave wants to differentiate between a state change indication that is the result of a trigger for all machines or a trigger for its specific machines, it can be detected using the machine name provided in the listener API.
- When the master triggers an all machines TCU-activity state change, only the machines that are not in the desired state will undergo the state transition, and the slaves to those machines will be notified.
- In the case of
  - suspend or shutdown trigger:
    - After becoming ready for state change, all slave clients should acknowledge back.
    - The master client will get notification about the consolidated acknowledgement status of all slave clients.
    - On getting a successful consolidated acknowledgement from all the slaves for the suspend trigger, the power framework allows the respective machine to suspend. On getting a successful consolidated acknowledgement from all the slaves for the shutdown trigger, the power framework triggers the respective machine shutdown without waiting further.
    - If the slave client sends a NACK to indicate that it is not ready for state transition or fails to acknowledge before the configured time, then the master will get to know via a consolidated acknowledgement / slave acknowledgement status notification.
    - In such failed cases, if the master wants to stop the state transition considering the information in the consolidated acknowledgement, then the master is allowed to trigger a new TCU-activity state change, or else the state transition will proceed after the configured timeout.
  - resume trigger:
    - Power framework will prevent the respective machine from going into suspend.
    - No acknowledgement will be required from slave clients and the master will not be getting consolidated acknowledgement / slave acknowledgement as machine will be already resumed.

When the application is notified about the service being unavailable, the TCU-activity state notifications will be inactive. After the service becomes available, the existing listener registrations will be maintained.

3.9.1 Call flow to register/remove listener for TCU-activity manager

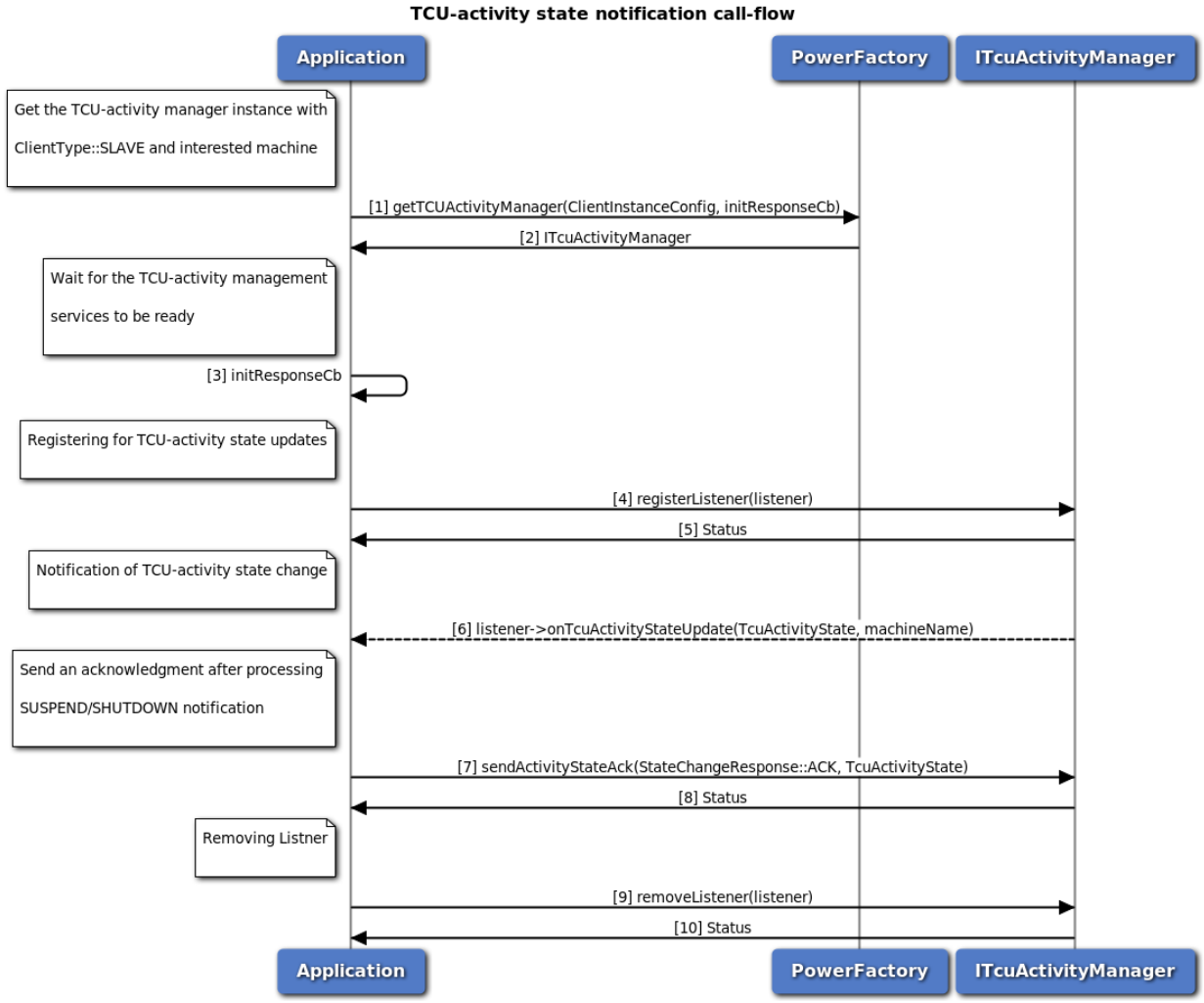

Figure 3-116 Call flow to register/remove listener for TCU-activity manager

1. Application requests power factory for TCU-activity manager object, with clientType as slave. slave clients set up to listen to their LOCAL_MACHINE. Also, it is recommended to give a unique name to each slave client so that client can be identified later in case of failure.
2. Power factory returns ITcuActivityManager object using which application will register or remove a listener.
3. Wait for the TCU-activity management services to be ready.
4. Application can register a listener for getting notifications on TCU-activity state updates on the machine. Some of the listener APIs are designed for slave clients and others for master clients. Listener APIs intended for the master client will not be sent to the slave client, and listner APIs intended for the slave client will not be sent for the master client.
5. Status of register listener i.e. either SUCCESS or FAILED will be returned to the application.
6. Application will get TCU-activity state notifications like suspend, resume and shutdown.

7. Application will send one acknowledgement, after processing(save any required information) suspend/shutdown notifications. This indicates the readiness of application for state-transition. However the TCU-activity management service doesn't wait for acknowledgement indefinitely, before performing the state transition. In the acknowledgement, slave can specify the machine name received in the notification in the previous step.
8. Application receives synchronous status which indicates if the acknowledgement was sent successfully.
9. Application can remove listener.
10. Status of remove listener i.e. either SUCCESS or FAILED will be returned to the application.

3.9.2 Call flow to set the TCU-activity state

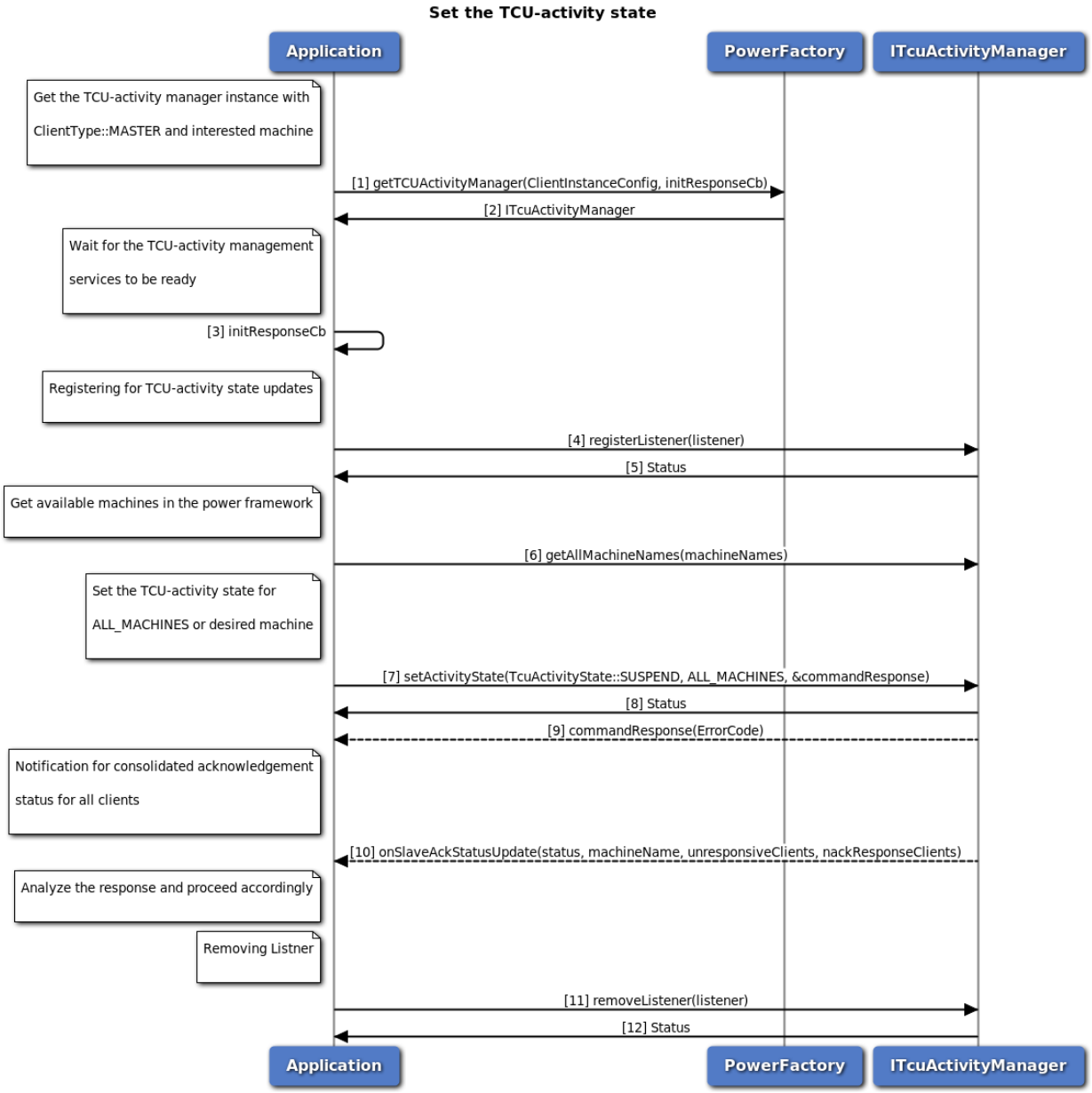

Figure 3-117 Call flow to set the TCU-activity state

1. Application requests power factory for TCU-activity manager object, with clientType as master.
2. Power factory returns ITcuActivityManager object using which application will set the TCU-activity state.
3. Wait for the TCU-activity management services to be ready.
4. Application can register a listener for getting notifications on TCU-activity state.
5. Status of register listener i.e. either SUCCESS or FAILED will be returned to the application

6. Get a list of available machine names in the power framework if the user is interested in the state transition of a specific machine.
7. Application can set the TCU-activity state to suspend, resume or shutdown.
8. Application receives synchronous status which indicates if the request was sent successfully.
9. Optionally, the response to setActivityState request can be received by the application.
10. The application waits for consolidated acknowledgement status and analyzes the response. If status is not SUCCESS and the master expects to stop state transition considering the provided information, then call setActivityState and revert state, or else state transition will proceed after the configured timeout in /etc/power_state.conf.
11. Application can remove listener.
12. Status of remove listener i.e. either SUCCESS or FAILED will be returned to the application.

3.10 Remote SIM call flow

Application will get the remote SIM manager object from phone factory. The application must register a listener to receive commands/messages from the modem to send to the SIM. After sending the connection available message, a onCardConnect() notification tells the application to connect to the SIM and perform an Answer to Reset. After sending the card reset message (with the AtR bytes), APDU messages will begin

to be sent/received.

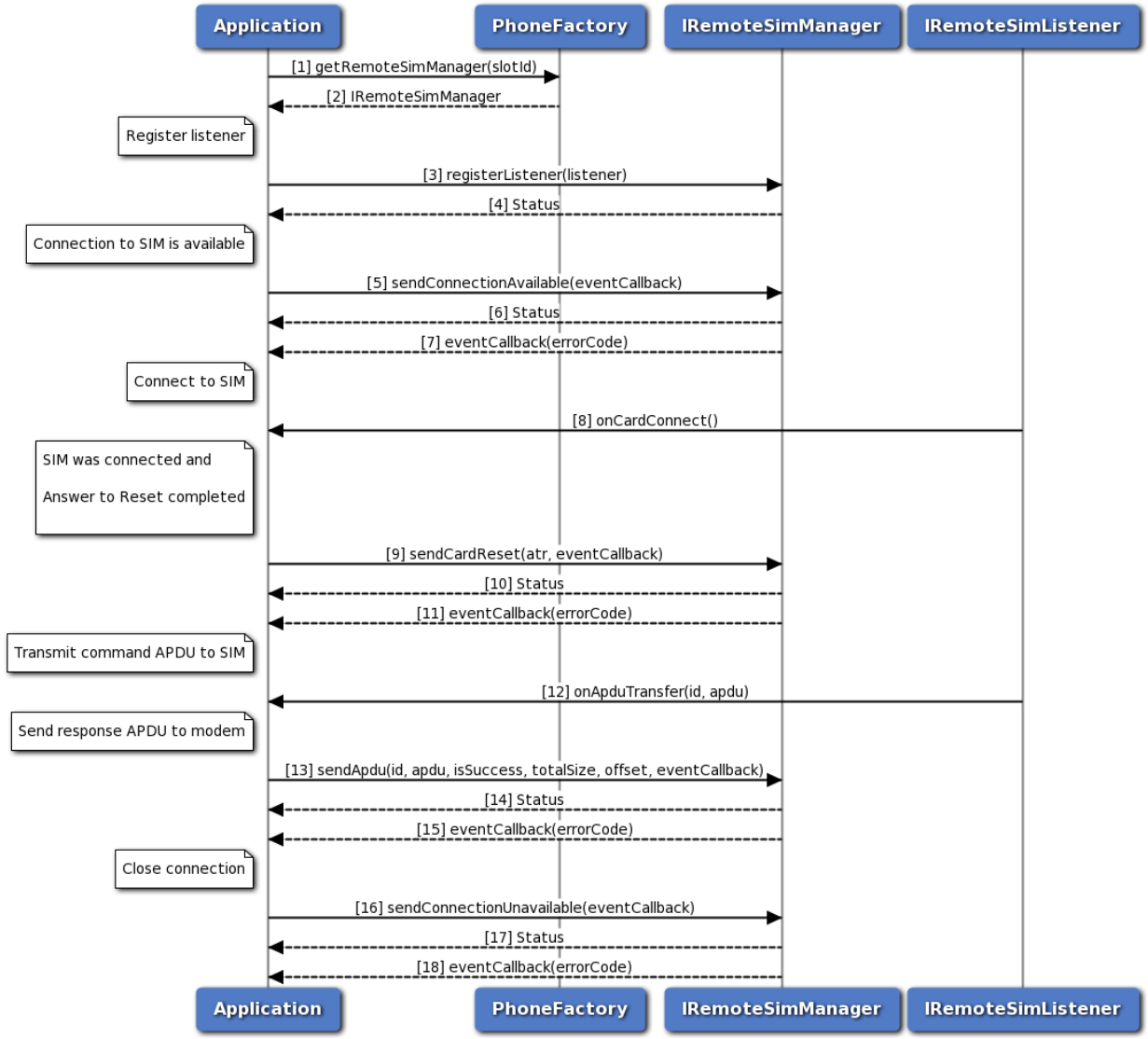

Figure 3-118 Remote SIM call flow

1. Application requests remote SIM manager object from phone factory, specifying a slot id.
2. Phone factory returns IRemoteSimManager object.
3. Application registers a listener to receive commands/messages from the modem to send to the SIM.
4. Status of register listener i.e. either SUCCESS or FAILED will be returned to the application.
5. Application sends a connection available message indicating that a SIM is available for use.
6. Status of send connection available i.e. either SUCCESS or FAILED will be returned to the application.
7. Optionally, the response to send connection available request can be received by the application.
8. Application will receive a card connect notification by the listener.

9. After the application successfully connects to the SIM and requests an AtR, it sends a card reset message with the AtR bytes.
10. Status of send card reset i.e. either SUCCESS or FAILED will be returned to the application.
11. Optionally, the response to send card reset request can be received by the application.
12. Application will receive an APDU transfer notification by the listener (with APDU message id).
13. After forwarding the APDU transfer to the SIM and receiving the response, application will send APDU response.
14. Status of send APDU i.e. either SUCCESS or FAILED will be returned to the application.
15. Optionally, the response to send APDU request can be received by the application.
16. To close the connection, application will send connection unavailable message.
17. Status of send connection unavailable i.e. either SUCCESS or FAILED will be returned to the application.
18. Optionally, the response to send connection unavailable can be received by the application.

3.11 Modem Config Call Flow

Modem Config manager provides APIs to request all configs from modem, load/delete modem config files from modem's storage, activate/deactivate a modem config file, get the active config details, set and get auto config selection mode. It also has listener interface for notifications for config activation update status. Application will get the Modem Config manager object from config factory. The application can register a listener for updates reagrding modem config activation.

3.11.1 Call flow to load and activate a modem config file.

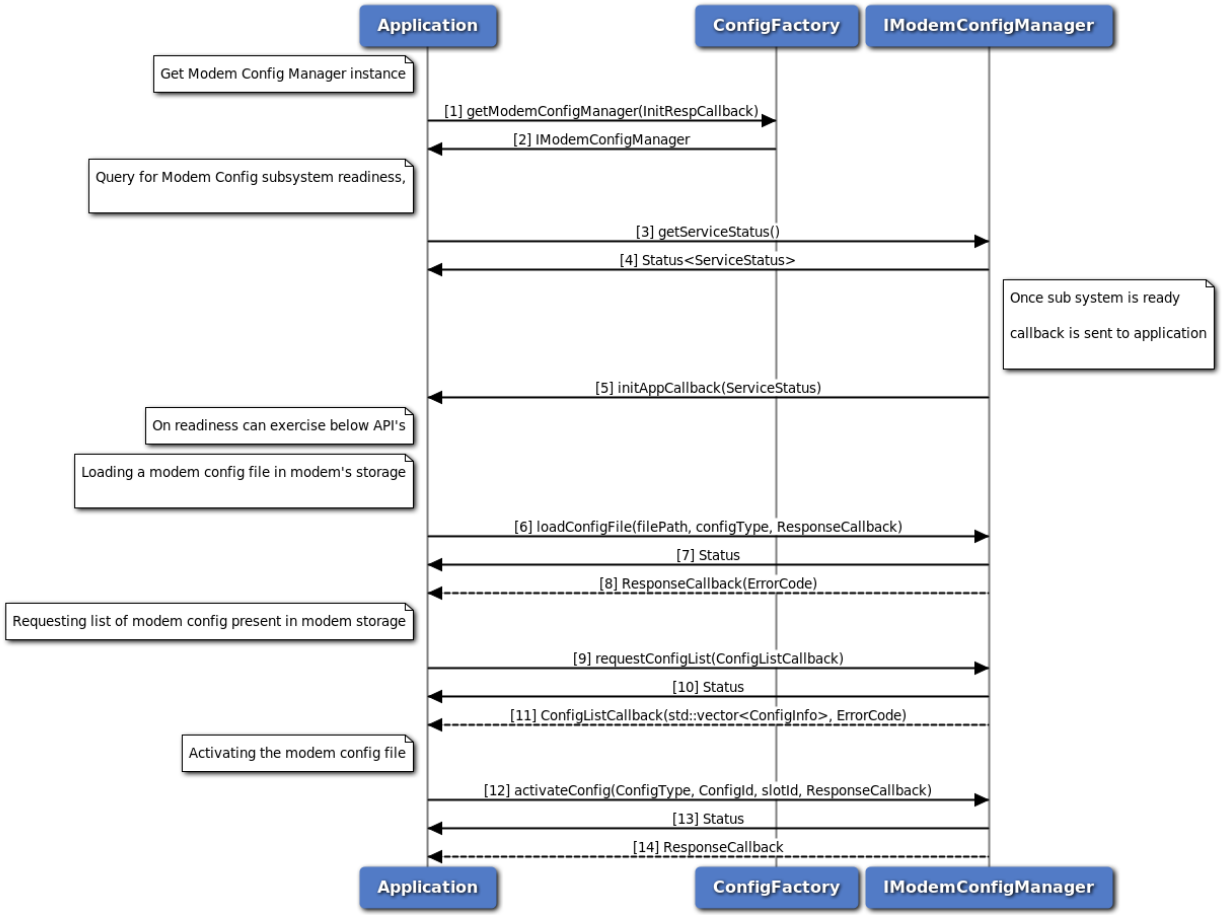

Figure 3-119 Modem Config load and activate call flow

1. Application requests Config factory for ModemConfig Manager and passes callback pointer.
2. Config factory return IModemConfigManager object to application.
3. Application can use IModemConfigManager::getServiceStatus to determine the state of sub system.
4. The application receives the ServiceStatus of sub system which indicates the state of service.
5. IModemConfigManager notifies the application when the subsystem is ready through the callback mechanism.
6. Application sends a request to load config file in modem's storage.
7. Application receives synchronous Status which indicates if the request to load config file was sent successfully.
8. Application is notified of the Status of the loadConfigFile request (either SUCCESS or FAILED) via the application-supplied callback.
9. Application sends a request to get list of all modem configs from modem's storage.
10. Application receives synchronous Status which indicates if the request to get config list was sent

successfully.

11. Application is notified of the Status of the requestConfigList request (either SUCCESS or FAILED) via the application-supplied callback along with list of modem configs.
12. Application sends a request to activate config file.
13. Application receives synchronous Status which indicates if the request to activate config file was sent successfully.
14. Application is notified of the Status of the activateConfig request (either SUCCESS or FAILED) via the application-supplied callback.

3.11.2 Call flow to deactivate and delete a modem config file.

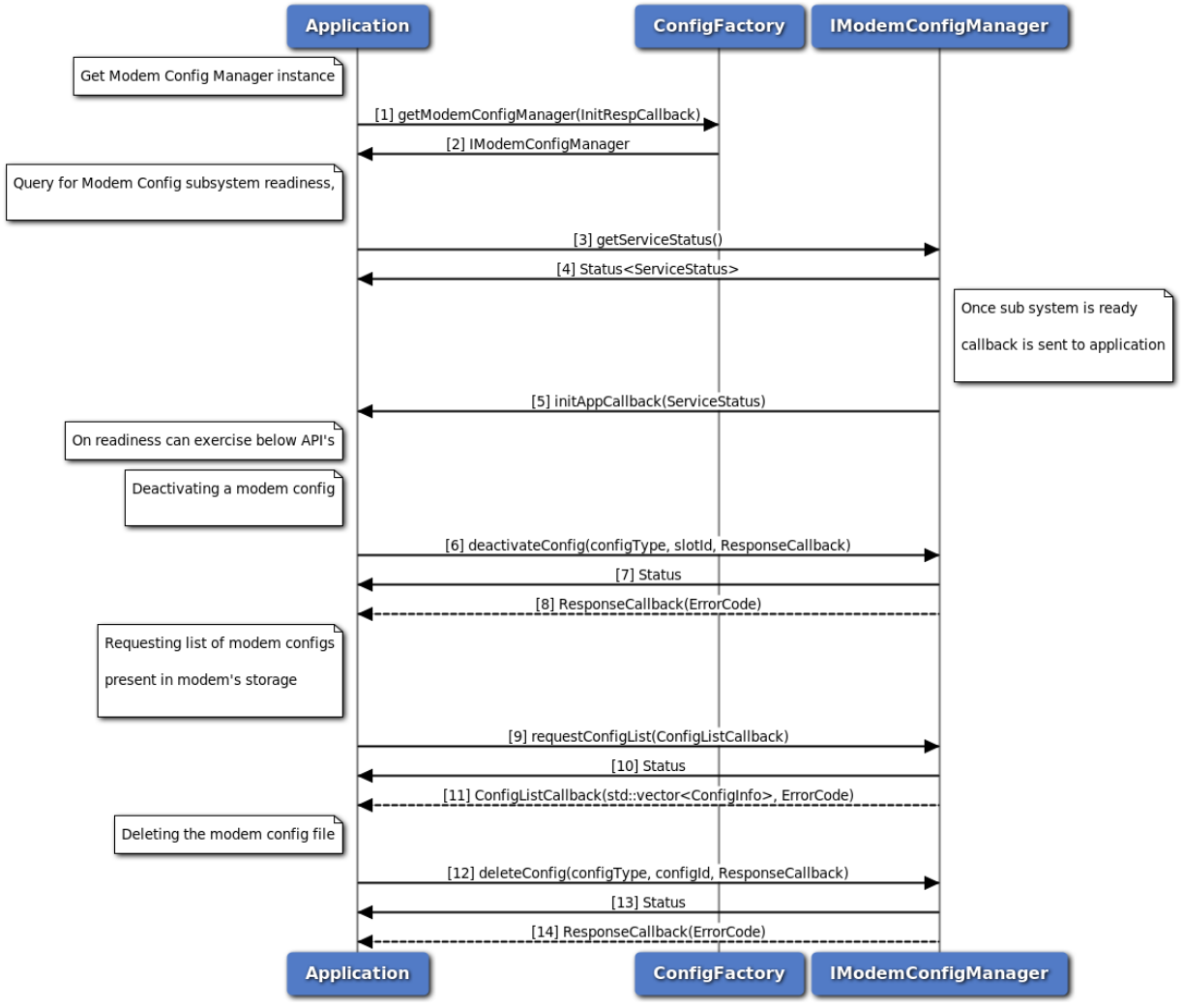

Figure 3-120 Modem Config deactivate and delete Call Flow

1. Application requests Config factory for ModemConfig Manager and passes callback pointer.
2. Config factory return IModemConfigManager object to application.

3. Application can use IModemConfigManager::getServiceStatus to determine the state of sub system.
4. The application receives the ServiceStatus of sub system which indicates the state of service.
5. IModemConfigManager notifies the application when the subsystem is ready through the callback mechanism.
6. Application sends a request to deactivate config file.
7. Application receives synchronous Status which indicates if the request to deactivate config file was sent successfully.
8. Application is notified of the Status of the deactivateConfig request (either SUCCESS or FAILED) via the application-supplied callback.
9. Application sends a request to get list of all modem configs from modem's storage.
10. Application receives synchronous Status which indicates if the request to get config list was sent successfully.
11. Application is notified of the Status of the requestConfigList request (either SUCCESS or FAILED) via the application-supplied callback along with list of modem configs.
12. Application sends a request to delete config file.
13. Application receives synchronous Status which indicates if the request to delete config file was sent successfully.
14. Application is notified of the Status of the deleteConfig request (either SUCCESS or FAILED) via the application-supplied callback.

3.11.3 Call flow to set and get config auto selection mode

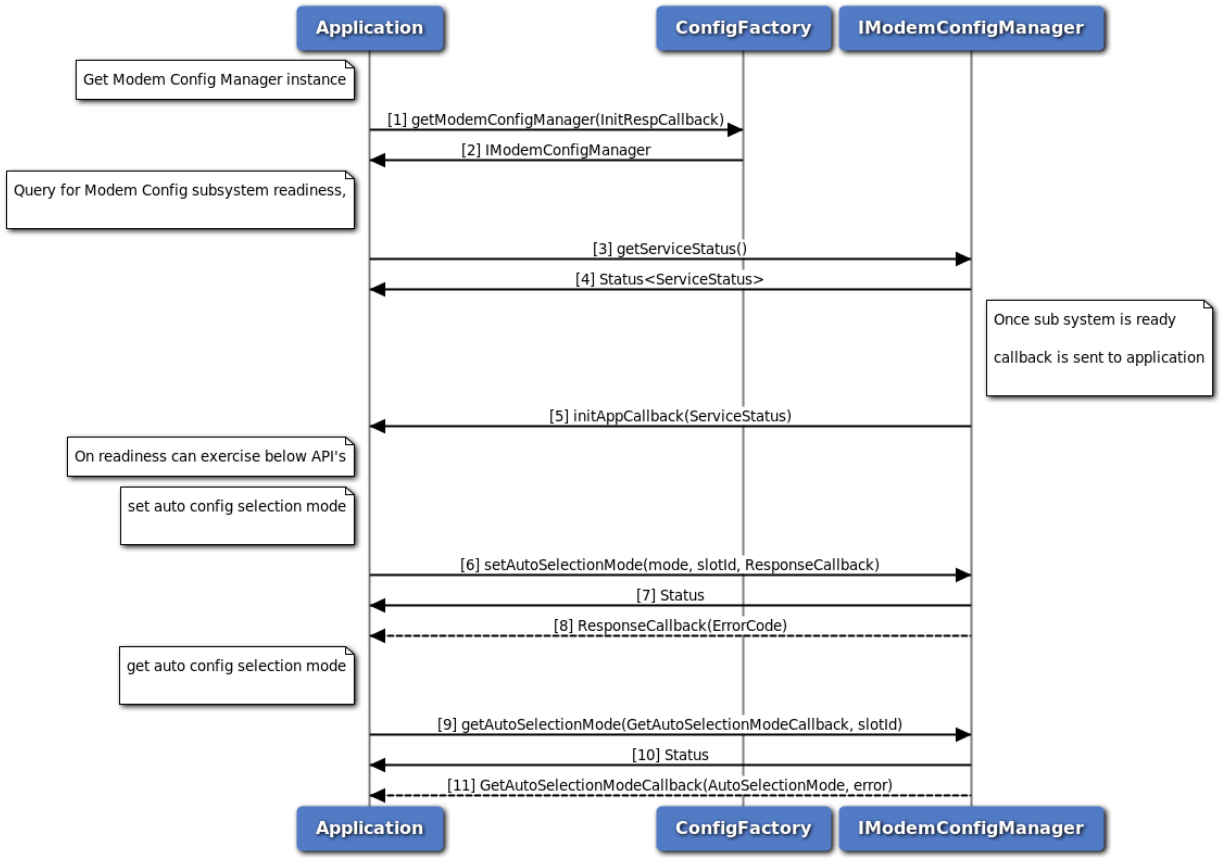

Figure 3-121 Modem Config get and set Auto Selection Mode Call Flow

1. Application requests Config factory for ModemConfig Manager and passes callback pointer.
2. Config factory return IModemConfigManager object to application.
3. Application can use IModemConfigManager::getServiceStatus to determine the state of sub system.
4. The application receives the ServiceStatus of sub system which indicates the state of service.
5. IModemConfigManager notifies the application when the subsystem is ready through the callback mechanism.
6. Application sends a request to set config auto selection mode.
7. Application receives synchronous Status which indicates if the request to set config auto selection mode was sent successfully.
8. Application is notified of the Status of the request setAutoSelectionMode (either SUCCESS or FAILED) via the application-supplied callback.
9. Application sends a request to get config auto selection mode.
10. Application receives synchronous Status which indicates if the request to get config auto selection mode was sent successfully.

11. Application is notified of the Status of the request setAutoSelectionMode (either SUCCESS or FAILED) via the application-supplied callback, along with mode and slot id.

3.12 Sensor

The sensor sub-system provides APIs to configure and acquire continuous stream of data from an underlying sensor, create multiple clients for a given sensor, each of which can have their own configuration (sampling rate, batch count) for data acquisition.

The sensor sub-system APIs are synchronous in nature.

3.12.1 Call flow for sensor sub-system start-up

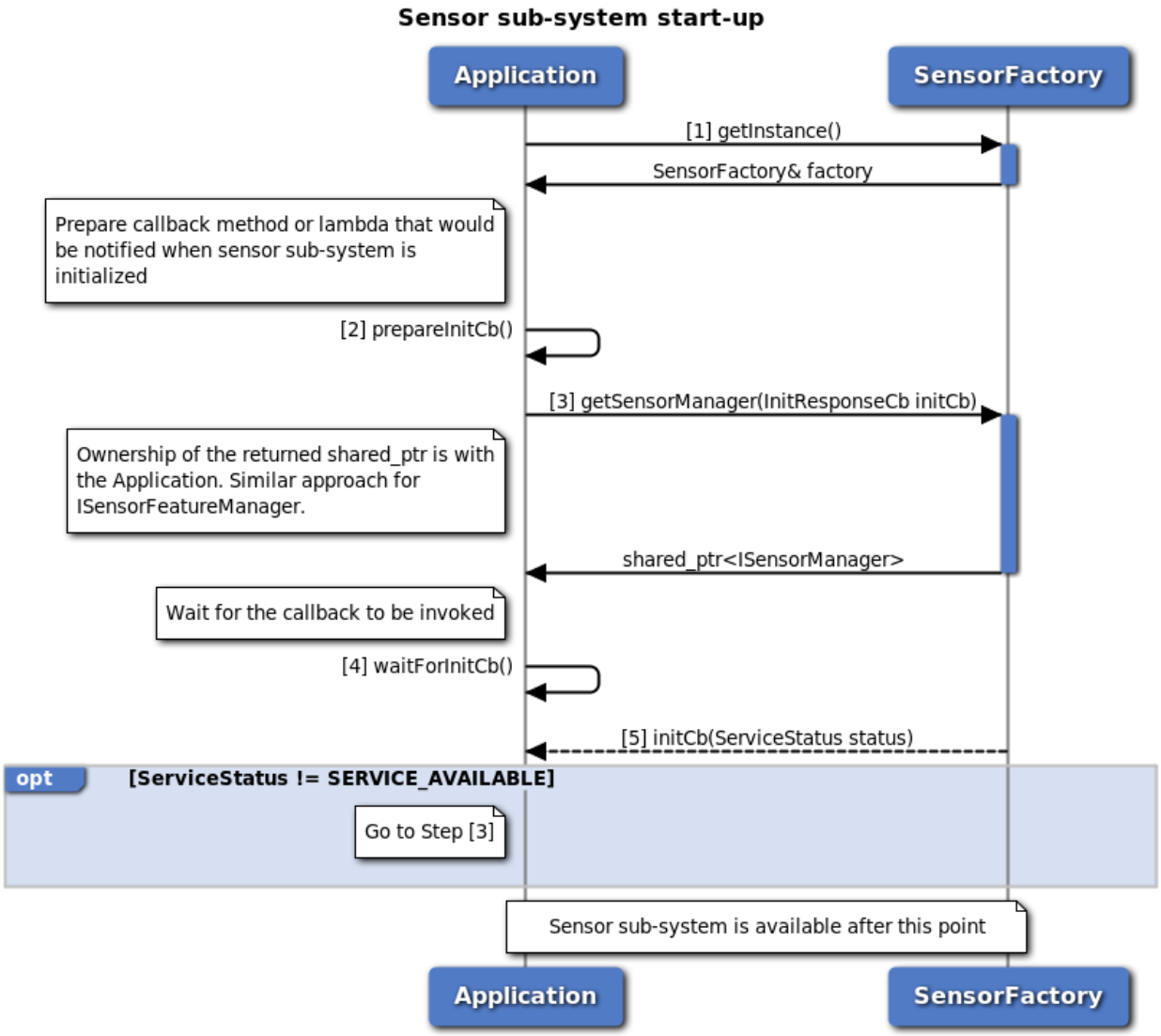

Figure 3-122 Sensor sub-system start-up call flow

1. Get the reference to the SensorFactory, with which we can further acquire other sensor sub-system objects.
2. Prepare an initialization callback method or lambda which will be called by the sensor sub-system once the initialization is complete.
3. Request for the SensorManager from SensorFactory and provide the initalization callback. Retain the SensorManager shared_ptr as long as necessary. SensorFactory does not hold on to the returned instance. If the received shared_ptr is released, SensorManager would be destroyed and requesting SensorFactory for SensorManager again would result in the creation of a new instance. Similar is the approach for SensorFeatureManager.
4. Wait for the initialization callback to be invoked.
5. The sensor sub-system invokes the callback once the underlying sub-system and sensor framework is available for usage. If the service status is notified as SERVICE_FAILED, retry intitialization starting with step (3). If the service status is notified as SERVICE_AVAILABLE, the sensor sub-system is ready for usage.

3.12.2 Call flow for sensor data acquisition

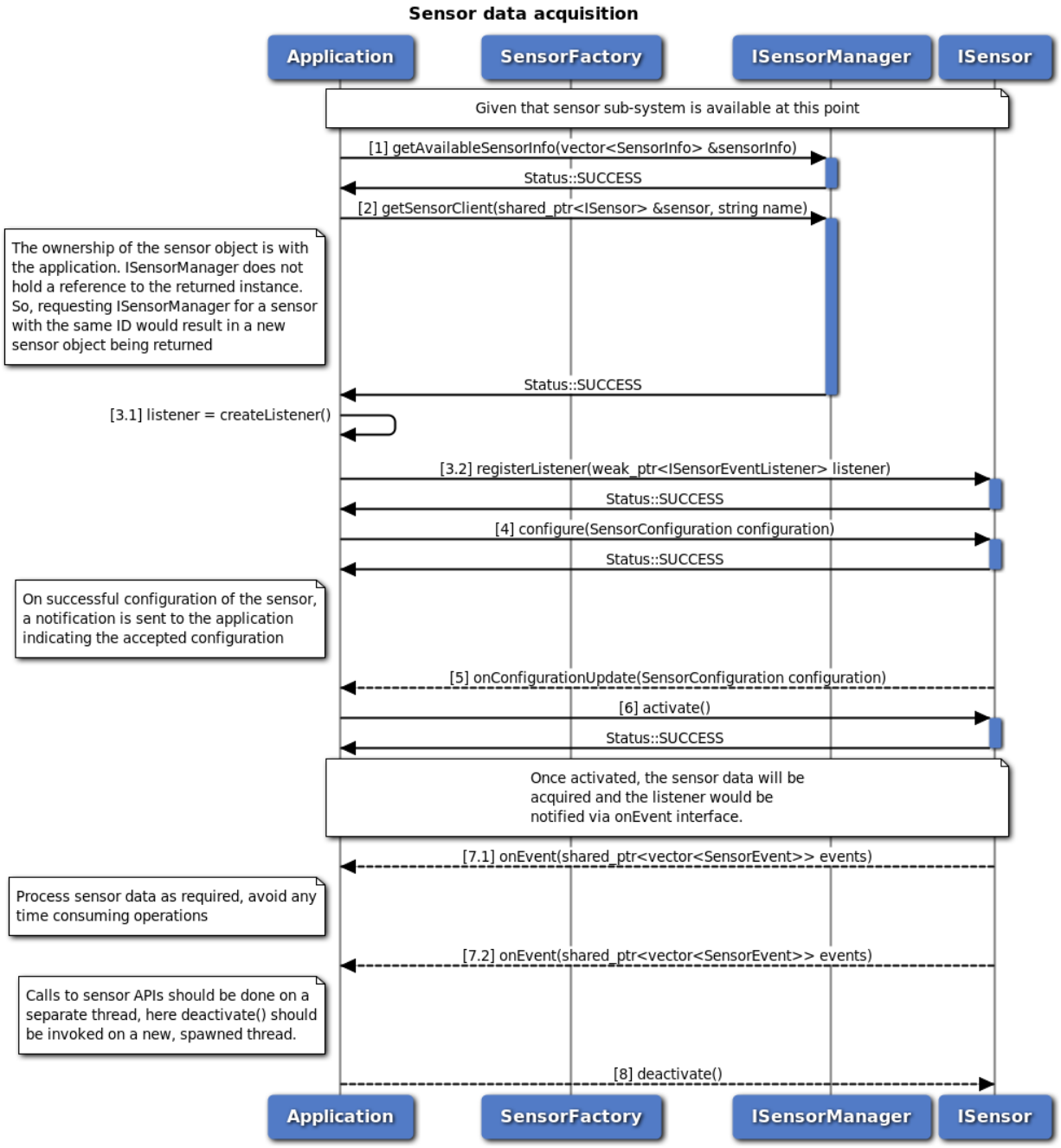

Figure 3-123 Sensor data acquisition call flow

The sensor sub-system should have been initialized successfully with SERVICE_AVAILABLE as a pre-requisite for sensor data acquisition and a valid SensorManager object is available.

1. Get the information about available sensors from ISensorManager. The information will be provided in the sensorInfo parameter that would be passed by reference.
2. Given the information about different sensors, identify the required sensor (name of the sensor) using

the provided attributes - type, name, vendor or required sampling frequency. Having identified the required sensor, request for the sensor object from ISensorManager with the required name of the sensor. If the request was successful, the provided reference to shared_ptr<ISensorClient> would be set by the ISensorManager which can be used to further configure and acquire data from the sensor. The ownership of the sensor object is with the application. SensorManager does not hold a reference to the returned instance. So, requesting ISensorManager for a sensor with the same name would result in a new sensor object being returned.

3. Create a listener of type ISensorEventListener which would receive notifications about updates to sensor configuration and sensor events. Register the created listener with the sensor object.
4. Create the desired sensor configuration. For continous data acquisition samplingRate and batchCount are necessary attributes to be set in the configuration. Be sure to set the validityMask in the SensorConfiguration structure.
5. On successful configuration, a notification is sent to the registered listeners indicating the configuration set.
6. Activate the sensor for acquiring sensor data.
7. When the sensor is activated successfully, the sensor data is sent to the registered listeners.

3.12.3 Call flow for sensor reconfiguration

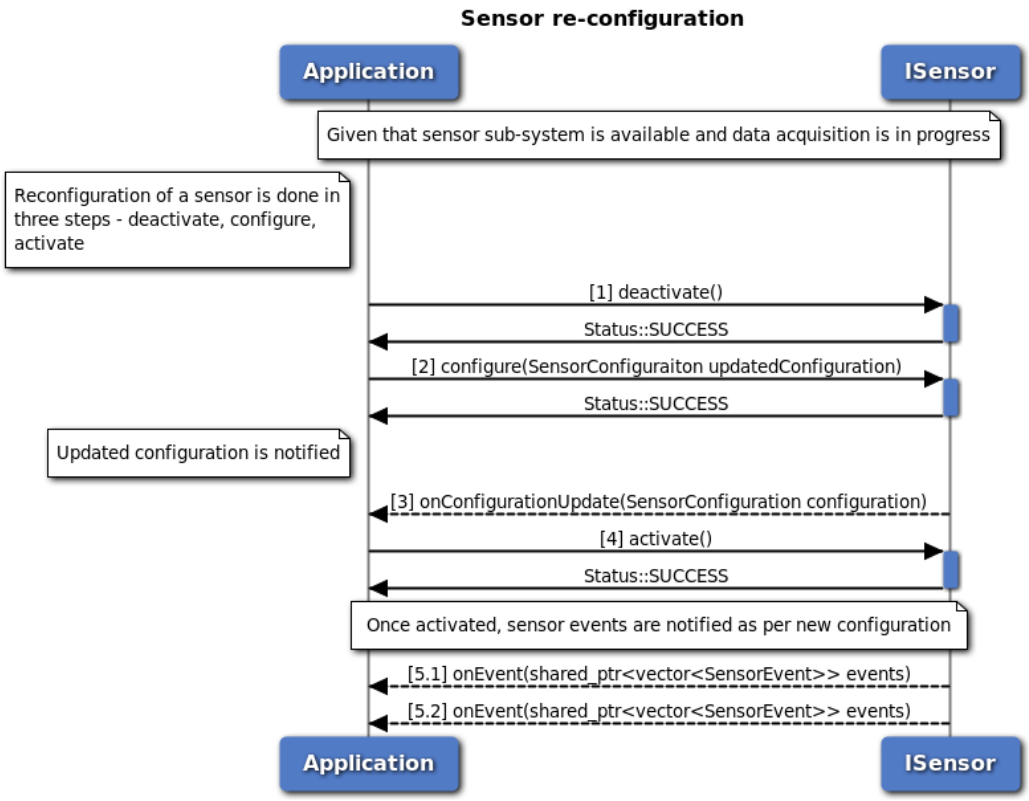

Figure 3-124 Sensor reconfiguration call flow

When the sensor sub-system has been initialized successfully with SERVICE_AVAILABLE and is already activated, reconfiguring the sensors involves the following steps.

1. Deactivate the sensor. This will stop the notifications about sensor events to the registered listeners.
2. Configure the sensor with the required attributes. Be sure to set the validityMask for the all required attributes in SensorConfiguration.
3. The underlying sub-system notifies the registered listeners about the new configuration set.
4. Activate the sensor.
5. When the sensor is activated successfully, the sensor data is sent to the registered listeners as per the new configuration.

3.12.4 Call flow for sensor sub-system cleanup

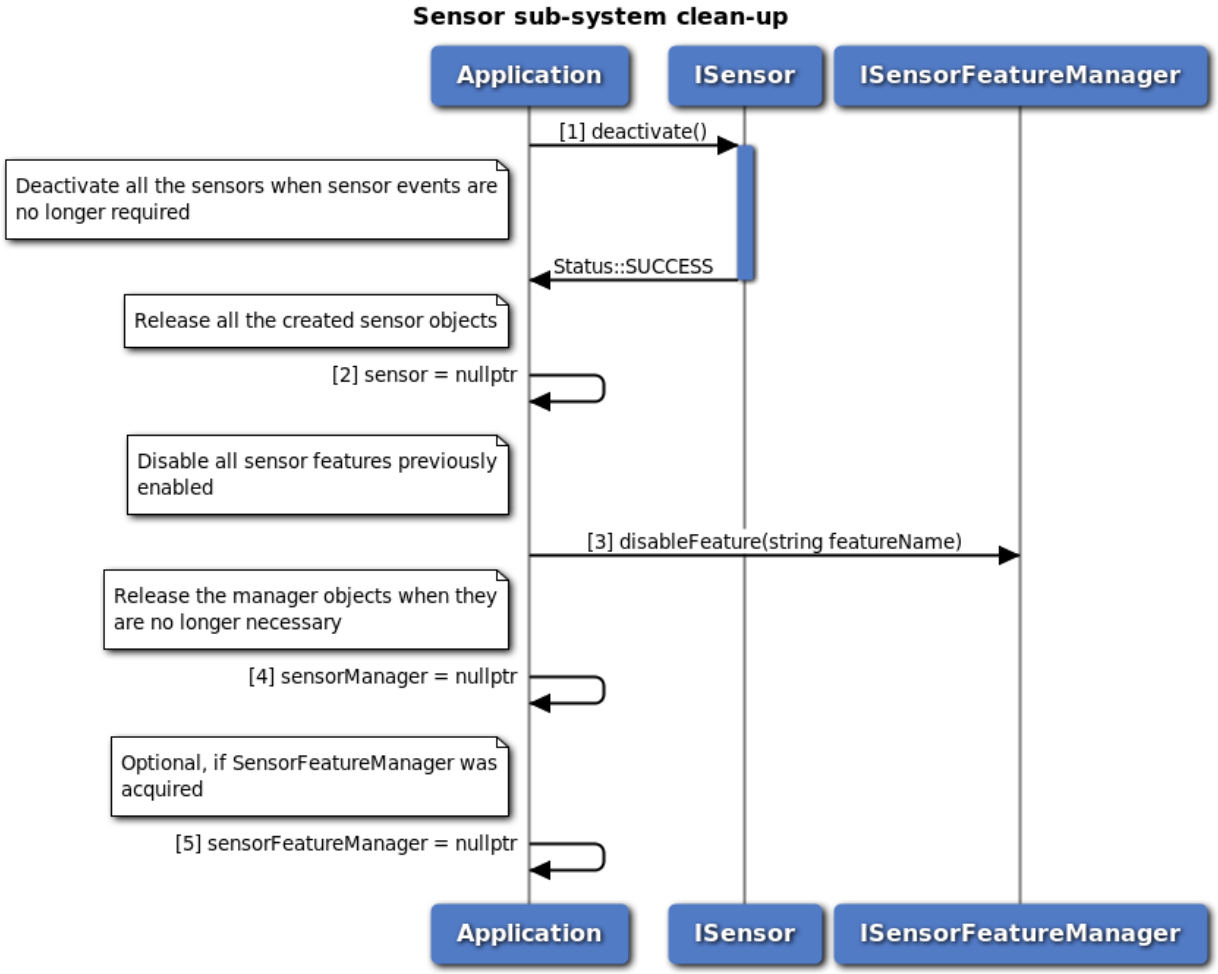

Figure 3-125 Sensor sub-system cleanup call flow

When the sensor sub-system has been initialized successfully with SERVICE_AVAILABLE and sensor objects have been created, the following steps ensure a cleanup of the sensor sub-system.

1. Deactivate all the sensors. With this, the registered listeners will no longer be notified of the sensor events.
2. Release all the sensor objects created by setting them to nullptr. Since the application owns the

objects, this would result in all the sensor objects getting destroyed.

3. Disable all the sensor features that were previously enabled.
4. Release the instance of ISensorManager by setting it to nullptr. Since the application owns the object, this would result in the sensor manager getting destroyed.
5. Release the instance of ISensorFeatureManager by setting it to nullptr. Since the application owns the object, this would result in the sensor feature manager getting destroyed.

3.12.5 Call flow for sensor power control

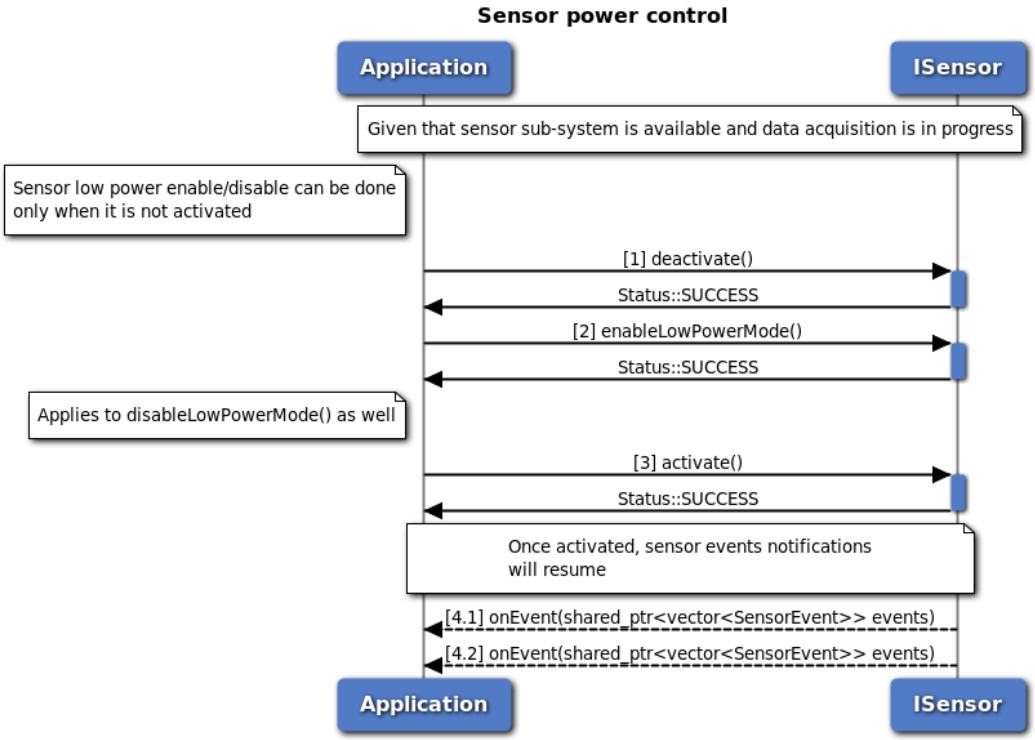

Figure 3-126 Sensor power control cleanup call flow

The below points are to be noted for sensor power control a) Power control is not offered by all sensor manufacturers. If the underlying hardware sensor does not support power control, the power control APIs fail. b) Enabling or disabling low power mode for the sensor is only possible when the sensor is not activated.

For achieving power control, the following steps are to be followed

1. Deactivate the sensor. This will stop the notifications about sensor events to the registered listeners.
2. Perform the required power control by enabling or disabling low power mode for the sensor.
3. Activate the sensor.
4. When the sensor is activated successfully, the sensor data is sent to the registered listeners.

3.12.6 Call flow for sensor feature control

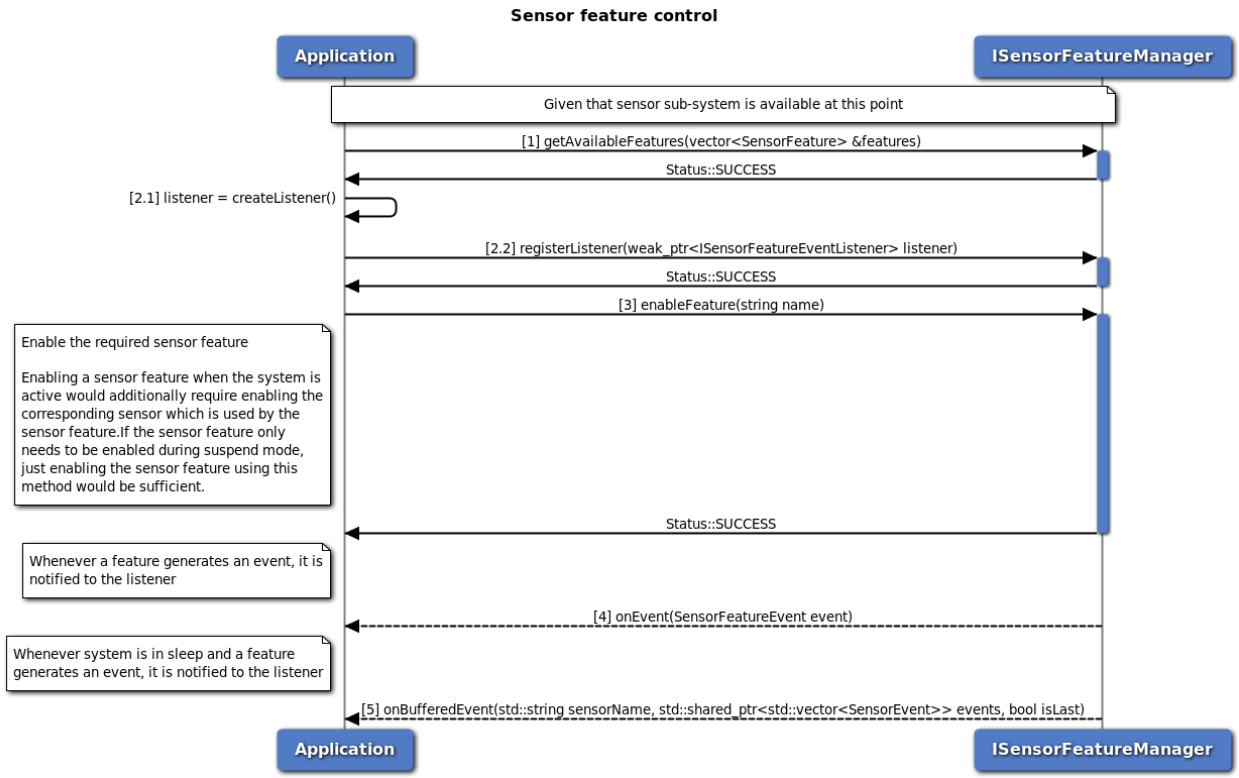

Figure 3-127 Sensor feature control cleanup call flow

The sensor sub-system offers certain features in addition to the data acquisition and these could be features offered by the underlying sensor hardware or the sensor software framework. If there are no features offered collectively, the sensor feature manager initialization would fail.

1. Retrieve a list of features offered by the sensor sub-system and identify the required feature that needs to be enabled. If the feature that needs to be enabled is known, this step is optional.
2. Create a sensor feature event listener, which would be notified of the different events that occur. Register this listener with the sensor feature manager.
3. Enable the required feature.
4. If system is not in sleep mode, Once the feature is enabled and an event related to the feature occurs, the listener is notified.
5. If system is in sleep mode, Once the feature is enabled and an event related to the feature occurs, the listener is notified.

3.13 Platform

The platform sub-system provides APIs to configure and control platform functionalities. This sub-system provides notifications about certain system related events, for instance filesystem events such as EFS restore and backup events.

3.13.1 Call flow for EFS restore notification registration and handling

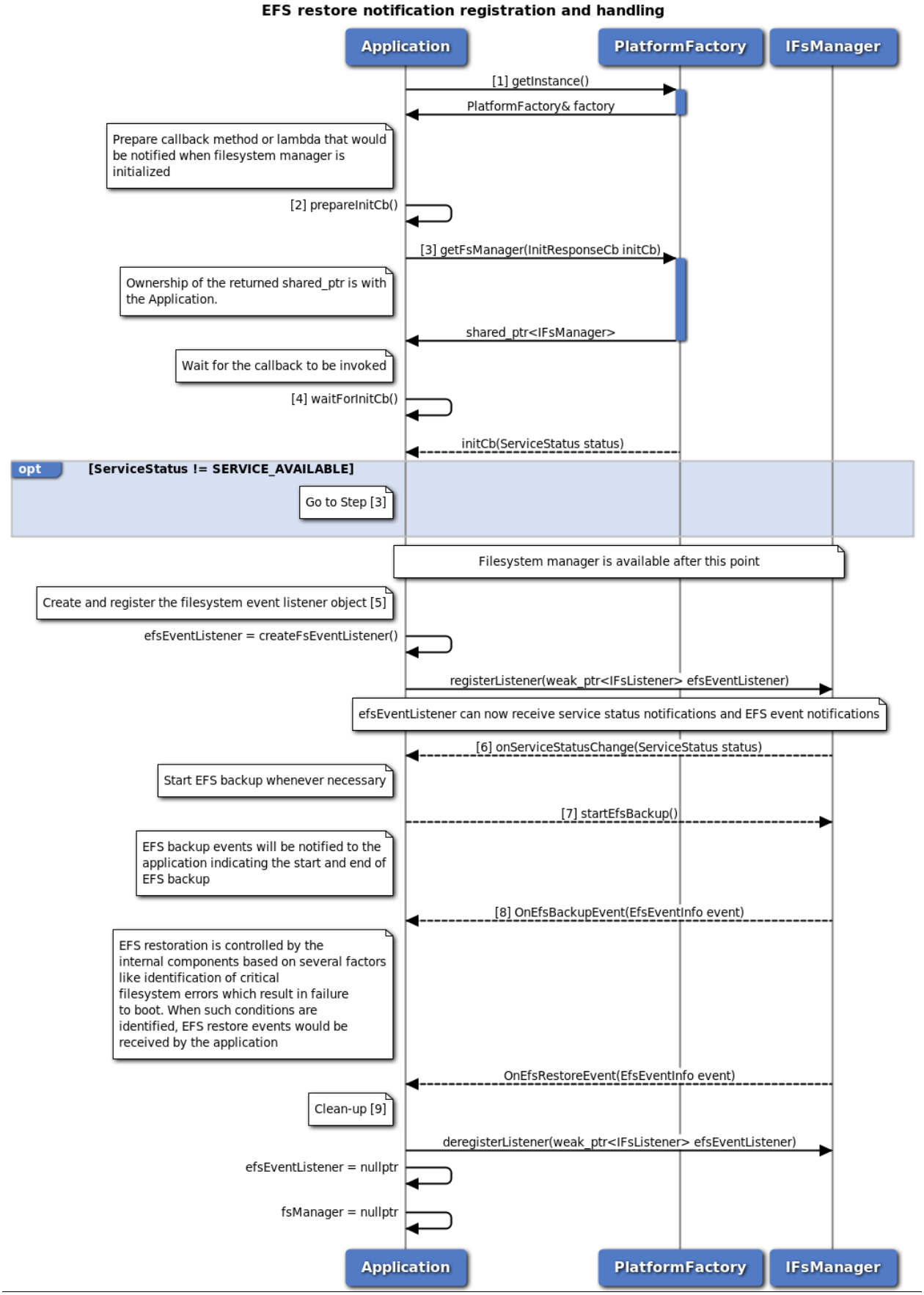

Figure 3-128 EFS restore notification registration and handling call flow

1. Get the reference to the PlatformFactory, with which we can further acquire other sub-system objects.
2. Prepare an initialization callback method or lambda which will be called by the platform sub-system once the initialization is complete.
3. Request for the IFsManager (filesystem manager) object from PlatformFactory and provide the initalization callback. Retain the IFsManager shared_ptr as long as necessary. PlatformFactory does not hold on to the returned instance. If the received shared_ptr is released, FsManager would be destroyed and requesting PlatformFactory for FsManager again would result in the creation of a new instance.
4. Wait for the initialization callback to be invoked. The platform sub-system invokes the callback once the underlying sub-system is available for usage. If the service status is notified as SERVICE_FAILED, retry intitialization starting with step (3). If the service status is notified as SERVICE_AVAILABLE, the filesystem manager is ready for usage.
5. Create an listener object of type IFsListener and register with IFsManager for notifications. Once registered, the listener receives service status notifications and EFS restore event notifications.
6. If a service status notification with status SERVICE_UNAVAILABLE, the application should wait for service to be re-initialized and once done, SERVICE_AVAILABLE will be notified. If the service fails, SERVICE_FAILED is notified and the IFsManager object held is no longer usable. A new object of type IFsManager needs to be re-acquired from the PlatformFactory.
7. When the application finds it appropriate to trigger a EFS backup, startEfsBackup should be invoked, which would return immediately indicating the status of the request
8. Once EFS backup starts and completes, the notifications are sent out to the application. EFS restoration is controlled by the internal components based on several factors like identification of critical filesystem errors which result in failure to boot. When such conditions are identified, EFS restore events would be received by the application via the OnEfsRestoreEvent method and the application can make use of the information appropriately. The EFS restore notification also has information if the restore started or ended and if the restore was successful or a failure.
9. Once clean-up is necessary, deregister the registered listener, set all shared pointers to nullptr. This will make the underlying sub-system relinquish resources that are no longer necessary.

3.13.2 Call flow of control filesystem for ECALL operation

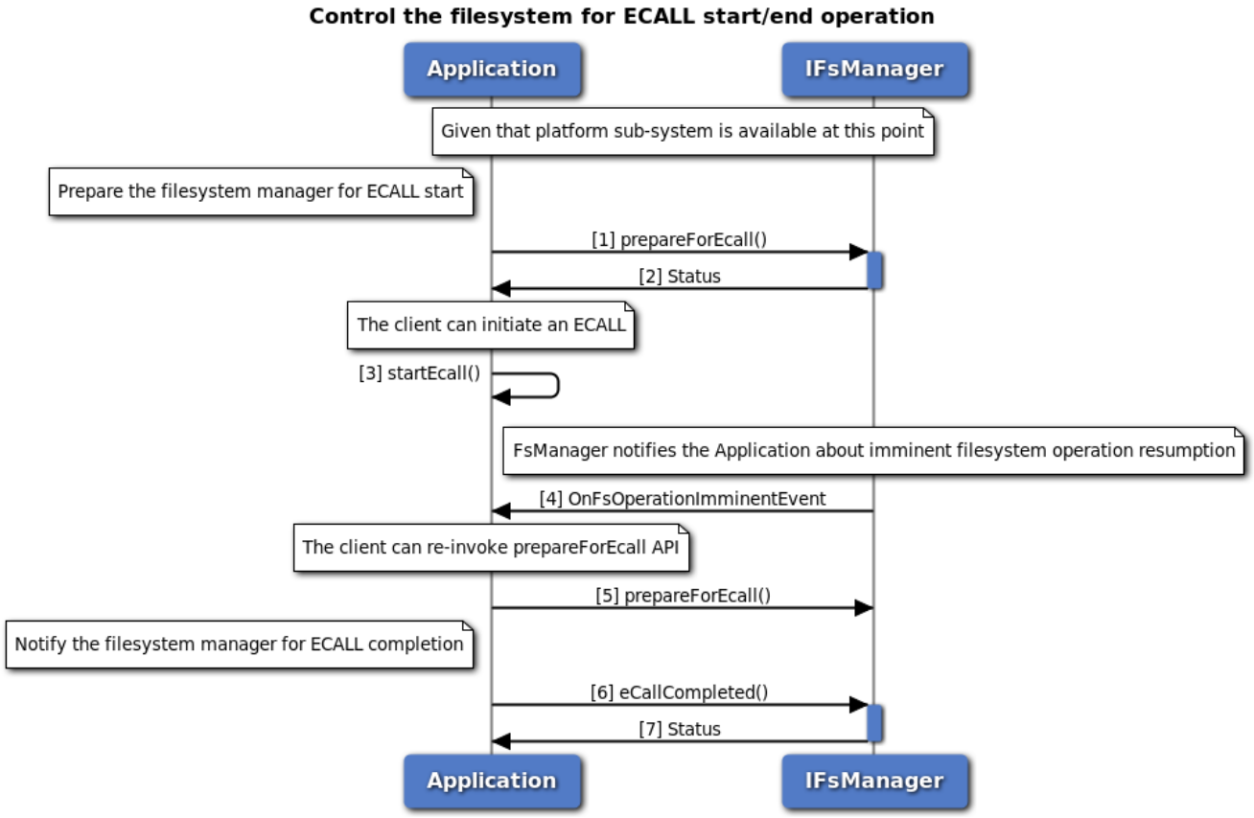

Figure 3-129 Control the filesystem for ECALL operation call flow

The platform sub-system should have been initialized successfully with SERVICE_AVAILABLE as a pre-requisite for any filesystem operations and a valid FilesystemManager object is available.

1. Before initiate an eCall, the application should prepare the filesystem for eCall operation, prepareForEcall should be invoked. If the status of this request fails, the client could re-invoke during an eCall ongoing.
2. The status of the request would return immediately indicating the preparation of the filesystem operation.
3. The client should start an eCall immediately even if the prepare for eCall request failed.
4. The filesystem manager shall notify the application when filesystem operation is about to resume.
5. If the client wants to suspend the filesystem operation to continue the eCall, they should invoke the prepareForEcall API.
6. Once an eCall completes, the client should notify eCall completion to the filesystem manager. eCallCompleted should be invoked.
7. The status of the request would return immediately indication the completion of the filesystem operation.

3.13.3 Call flow of control filesystem for OTA operation

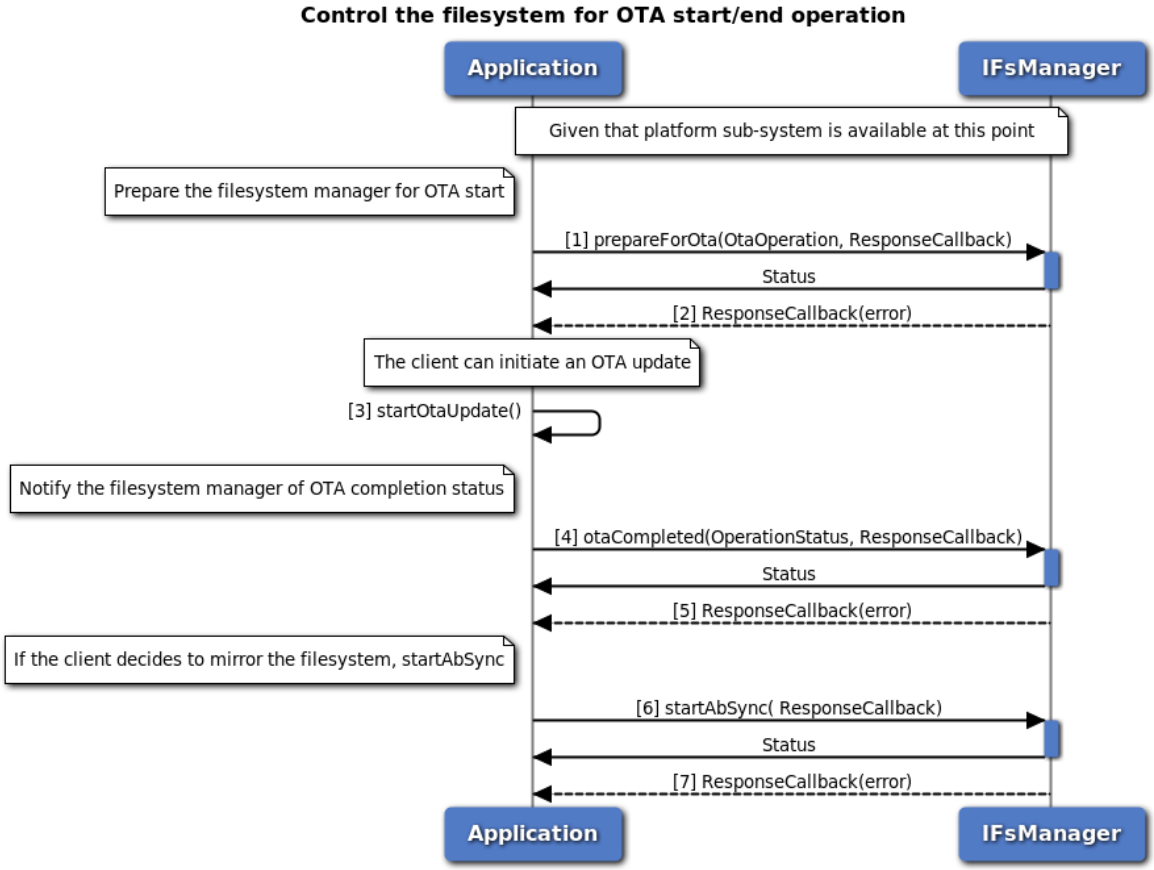

Figure 3-130 Control the filesystem for OTA operation call flow

The platform sub-system should have been initialized successfully with SERVICE_AVAILABLE as a pre-requisite for any filesystem operations and a valid FilesystemManager object is available.

1. Before initiate an OTA update, the application should prepare the filesystem for the OTA operation, prepareForOta should be invoked, which would return immediately indicating the status of the request.
2. Once the filesystem has prepared for the OTA operation, the filesystem manager would invoke the callback which indicates the response of the API.
3. Once the filesystem has prepared for the OTA operation, the client could initiate an OTA update.
4. On OTA update completion, the client should notify the filesystem manager of the OTA completion status, otaCompleted should be invoked, which would return immediately indicating the status of the request.
5. The filesystem manager would intern update the OTA status, the callback would invoked which indicates the response of the API.
6. If the client decides to mirror the system, startAbSync should be invoked which would return immediately indicating the status of the request.

7. Once the filesystem partition sync opration complete, the filesystem manager would invoke the callback which indicates the response of the API.

3.13.4 Call flow for sensor self test

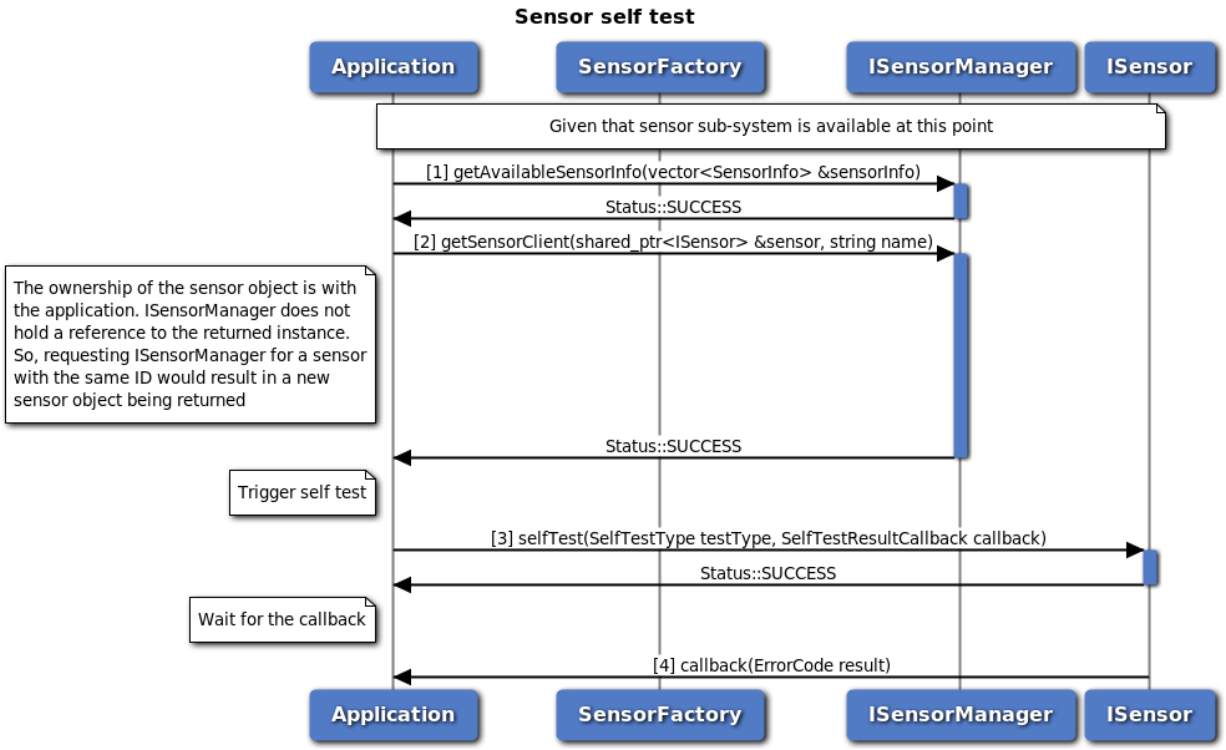

Figure 3-131 Sensor self test call flow

Certain sensors offer self test feature which can be invoked whenever needed by the application using the selfTest API. The sensor sub-system should have been initialized successfully with SERVICE_AVAILABLE as a pre-requisite for sensor data acquisition and a valid SensorManager object is available.

1. Get the information about available sensors from ISensorManager. The information will be provided in the sensorInfo parameter that would be passed by reference.
2. Given the information about different sensors, identify the required sensor (name of the sensor) using the provided attributes - type, name, vendor or required sampling frequency. Having identified the required sensor, request for the sensor object from ISensorManager with the required name of the sensor. If the request was successful, the provided reference to shared_ptr<ISensorClient> would be set by the ISensorManager which can be used to further configure and acquire data from the sensor. The ownership of the sensor object is with the application. SensorManager does not hold a reference to the returned instance. So, requesting ISensorManager for a sensor with the same name would result in a new sensor object being returned.
3. Trigger the self test with the require self test type and provide a callback that would be invoked by the sensor framework once the self test is completed.
4. Once the self test is completed, the callback gets invoked indicating the result of the self test.

3.14 Crypto

The CryptoManager class provides APIs to generate, import, export and upgrade key based on various cryptographic algorithms. Data can be signed and verified using this key. Further data can be encrypted and decrypted with the key.

3.14.1 Call flow to generate and export key

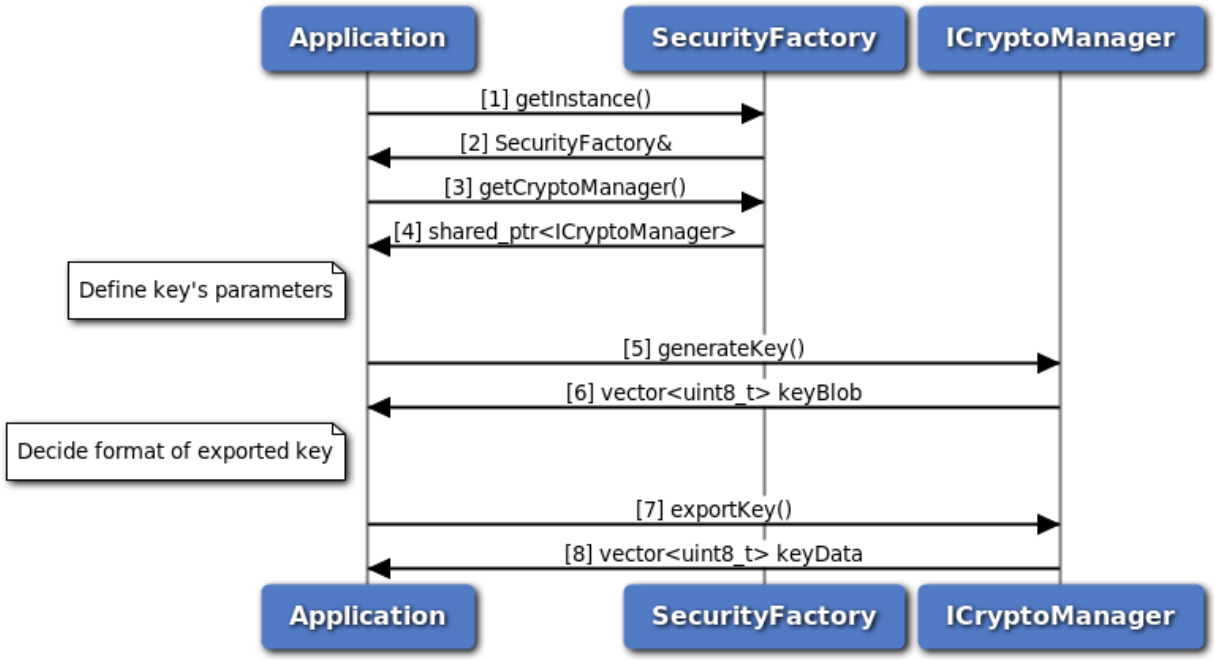

Figure 3-132 Call flow to generate and export key

1. Application request an instance of SecurityFactory.
2. An instance of SecurityFactory is received by the application.
3. From the SecurityFactory, application request an instance of ICryptoManager.
4. An instance of ICryptoManager is received by the application.
5. Application creates an instance of ICryptoParam using CryptoParamBuilder to define input parameters for the key.
6. Application calls generateKey() API of ICryptoManager. Now, the application has a key blob which represents the crypto key. Application should use this key blob for signing, verification, encryption and decryption operations.
7. For a given use-case, if the application wants to extract public key out of the key blob, it can do so by calling export key.
8. ICryptoManager returns key in the format requested for ex; X509.

3.14.2 Call flow to sign and verify data

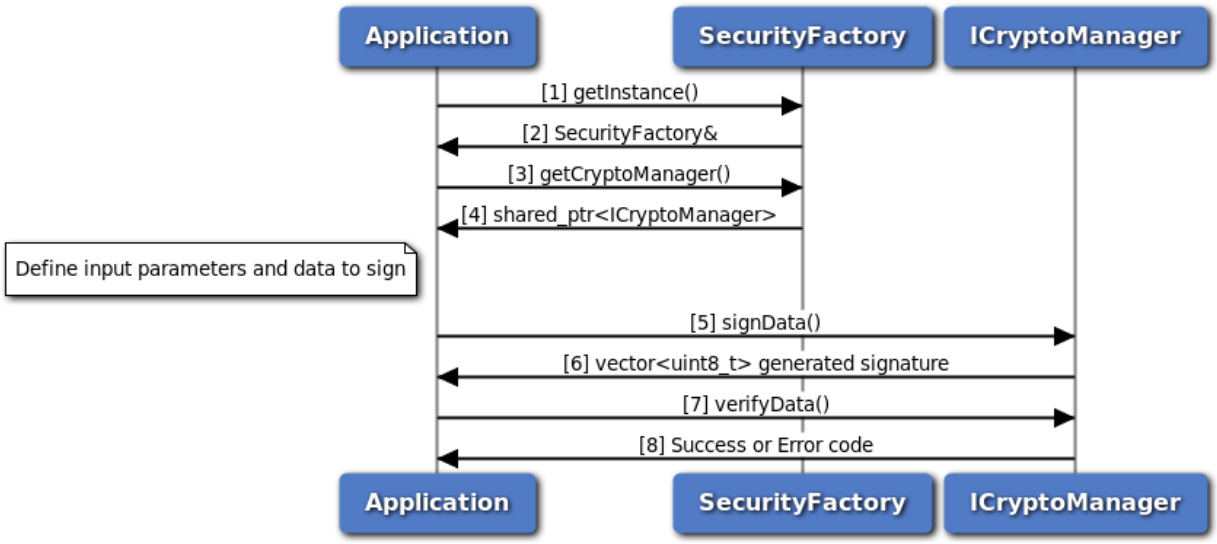

Figure 3-133 Call flow to sign and verify data

1. Application request an instance of SecurityFactory.
2. An instance of SecurityFactory is received by the application.
3. From the SecurityFactory, application request an instance of ICryptoManager.
4. Application creates an instance of ICryptoParam using CryptoParamBuilder to define input parameters for how the signing should occur.
5. Application calls signData() API of ICryptoManager passing the key blob, data to be signed and input parameters.
6. ICryptoManager returns signature corresponding to the inputs given.
7. Application creates an instance of ICryptoParam using CryptoParamBuilder to define input parameters for how the verification should occur.
8. Application calls verifyData() API of ICryptoManager passing the key blob, signature, data to be verified and input parameters.
9. ICryptoManager returns success if the verification succeeds (valid data and signature) otherwise appropriate error code.

3.14.3 Call flow to encrypt and decrypt data

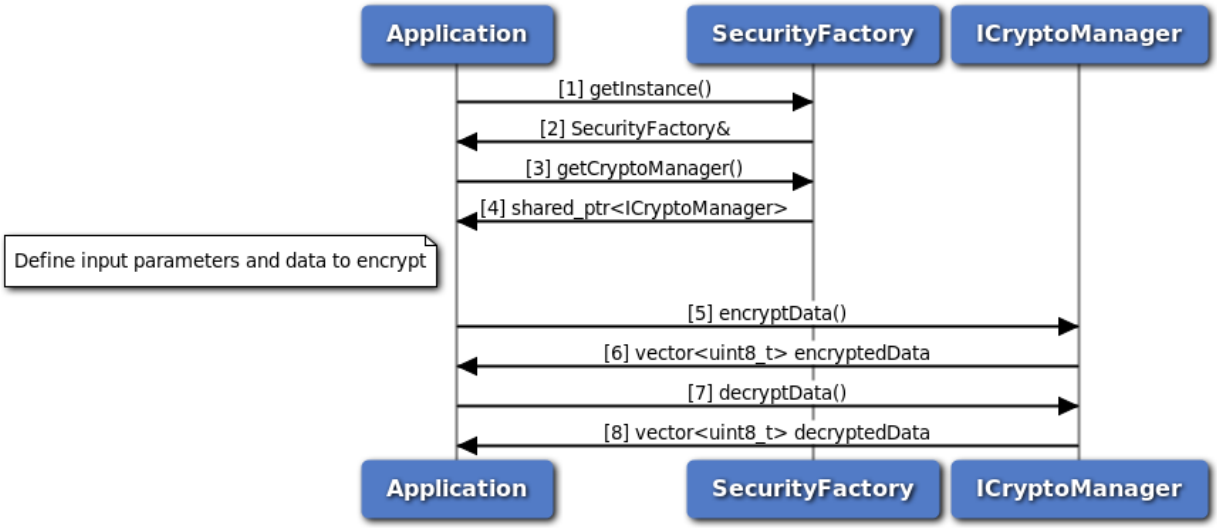

Figure 3-134 Call flow to encrypt and decrypt data

1. Application request an instance of SecurityFactory.
2. An instance of SecurityFactory is received by the application.
3. From the SecurityFactory, application request an instance of ICryptoManager.
4. Application creates an instance of ICryptoParam using CryptoParamBuilder to define input parameters for how the encryption should occur.
5. Application calls encryptData() API of ICryptoManager passing the key blob, data to be encrypted and input parameters.
6. ICryptoManager returns encrypted data corresponding to the inputs given.
7. Application creates an instance of ICryptoParam using CryptoParamBuilder to define input parameters for how the decryption should occur.
8. Application calls decryptData() API of ICryptoManager passing the key blob, encrypted data and input parameters.
9. ICryptoManager returns decrypted data otherwise appropriate error code.

3.15 Crypto accelerator

The ICryptoAcceleratorManager provides APIs to verify signature and do ECQV calculation based on elliptic-curve cryptography.

3.15.1 Call flow for signature verification synchronous mode

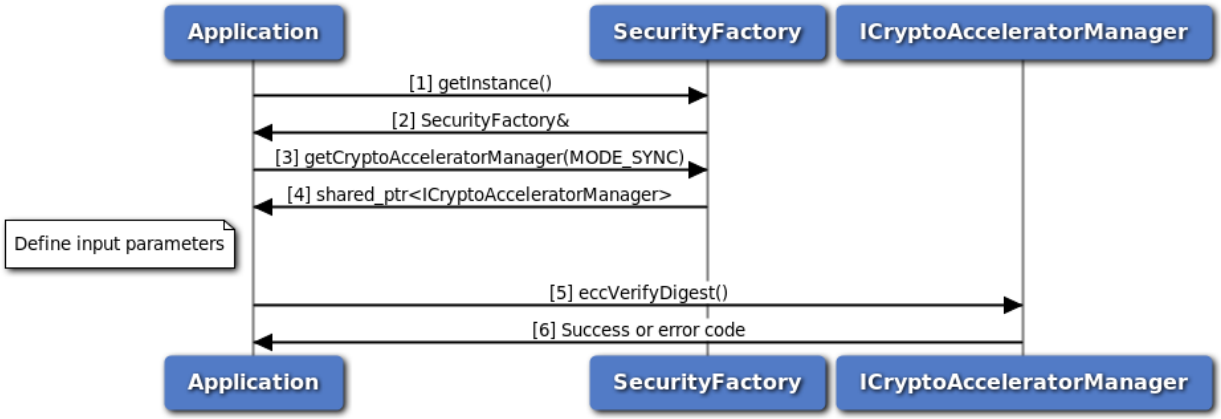

Figure 3-135 Call flow for signature verification synchronous mode

1. Application request an instance of SecurityFactory.
2. An instance of SecurityFactory is received by the application.
3. From the SecurityFactory, application request an instance of ICryptoAcceleratorManager.
4. An instance of ICryptoAcceleratorManager is received by the application.
5. Application defines input parameters for verification.
6. Application calls eccVerifyDigest() API of ICryptoAcceleratorManager.
7. ICryptoAcceleratorManager returns success if verification passed otherwise appropriate error code.

3.15.2 Call flow for signature verification asynchronous poll mode

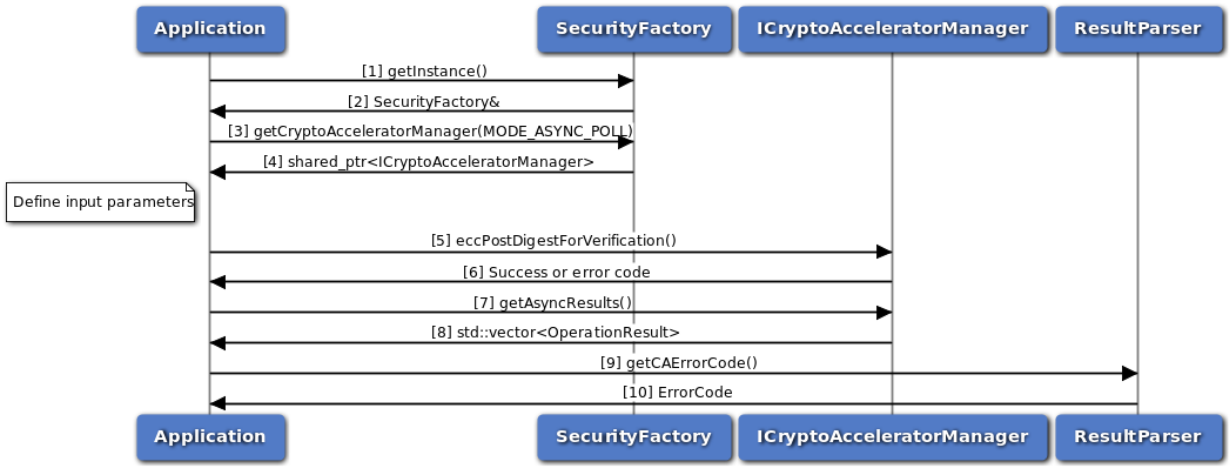

Figure 3-136 Call flow for signature verification asynchronous poll mode

1. Application request an instance of SecurityFactory.

2. An instance of SecurityFactory is received by the application.
3. From the SecurityFactory, application request an instance of ICryptoAcceleratorManager.
4. An instance of ICryptoAcceleratorManager is received by the application.
5. Application defines input parameters for verification.
6. Application calls eccPostDigestForVerification() API of ICryptoAcceleratorManager.
7. ICryptoAcceleratorManager returns success if data is sent for verification.
8. Application calls getAsyncResults() API of ICryptoAcceleratorManager to get the results.
9. ICryptoAcceleratorManager returns verification data obtained from crypto accelerator.
10. Application calls various APIs of ResultParser to get exact verification result.

3.15.3 Call flow for signature verification asynchronous listener mode

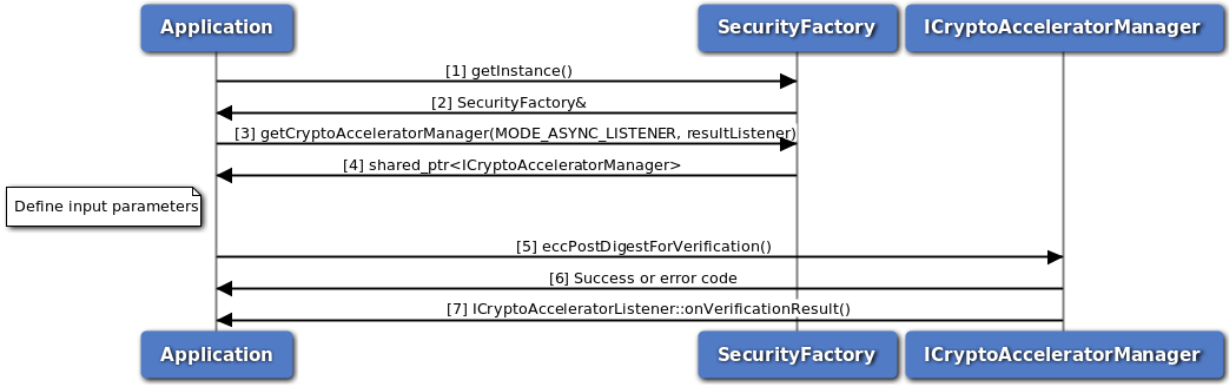

Figure 3-137 Call flow for signature verification asynchronous listener mode

1. Application request an instance of SecurityFactory.
2. An instance of SecurityFactory is received by the application.
3. From the SecurityFactory, application request an instance of ICryptoAcceleratorManager.
4. An instance of ICryptoAcceleratorManager is received by the application.
5. Application defines input parameters for verification.
6. Application calls eccPostDigestForVerification() API of ICryptoAcceleratorManager.
7. ICryptoAcceleratorManager returns success if data is sent for verification.
8. Application receives result in method onVerificationResult() of class implementing ICryptoAcceleratorListener interface.

3.15.4 Call flow for ECQV calculation synchronous mode

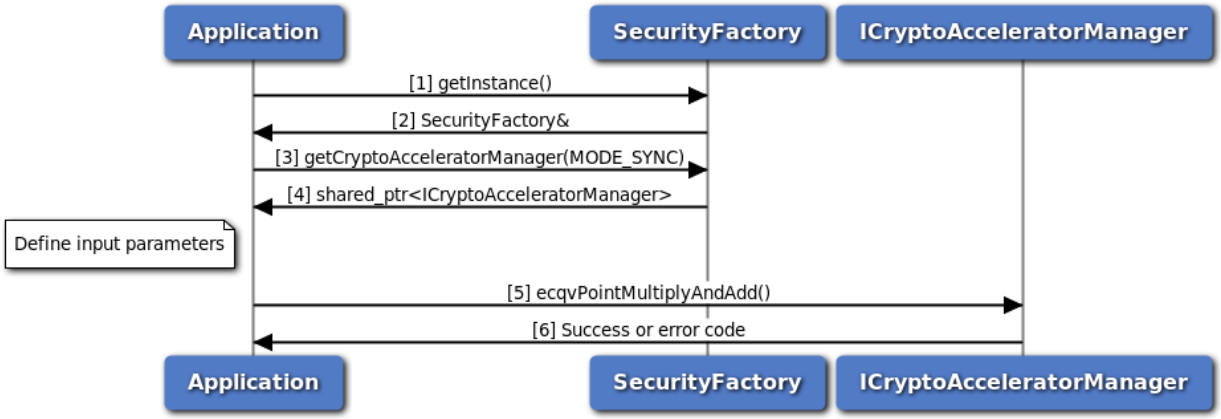

Figure 3-138 Call flow for ECQV calculation synchronous mode

1. Application request an instance of SecurityFactory.
2. An instance of SecurityFactory is received by the application.
3. From the SecurityFactory, application request an instance of ICryptoAcceleratorManager.
4. An instance of ICryptoAcceleratorManager is received by the application.
5. Application defines input parameters for calculation.
6. Application calls ecqvPointMultiplyAndAdd() API of ICryptoAcceleratorManager.
7. ICryptoAcceleratorManager returns success if calculation result otherwise appropriate error code.

3.15.5 Call flow for ECQV calculation asynchronous poll mode

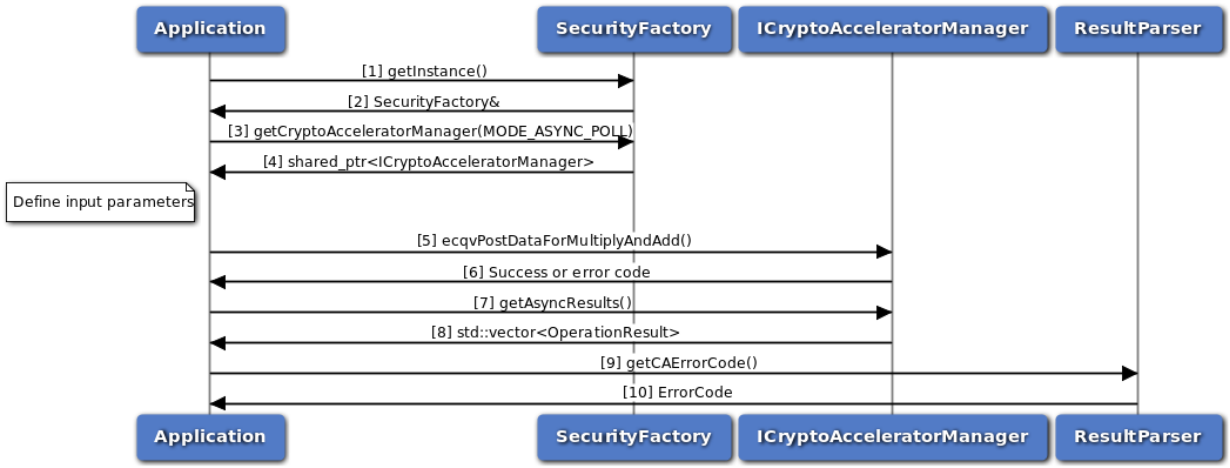

Figure 3-139 Call flow for ECQV calculation asynchronous poll mode

1. Application request an instance of SecurityFactory.

2. An instance of SecurityFactory is received by the application.
3. From the SecurityFactory, application request an instance of ICryptoAcceleratorManager.
4. An instance of ICryptoAcceleratorManager is received by the application.
5. Application defines input parameters for calculation.
6. Application calls ecqvPostDataForMultiplyAndAdd() API of ICryptoAcceleratorManager.
7. ICryptoAcceleratorManager returns success if data is sent for calculation.
8. Application calls getAsyncResults() API of ICryptoAcceleratorManager to get the results.
9. ICryptoAcceleratorManager returns calculation data obtained from crypto accelerator.
10. Application calls various APIs of ResultParser to get exact calculation result.

3.15.6 Call flow for ECQV calculation asynchronous listener mode

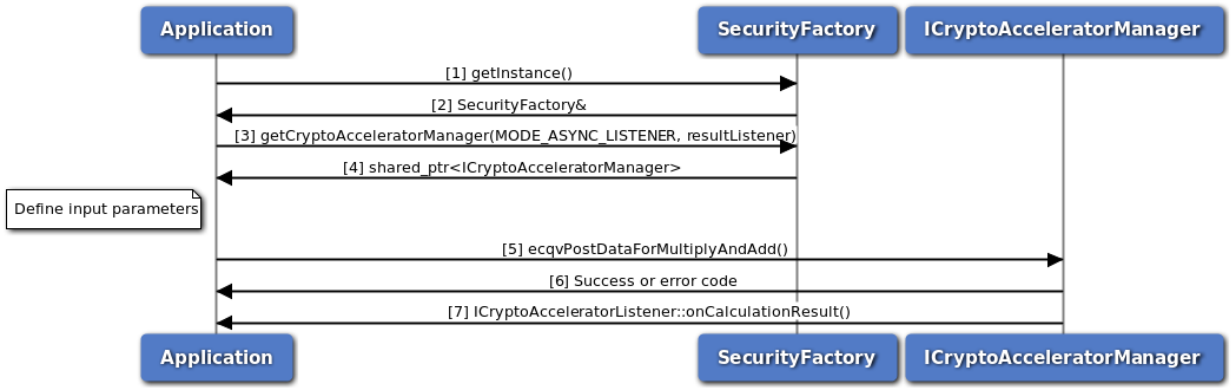

Figure 3-140 Call flow for ECQV calculation asynchronous listener mode

1. Application request an instance of SecurityFactory.
2. An instance of SecurityFactory is received by the application.
3. From the SecurityFactory, application request an instance of ICryptoAcceleratorManager.
4. An instance of ICryptoAcceleratorManager is received by the application.
5. Application defines input parameters for calculation.
6. Application calls ecqvPostDataForMultiplyAndAdd() API of ICryptoAcceleratorManager.
7. ICryptoAcceleratorManager returns success if data is sent for calculation.
8. Application receives result in method onCalculationResult() of class implementing ICryptoAcceleratorListener interface.

3.16 WLAN

3.16.1 Call flow to modify WLAN configuration

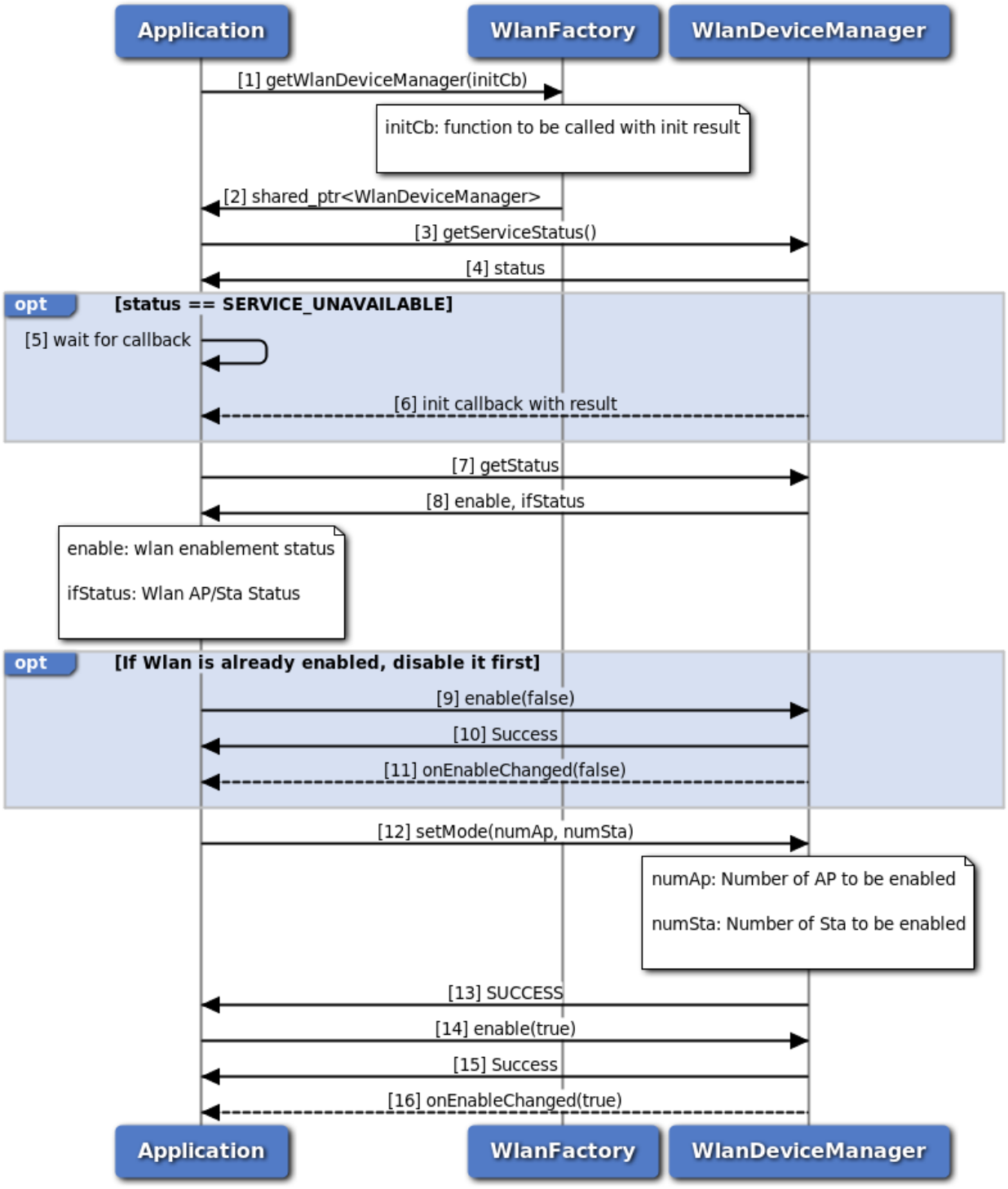

Figure 3-141 Modify WLAN Configuration Call Flow

1. Application requests IWlanDeviceManager object from WLAN factory.
2. WLAN factory returns a shared pointer to the IWlanDeviceManger object to the application.

3. Application can use IWlanDeviceManager::getServiceStatus to determine if the system is ready.
4. The application receives the status, i.e., SERVICE_AVAILABLE or SERVICE_UNAVAILABLE, to indicate if the subsystem is ready.
   (a) If the subsystem is not ready, the application can wait for the callback provided in Step 1 for subsystem initialization status.
   (b) Application provided callback is invoked with subsystem status (SERVICE_AVAILABLE/SERVICE_FAILED).
5. When the subsystem is ready, application calls IWlanDeviceManager::getStatus to get WLAN enablement status.
6. Application receives WLAN enablement status.
   (a) If WLAN is enabled, application calls IWlanDeviceManager::enable(false) to disable WLAN.
   (b) Application receives WLAN disablement response and waits for WLAN status indication.
   (c) Application receives WLAN status indication IWlanDeviceManager::onEnableChanged(false) to indicate WLAN is disabled.
7. Application sets desired configuration using IWlanDeviceManager::setMode to set desired number of access points and stations to be enabled.
8. Application receives response.
9. Application calls IWlanDeviceManager::enable(true) to enable WLAN.
10. Application receives WLAN enablement response and waits for WLAN status indication.
11. Application receives WLAN status indication IWlanDeviceManager::onEnableChanged(true) to indicate WLAN is enbaled.

3.16.2 Call flow to modify WLAN station configuration

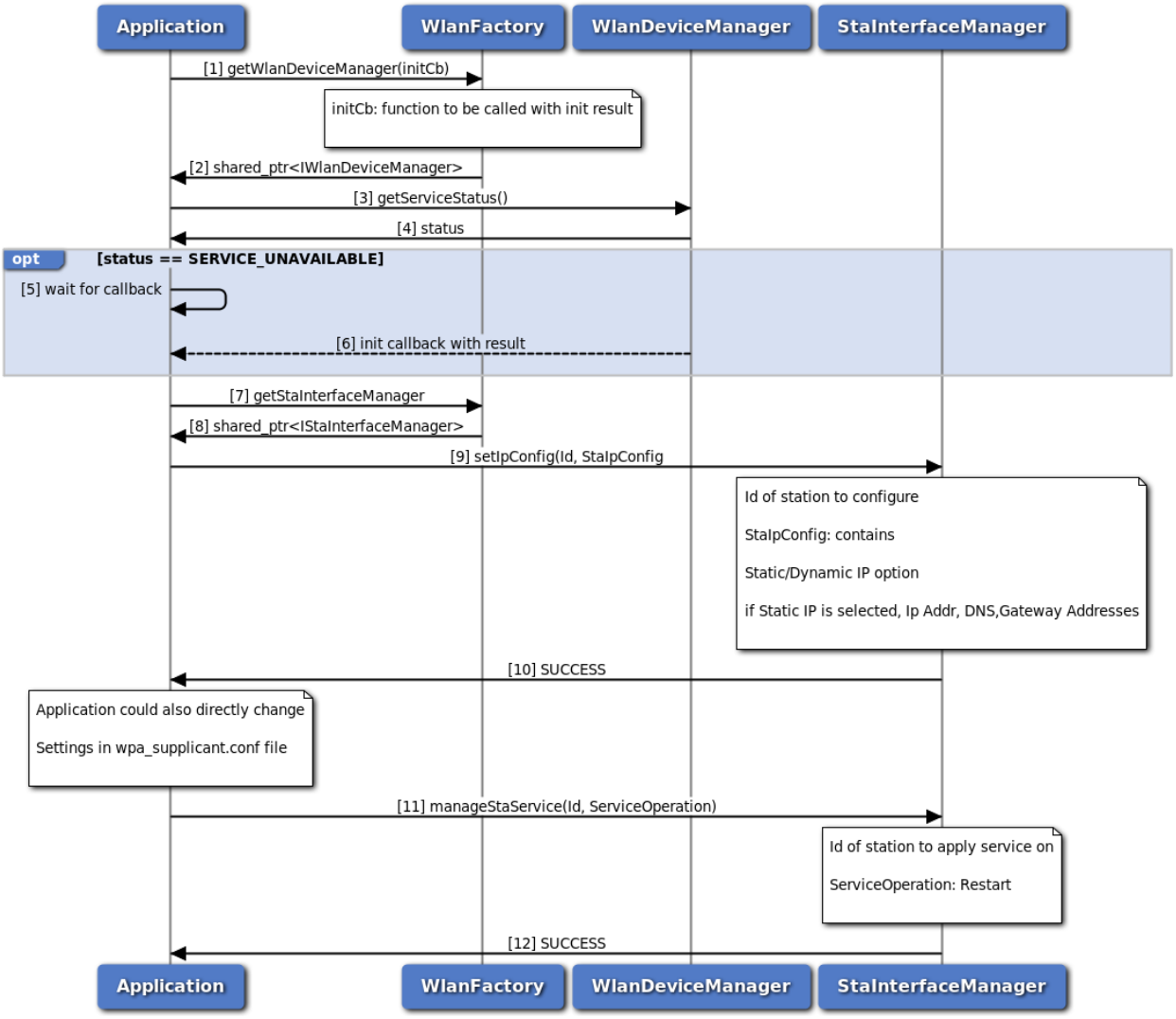

Figure 3-142 Modify WLAN Station Configuration Call Flow

1. Application requests IWlanDeviceManager object from WLAN factory.
2. WLAN factory returns shared pointer to IWlandeviceManager to the application.
3. Application can use IWlanDeviceManager::getServiceStatus to determine if the system is ready.
4. The application receives the Status i.e. either SERVICE_AVAILABLE or SERVICE_UNAVALABLE to indicate whether sub-system is ready or not.
   (a) If subsystem is not ready, then the application could wait for callback provided in step 1 for subsystem initialization status.
   (b) Application pprovided callback is invoked with subsystem status (SERVICE_AVAILABLE/SERVICE_FAILED).
5. On Readiness, requests IStaInterfaceManager object from WLAN factory.

6. WLAN factory returns shared pointer to IStaInterfaceManager to the application.
7. Application calls IStaInterfaceManager::setIpConfig to set desired IP configurations.
8. Application receives response set IP configuration.
9. Application calls IStaInterfaceManager::managerStaService to restart wpa_supplicant daemon.
10. Application receives response to restart wpa_supplicant daemon and station configuration shall be active at this stage.

3.16.3 Call flow to Modify WLAN Access Point Configuration

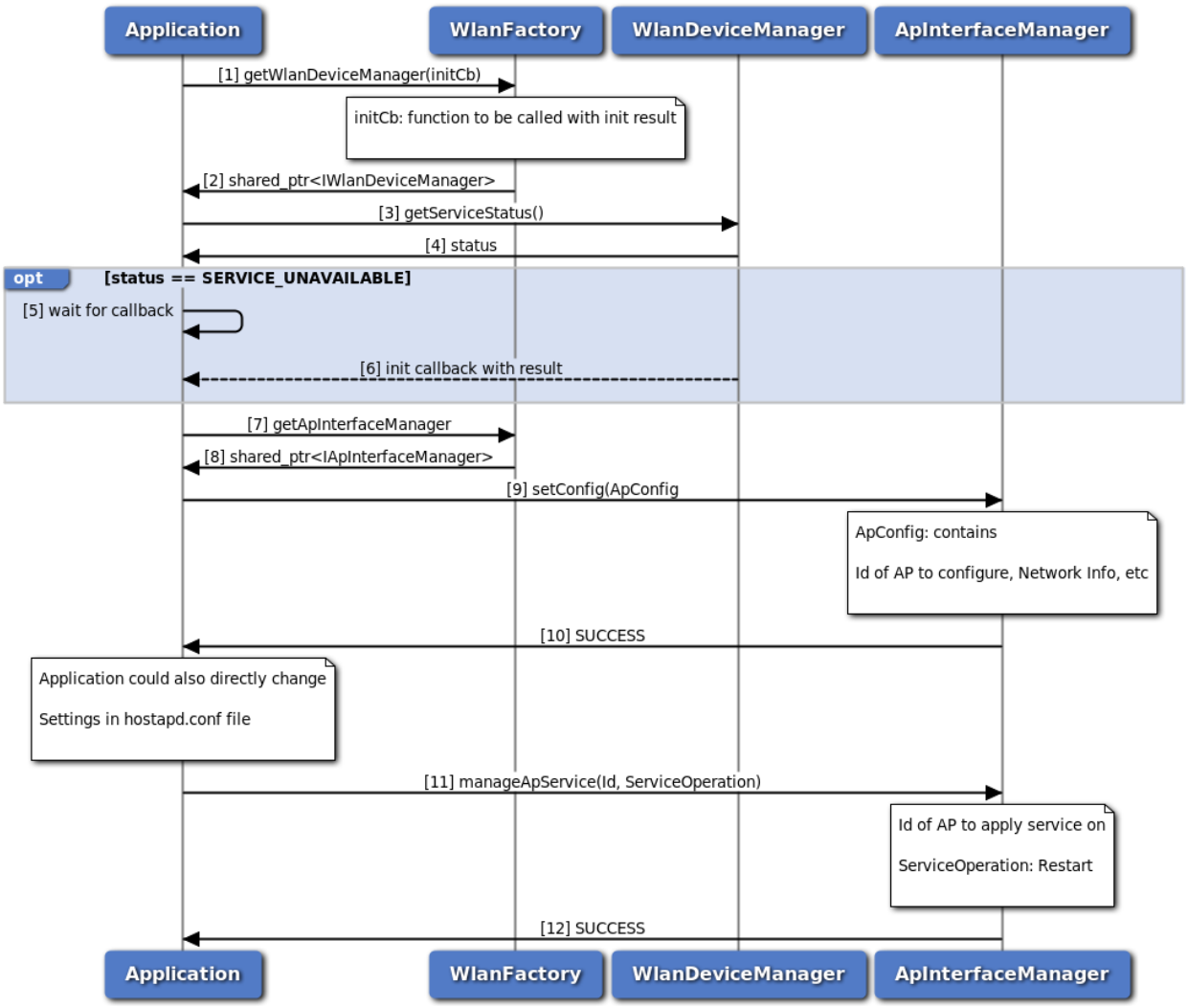

Figure 3-143 Modify WLAN Access Point Configuration Call Flow

1. Application requests IWlanDeviceManager object from WLAN factory.
2. WLAN factory returns shared pointer to IWlandeviceManager to the application.
3. Application can use IWlanDeviceManager::getServiceStatus to determine if the system is ready.

4. The application receives the Status i.e. either SERVICE_AVAILABLE or SERVICE_UNAVALABLE to indicate whether sub-system is ready or not.
   (a) If subsystem is not ready, then the application could wait for callback provided in step 1 for subsystem initialization status.
   (b) Application pprovided callback is invoked with subsystem status (SERVICE_AVAILABLE/SERVICE_FAILED).
5. On Readiness, requests IApInterfaceManager object from WLAN factory.
6. WLAN factory returns shared pointer to IApInterfaceManager to the application.
7. Application calls IApInterfaceManager::setConfig to set desired IP configurations.
8. Application receives response set configuration.
9. Application calls IApInterfaceManager::managerApService to restart hostapd daemon.
10. Application receives response to restart hostapd daemon and access point configuration shall be active at this stage.

4 Interfaces

4.1 Telematics SDK APIs

- Telephony
- Location
- Common
- Audio
- Thermal
- Power
- Modem Configuration
- Sensor
- Platform
- Wlan
- Cellular Data
- Cellular V2X

4.2 Telephony

- Phone
- Call
- SMS
- SIM Card Services
- Cell Broadcast
- IMS Settings
- Multi SIM
- Subscription Management
- Network Selection
- Serving System
- Remote SIM Provisioning
- Remote SIM

This section contains APIs related to Telephony.

4.3 Phone

This section contains APIs related to Phone, Signal Strength and interfaces to register global listeners to event notifications.

4.3.1 Data Structure Documentation

4.3.1.1 class telux::tel::GsmCellIdentity

GsmCellIdentity class provides methods to get mobile country code, mobile network code, location area code, cell identity, absolute RF channel number and base station identity code.

Public member functions

- GsmCellIdentity (std::string mcc, std::string mnc, int lac, int cid, int arfcn, int bsic)
- const int getMcc ()
- const int getMnc ()
- const std::string getMobileCountryCode ()
- const std::string getMobileNetworkCode ()
- const int getLac ()
- const int getIdentity ()
- const int getArfcn ()
- const int getBaseStationIdentityCode ()

4.3.1.1.1 Constructors and Destructors

4.3.1.1.1.1 telux::tel::GsmCellIdentity::GsmCellIdentity (std::string *mcc,* std::string *mnc,* int *lac,* int *cid,* int *arfcn,* int *bsic*)

4.3.1.1.2 Member Function Documentation

4.3.1.1.2.1 const int telux::tel::GsmCellIdentity::getMcc ( )

Get the Mobile Country Code.

Returns

Mcc value.

Deprecated

Use getMobileCountryCode() API instead

4.3.1.1.2.2 const int telux::tel::GsmCellIdentity::getMnc ( )

Get the Mobile Network Code.

Returns

Mnc value.

Deprecated

Use getMobileNetworkCode() API instead

4.3.1.1.2.3 const std::string telux::tel::GsmCellIdentity::getMobileCountryCode ( )

Get the Mobile Country Code.

Returns

Mcc value.

4.3.1.1.2.4 const std::string telux::tel::GsmCellIdentity::getMobileNetworkCode ( )

Get the Mobile Network Code.

Returns

Mnc value.

4.3.1.1.2.5 const int telux::tel::GsmCellIdentity::getLac ( )

Get the location area code.

Returns

Location area code.

4.3.1.1.2.6 const int telux::tel::GsmCellIdentity::getIdentity ( )

Get the cell identity.

Returns

Cell identity.

4.3.1.1.2.7 const int telux::tel::GsmCellIdentity::getArfcn ( )

Get the absolute RF channel number.

Returns

Absolute RF channel number.

4.3.1.1.2.8 const int telux::tel::GsmCellIdentity::getBaseStationIdentityCode ( )

Get the base station identity code.

Returns

Base station identity code.

4.3.1.2 class telux::tel::CdmaCellIdentity

CdmaCellIdentity class provides methods to get the network identifier, system identifier, base station identifier, longitude and latitude.

Deprecated

As of version 1.53.0 this API is no longer supported.

Public member functions

- CdmaCellIdentity (int networkId, int systemId, int baseStationId, int longitude, int latitude)
- const int getNid ()
- const int getSid ()
- const int getBaseStationId ()
- const int getLongitude ()
- const int getLatitude ()

4.3.1.2.1 Constructors and Destructors

4.3.1.2.1.1 telux::tel::CdmaCellIdentity::CdmaCellIdentity (int *networkId,* int *systemId,* int *baseStationId,* int *longitude,* int *latitude*)

4.3.1.2.2 Member Function Documentation

4.3.1.2.2.1 const int telux::tel::CdmaCellIdentity::getNid ( )

Get the network identifier.

Returns

Network identifier.

4.3.1.2.2.2 const int telux::tel::CdmaCellIdentity::getSid ( )

Get the system identifier.

Returns

System identifier.

4.3.1.2.2.3 const int telux::tel::CdmaCellIdentity::getBaseStationId ( )

Get the base station identifier.

Returns

Base station identifier.

4.3.1.2.2.4 const int telux::tel::CdmaCellIdentity::getLongitude ( )

Get the longitude.

Returns

Longitude.

4.3.1.2.2.5 const int telux::tel::CdmaCellIdentity::getLatitude ( )

Get the latitude.

Returns

Latitude.

4.3.1.3 class telux::tel::LteCellIdentity

LteCellIdentity class provides methods to get the mobile country code, mobile network code, cell identity, physical cell identifier, tracking area code and absolute Rf channel number.

Public member functions

- LteCellIdentity (std::string mcc, std::string mnc, int ci, int pci, int tac, int earfcn)
- const int getMcc ()
- const int getMnc ()
- const std::string getMobileCountryCode ()
- const std::string getMobileNetworkCode ()
- const int getIdentity ()
- const int getPhysicalCellId ()
- const int getTrackingAreaCode ()
- const int getEarfcn ()

4.3.1.3.1 Constructors and Destructors

4.3.1.3.1.1 telux::tel::LteCellIdentity::LteCellIdentity (std::string *mcc,* std::string *mnc,* int *ci,* int *pci,* int *tac,* int *earfcn*)

4.3.1.3.2 Member Function Documentation

4.3.1.3.2.1 const int telux::tel::LteCellIdentity::getMcc ( )

Get the Mobile Country Code.

Returns

Mcc value.

Deprecated

Use getMobileCountryCode() API instead

4.3.1.3.2.2 const int telux::tel::LteCellIdentity::getMnc ( )

Get the Mobile Network Code.

Returns

Mnc value.

Deprecated

Use getMobileNetworkCode() API instead

4.3.1.3.2.3 const std::string telux::tel::LteCellIdentity::getMobileCountryCode ( )

Get the Mobile Country Code.

Returns

Mcc value.

4.3.1.3.2.4 const std::string telux::tel::LteCellIdentity::getMobileNetworkCode ( )

Get the Mobile Network Code.

Returns

Mnc value.

4.3.1.3.2.5 const int telux::tel::LteCellIdentity::getIdentity ( )

Get the cell identity.

Returns

Cell identity.

4.3.1.3.2.6 const int telux::tel::LteCellIdentity::getPhysicalCellId ( )

Get the physical cell identifier.

Returns

Physical cell identifier.

4.3.1.3.2.7 const int telux::tel::LteCellIdentity::getTrackingAreaCode ( )

Get the tracking area code.

Returns

Tracking area code.

4.3.1.3.2.8 const int telux::tel::LteCellIdentity::getEarfcn ( )

Get the absolute RF channel number.

Returns

Absolute RF channel number.

4.3.1.4 class telux::tel::WcdmaCellIdentity

WcdmaCellIdentity class provides methods to get the mobile country code, mobile network code, location area code, cell identitifier, primary scrambling code and absolute RF channel number.

Public member functions

- WcdmaCellIdentity (std::string mcc, std::string mnc, int lac, int cid, int psc, int uarfcn)
- const int getMcc ()
- const int getMnc ()
- const std::string getMobileCountryCode ()
- const std::string getMobileNetworkCode ()
- const int getLac ()
- const int getIdentity ()
- const int getPrimaryScramblingCode ()
- const int getUarfcn ()

4.3.1.4.1 Constructors and Destructors

4.3.1.4.1.1 telux::tel::WcdmaCellIdentity::WcdmaCellIdentity (std::string *mcc,* std::string *mnc,* int *lac,* int *cid,* int *psc,* int *uarfcn*)

4.3.1.4.2 Member Function Documentation

4.3.1.4.2.1 const int telux::tel::WcdmaCellIdentity::getMcc ( )

Get the Mobile Country Code.

Returns

Mcc value.

Deprecated

Use getMobileCountryCode() API instead

4.3.1.4.2.2 const int telux::tel::WcdmaCellIdentity::getMnc ( )

Get the Mobile Network Code.

Returns

Mnc value.

Deprecated

Use getMobileNetworkCode() API instead

4.3.1.4.2.3 const std::string telux::tel::WcdmaCellIdentity::getMobileCountryCode ( )

Get the Mobile Country Code.

Returns

Mcc value.

4.3.1.4.2.4 const std::string telux::tel::WcdmaCellIdentity::getMobileNetworkCode ( )

Get the Mobile Network Code.

Returns

Mnc value.

4.3.1.4.2.5 const int telux::tel::WcdmaCellIdentity::getLac ( )

Get the location area code.

Returns

Location area code.

4.3.1.4.2.6 const int telux::tel::WcdmaCellIdentity::getIdentity ( )

Get the cell identity.

Returns

Cell identity.

4.3.1.4.2.7 const int telux::tel::WcdmaCellIdentity::getPrimaryScramblingCode ( )

Get the primary scrambling code.

Returns

Primary scrambling code.

4.3.1.4.2.8 const int telux::tel::WcdmaCellIdentity::getUarfcn ( )

Get the absolute RF channel number.

Returns

Absolute RF channel number.

4.3.1.5 class telux::tel::TdscdmaCellIdentity

TdscdmaCellIdentity class provides methods to get the mobile country code, mobile network code, location area code, cell identity and cell parameters identifier.

Deprecated

As of version 1.53.0 this API is no longer supported.

Public member functions

- TdscdmaCellIdentity (std::string mcc, std::string mnc, int lac, int cid, int cpid)
- const int getMcc ()
- const int getMnc ()
- const std::string getMobileCountryCode ()
- const std::string getMobileNetworkCode ()
- const int getLac ()
- const int getIdentity ()
- const int getParametersId ()

4.3.1.5.1 Constructors and Destructors

4.3.1.5.1.1 telux::tel::TdscdmaCellIdentity::TdscdmaCellIdentity (std::string *mcc,* std::string *mnc,* int *lac,* int *cid,* int *cpid*)

4.3.1.5.2 Member Function Documentation

4.3.1.5.2.1 const int telux::tel::TdscdmaCellIdentity::getMcc ( )

Get the Mobile Country Code.

Returns

Mcc value.

Deprecated

Use getMobileCountryCode() API instead

4.3.1.5.2.2 const int telux::tel::TdscdmaCellIdentity::getMnc ( )

Get the Mobile Network Code.

Returns

Mnc value.

Deprecated

Use getMobileNetworkCode() API instead

4.3.1.5.2.3 const std::string telux::tel::TdscdmaCellIdentity::getMobileCountryCode ( )

Get the Mobile Country Code.

Returns

Mcc value.

4.3.1.5.2.4 const std::string telux::tel::TdscdmaCellIdentity::getMobileNetworkCode ( )

Get the Mobile Network Code.

Returns

Mnc value.

4.3.1.5.2.5 const int telux::tel::TdscdmaCellIdentity::getLac ( )

Get the location area code

Returns

Location area code.

4.3.1.5.2.6 const int telux::tel::TdscdmaCellIdentity::getIdentity ( )

Get the cell identity.

Returns

Cell identity.

4.3.1.5.2.7 const int telux::tel::TdscdmaCellIdentity::getParametersId ( )

Get the cell parameters identifier.

Returns

Cell parameters identifier.

4.3.1.6 class telux::tel::Nr5gCellIdentity

Nr5gCellIdentity class provides methods to get the mobile country code, mobile network code, cell identity, physical cell identifier, tracking area code and absolute RF channel number information of the Serving cell

Public member functions

- Nr5gCellIdentity (std::string mcc, std::string mnc, uint64_t ci, uint32_t pci, int32_t tac, int32_t arfcn)
- const std::string getMobileCountryCode ()
- const std::string getMobileNetworkCode ()
- const uint64_t getIdentity ()
- const uint32_t getPhysicalCellId ()
- const int32_t getTrackingAreaCode ()
- const int32_t getArfcn ()

4.3.1.6.1 Constructors and Destructors

4.3.1.6.1.1 telux::tel::Nr5gCellIdentity::Nr5gCellIdentity (std::string *mcc,* std::string *mnc,* uint64_t *ci,* uint32_t *pci,* int32_t *tac,* int32_t *arfcn*)

4.3.1.6.2 Member Function Documentation

4.3.1.6.2.1 const std::string telux::tel::Nr5gCellIdentity::getMobileCountryCode ( )

Get the Mobile Country Code.

Returns

Mcc value.

4.3.1.6.2.2 const std::string telux::tel::Nr5gCellIdentity::getMobileNetworkCode ( )

Get the Mobile Network Code.

Returns

Mnc value.

4.3.1.6.2.3 const uint64_t telux::tel::Nr5gCellIdentity::getIdentity ( )

Get the cell identity.

Returns

Cell identity.

4.3.1.6.2.4 const uint32_t telux::tel::Nr5gCellIdentity::getPhysicalCellId ( )

Get the physical cell identifier.

Returns

Physical cell identifier.

4.3.1.6.2.5 const int32_t telux::tel::Nr5gCellIdentity::getTrackingAreaCode ( )

Get the tracking area code.

Returns

Tracking area code.

4.3.1.6.2.6 const int32_t telux::tel::Nr5gCellIdentity::getArfcn ( )

Get the absolute RF channel number.

Returns

Absolute RF channel number. '-1' denotes that the value is unknown.

4.3.1.7 class telux::tel::CellInfo

CellInfo class provides cell info type and checks whether the current cell is registered or not.

Public member functions

- virtual CellType getType ()
- virtual bool isRegistered ()

Protected Attributes

- CellType type_
- int registered_

4.3.1.7.1 Member Function Documentation

4.3.1.7.1.1 virtual CellType telux::tel::CellInfo::getType ( ) [virtual]

Get the cell type.

Returns

CellType.

4.3.1.7.1.2 virtual bool telux::tel::CellInfo::isRegistered ( ) [virtual]

Checks whether the current cell is registered or not.

Returns

If true cell is registered or vice-versa.

4.3.1.7.2 Field Documentation

4.3.1.7.2.1 CellType telux::tel::CellInfo::type_ [protected]

4.3.1.7.2.2 int telux::tel::CellInfo::registered_ [protected]

4.3.1.8 class telux::tel::GsmCellInfo

GsmCellInfo class provides methods to get cell type, cell registration status, cell identity and signal strength information.

Public member functions

- GsmCellInfo (int registered, GsmCellIdentity id, GsmSignalStrengthInfo ssInfo)
- GsmCellIdentity getCellIdentity ()
- GsmSignalStrengthInfo getSignalStrengthInfo ()

Additional Inherited Members

4.3.1.8.1 Constructors and Destructors

4.3.1.8.1.1 telux::tel::GsmCellInfo::GsmCellInfo (int *registered,* GsmCellIdentity *id,* GsmSignal↩ StrengthInfo *ssInfo*)

GsmCellInfo constructor.

Parameters

in	*registered*	- Registration status of the cell.
in	*id*	- GSM cell identity.
in	*ssInfo*	- GSM cell signal strength.

4.3.1.8.2 Member Function Documentation

4.3.1.8.2.1 GsmCellIdentity telux::tel::GsmCellInfo::getCellIdentity ( )

Get GSM cell identity information.

Returns

GsmCellIdentity.

4.3.1.8.2.2 GsmSignalStrengthInfo telux::tel::GsmCellInfo::getSignalStrengthInfo ( )

Get GSM cell signal strength information.

Returns

GsmSignalStrengthInfo.

4.3.1.9 class telux::tel::CdmaCellInfo

CdmaCellInfo class provides methods to get cell type, cell registration status, cell identity and signal strength information.

Deprecated

As of version 1.53.0 this API is no longer supported.

Public member functions

- CdmaCellInfo (int registered, CdmaCellIdentity id, CdmaSignalStrengthInfo ssInfo)
- CdmaCellIdentity getCellIdentity ()
- CdmaSignalStrengthInfo getSignalStrengthInfo ()

Additional Inherited Members

4.3.1.9.1 Constructors and Destructors

4.3.1.9.1.1 telux::tel::CdmaCellInfo::CdmaCellInfo (int *registered,* CdmaCellIdentity *id,* CdmaSignal↩ StrengthInfo *ssInfo*)

CdmaCellInfo constructor

Parameters

in	*registered*	- Registration status of the cell.
in	*id*	- CDMA cell identity.
in	*ssInfo*	- CDMA cell signal strength.

4.3.1.9.2 Member Function Documentation

4.3.1.9.2.1 CdmaCellIdentity telux::tel::CdmaCellInfo::getCellIdentity ( )

Get CDMA cell identity information.

Returns

CdmaCellIdentity.

4.3.1.9.2.2 CdmaSignalStrengthInfo telux::tel::CdmaCellInfo::getSignalStrengthInfo ( )

Get CDMA cell signal strength information.

Returns

CdmaSignalStrengthInfo.

4.3.1.10 class telux::tel::LteCellInfo

LteCellInfo class provides methods to get cell type, cell registration status, cell identity and signal strength information.

Public member functions

- LteCellInfo (int registered, LteCellIdentity id, LteSignalStrengthInfo ssInfo)
- LteCellIdentity getCellIdentity ()
- LteSignalStrengthInfo getSignalStrengthInfo ()

Additional Inherited Members

4.3.1.10.1 Constructors and Destructors

4.3.1.10.1.1 telux::tel::LteCellInfo::LteCellInfo (int *registered,* LteCellIdentity *id,* LteSignalStrengthInfo *ssInfo*)

LteCellInfo constructor.

Parameters

in	*registered*	- Registration status of the cell.
in	*id*	- LTE cell identity class.
in	*ssInfo*	- LTE cell signal strength.

4.3.1.10.2 Member Function Documentation

4.3.1.10.2.1 LteCellIdentity telux::tel::LteCellInfo::getCellIdentity ( )

Get LTE cell identity information.

Returns

LteCellIdentity.

4.3.1.10.2.2 LteSignalStrengthInfo telux::tel::LteCellInfo::getSignalStrengthInfo ( )

Get LTE cell signal strength information.

Returns

LteSignalStrengthInfo.

4.3.1.11 class telux::tel::WcdmaCellInfo

WcdmaCellInfo class provides methods to get cell type, cell registration status, cell identity and signal strength information.

Public member functions

- WcdmaCellInfo (int registered, WcdmaCellIdentity id, WcdmaSignalStrengthInfo ssInfo)
- WcdmaCellIdentity getCellIdentity ()
- WcdmaSignalStrengthInfo getSignalStrengthInfo ()

Additional Inherited Members

4.3.1.11.1 Constructors and Destructors

4.3.1.11.1.1 telux::tel::WcdmaCellInfo::WcdmaCellInfo (int *registered,* WcdmaCellIdentity *id,* WcdmaSignalStrengthInfo *ssInfo*)

WcdmaCellInfo constructor.

Parameters

in	*registered*	- Registration status of the cell.
in	*id*	- WCDMA cell identity.

in	*ssInfo*	- WCDMA cell signal strength.

4.3.1.11.2 Member Function Documentation

4.3.1.11.2.1 WcdmaCellIdentity telux::tel::WcdmaCellInfo::getCellIdentity ( )

Get WCDMA cell identity information.

Returns

WcdmaCellIdentity.

4.3.1.11.2.2 WcdmaSignalStrengthInfo telux::tel::WcdmaCellInfo::getSignalStrengthInfo ( )

Get WCDMA cell signal strength information.

Returns

WcdmaSignalStrengthInfo.

4.3.1.12 class telux::tel::TdscdmaCellInfo

TdscdmaCellInfo class provides methods to get cell type, cell registration status, cell identity and signal strength information.

Deprecated

As of version 1.53.0 this API is no longer supported.

Public member functions

- TdscdmaCellInfo (int registered, TdscdmaCellIdentity id, TdscdmaSignalStrengthInfo ssInfo)
- TdscdmaCellIdentity getCellIdentity ()
- TdscdmaSignalStrengthInfo getSignalStrengthInfo ()

Additional Inherited Members

4.3.1.12.1 Constructors and Destructors

4.3.1.12.1.1 telux::tel::TdscdmaCellInfo::TdscdmaCellInfo (int *registered,* TdscdmaCellIdentity *id,* TdscdmaSignalStrengthInfo *ssInfo*)

TdscdmaCellInfo constructor.

Parameters

in	*registered*	- Registration status of the cell
in	*id*	- TDSCDMA cell identity.
in	*ssInfo*	- TDSCDMA cell signal strength.

4.3.1.12.2 Member Function Documentation

4.3.1.12.2.1 TdscdmaCellIdentity telux::tel::TdscdmaCellInfo::getCellIdentity ( )

Get TDSCDMA cell identity information.

Returns

TdscdmaCellIdentity.

4.3.1.12.2.2 TdscdmaSignalStrengthInfo telux::tel::TdscdmaCellInfo::getSignalStrengthInfo ( )

Get TDSCDMA cell signal strength information.

Returns

TdscdmaSignalStrengthInfo.

4.3.1.13 class telux::tel::Nr5gCellInfo

Nr5gCellInfo class provides methods to get cell type, cell registration status, cell identity and signal strength information corresponding to the Serving cell.

Public member functions

- Nr5gCellInfo (int registered, Nr5gCellIdentity id, Nr5gSignalStrengthInfo ssInfo)
- Nr5gCellIdentity getCellIdentity ()
- Nr5gSignalStrengthInfo getSignalStrengthInfo ()

Additional Inherited Members

4.3.1.13.1 Constructors and Destructors

4.3.1.13.1.1 telux::tel::Nr5gCellInfo::Nr5gCellInfo (int *registered,* Nr5gCellIdentity *id,* Nr5gSignal↩ StrengthInfo *ssInfo*)

4.3.1.13.2 Member Function Documentation

4.3.1.13.2.1 Nr5gCellIdentity telux::tel::Nr5gCellInfo::getCellIdentity ( )

Get NR5G cell identity information.

Returns

Nr5gCellIdentity.

4.3.1.13.2.2 Nr5gSignalStrengthInfo telux::tel::Nr5gCellInfo::getSignalStrengthInfo ( )

Get NR5G cell signal strength information.

Returns

Nr5gSignalStrengthInfo.

4.3.1.14 struct telux::tel::ECallMsdOptionals

Represents the availability of some optional parameters in MSD as per European eCall MSD standard EN 15722.

Data fields

Type	Field	Description
ECall↩ OptionalData↩ Type	optionalData↩ Type	Type of optional data
bool	optionalData↩ Present	Availability of Optional data: true - Present or false - Absent
bool	recentVehicle↩ LocationN1↩ Present	Availability of Recent Vehicle Location N1 data: true - Present or false - Absent. In MSD version-3 (as per EN 15722:2020), as recentVehicleLocationN1 is mandatory, this should be set to true by client
bool	recentVehicle↩ LocationN2↩ Present	Availability of Recent Vehicle Location N2 data: true - Present or false - Absent. In MSD version-3 (as per EN 15722:2020), as recentVehicleLocationN2 is mandatory, this should be set to true by client
bool	numberOf↩ Passengers↩ Present	Availability of number of seat belts fastened data: true - Present or false - Absent

Type	Field	Description

4.3.1.15 struct telux::tel::ECallMsdControlBits

Represents ECallMsdControlBits structure as per European eCall MSD standard. i.e. EN 15722.

Data fields

Type	Field	Description
bool	automatic↩ Activation	auto / manual activation
bool	testCall	test / emergency call
bool	positionCan↩ BeTrusted	false if coincidence < 95% of reported pos within +/- 150m
ECallVehicle↩ Type	vehicleType: 5	Represents a vehicle class as per EN 15722

4.3.1.16 struct telux::tel::ECallVehicleIdentificationNumber

Represents VehicleIdentificationNumber structure as per European eCall MSD standard. i.e. EN 15722. Vehicle Identification Number confirming ISO3779.

Data fields

Type	Field	Description
string	isowmi	World Manufacturer Index (WMI)
string	isovds	Vehicle Type Descriptor (VDS)
string	isovis↩ Modelyear	Model year from Vehicle Identifier Section (VIS)
string	isovisSeqPlant	Plant code + sequential number from VIS

4.3.1.17 struct telux::tel::ECallVehiclePropulsionStorageType

Represents VehiclePropulsionStorageType structure as per European eCall MSD standard. i.e. EN 15722. Vehicle Propulsion type (energy storage): True- Present, False - Absent

Data fields

Type	Field	Description
bool	gasolineTank↩ Present	Represents the presence of Gasoline Tank in the vehicle.
bool	dieselTank↩ Present	Represents the presence of Diesel Tank in the vehicle
bool	compressed↩ NaturalGas	Represents the presence of CNG in the vehicle
bool	liquid↩ PropaneGas	Represents the presence of Liquid Propane Gas in the vehicle
bool	electric↩ EnergyStorage	Represents the presence of Electronic Storage in the vehicle

Type	Field	Description
bool	hydrogen↩ Storage	Represents the presence of Hydrogen Storage in the vehicle
bool	otherStorage	Represents the presence of Other types of storage in the vehicle

4.3.1.18 struct telux::tel::ECallVehicleLocation

Represents VehicleLocation structure as per European eCall MSD standard. i.e. EN 15722.

Data fields

Type	Field	Description
int32_t	position↩ Latitude	latitude in milliarcsec, range is (-2147483648 to 2147483647)
int32_t	position↩ Longitude	longitude in milliarcsec, range is (-2147483648 to 2147483647)

4.3.1.19 struct telux::tel::ECallVehicleLocationDelta

Represents VehicleLocationDelta structure as per European eCall MSD standard. i.e. EN 15722. Delta with respect to Current Vehicle location.

Data fields

Type	Field	Description
int16_t	latitudeDelta	(1 Unit = 100 milliarcseconds, range: -512 to 511)
int16_t	longitudeDelta	(1 Unit = 100 milliarcseconds, range: -512 to 511)

4.3.1.20 struct telux::tel::ECallOptionalPdu

Optional information for the emergency rescue service.

Data fields

Type	Field	Description
ECallDefault↩ Options	eCallDefault↩ Options	Optional information

4.3.1.21 struct telux::tel::ECallMsdData

Data structure to hold all details required to construct an MSD. Supports MSD version-2(as per EN 15722:2015) and MSD version-3(as per EN 15722:2020)

Data fields

Type	Field	Description
ECallMsd↩ Optionals	optionals	Indicates presence of optionals in ECall MSD

Type	Field	Description
uint8_t	message↩ Identifier	Starts with 1 for each new eCall and to be incremented with every retransmission
ECallMsd↩ ControlBits	control	ECallMsdControlBits structure as per European standard i.e. EN 15722
ECallVehicle↩ Identification↩ Number	vehicle↩ Identification↩ Number	VIN (vehicle identification number) according to ISO3779
ECallVehicle↩ Propulsion↩ StorageType	vehicle↩ Propulsion↩ Storage	VehiclePropulsionStorageType structure as per European standard i.e. EN 15722
uint32_t	timestamp	Seconds elapsed since midnight 01.01.1970 UTC
ECallVehicle↩ Location	vehicleLocation	VehicleLocation structure as per European standard. i.e. EN 15722
uint8_t	vehicle↩ Direction	Direction of travel in 2 degrees steps from magnetic north
ECallVehicle↩ LocationDelta	recentVehicle↩ LocationN1	Change in latitude and longitude compared to the last MSD transmission. Optional field for MSD version-2
ECallVehicle↩ LocationDelta	recentVehicle↩ LocationN2	Change in latitude and longitude compared to the last but one MSD transmission. Optional field for MSD version-2
uint8_t	numberOf↩ Passengers	Number of occupants in the vehicle. Optional field for MSD version-2 and version-3
ECall↩ OptionalPdu	optionalPdu	Optional information for the emergency rescue service (103 bytes, ASN.1 encoded); may also point to an address, where this information is locatedOptional information for the emergency rescue service
uint8_t	msdVersion	MSD format version that is being used

4.3.1.22 struct telux::tel::ECallModeInfo

Represents eCall operating mode information

Data fields

Type	Field	Description
ECallMode	mode	Represents eCall operating mode
ECallMode↩ Reason	reason	Represents eCall operating mode change reason

4.3.1.23 struct telux::tel::ECallHlapTimerStatus

Represents status of various eCall High Level Application Protocol(HLAP) timers that are maintained by UE state machine. This does not retrieve status of timers maintained by the PSAP. The timers are represented according to EN 16062:2015 standard.

Data fields

Type	Field	Description
HlapTimer↩ Status	t2	T2 Timer status
HlapTimer↩ Status	t5	T5 Timer status
HlapTimer↩ Status	t6	T6 Timer status
HlapTimer↩ Status	t7	T7 Timer status
HlapTimer↩ Status	t9	T9 Timer status
HlapTimer↩ Status	t10	T10 Timer status

4.3.1.24 struct telux::tel::ECallHlapTimerEvents

Represents events that changes the status of various eCall High Level Application Protocol(HLAP) timers that are maintained by UE state machine. This does not retrieve events of timers maintained by the PSAP. The timers are represented according to EN 16062:2015 standard.

Data fields

Type	Field	Description
HlapTimer↩ Event	t2	T2 Timer event
HlapTimer↩ Event	t5	T5 Timer event
HlapTimer↩ Event	t6	T6 Timer event
HlapTimer↩ Event	t7	T7 Timer event
HlapTimer↩ Event	t9	T9 Timer event
HlapTimer↩ Event	t10	T10 Timer event

Type	Field	Description

4.3.1.25 struct telux::tel::CustomSipHeader

Represents custom SIP headers for content type and accept info for a PSAP. This provides clients the ability to transfer custom SIP headers with the SIP INVITE that is sent as part of call connect on TPS eCall over IMS. The value corresponding to these data fields should be recognised by a PSAP otherwise no acknowledgement would be received by device.

Data fields

Type	Field	Description
string	contentType	Type of data being transmitted and should be filled as per RFC 8147 i.e MSD. Max Length 128 bytes
string	acceptInfo	SIP Accept header. Max length 128 bytes

4.3.1.26 struct telux::tel::EcallConfig

Represents various configuration parameters related to automotive emergency call

Data fields

Type	Field	Description
EcallConfig↩ Validity	config↩ ValidityMask	Indicates the valid configuration parameters in the structure. A bit set to 1 denotes that the corresponding configuration parameter is valid
bool	muteRxAudio	
ECallNumType	numType	
string	overriddenNum	
bool	useCannedMsd	
uint32_t	gnssUpdate↩ Interval	
uint32_t	t2Timer	
uint32_t	t7Timer	
uint32_t	t9Timer	
uint8_t	msdVersion	

4.3.1.27 class telux::tel::IPhone

This class allows getting system information and registering for system events. Each Phone instance is associated with a single SIM. So on a dual SIM device you would have 2 Phone instances.

Public member functions

- virtual telux::common::Status getPhoneId (int &phoneId)=0
- virtual RadioState getRadioState ()=0
- virtual telux::common::Status requestVoiceRadioTechnology (VoiceRadioTechResponseCb callback)=0

- virtual ServiceState getServiceState ()=0
- virtual telux::common::Status requestVoiceServiceState (std::weak_ptr< IVoiceServiceStateCallback > callback)=0
- virtual telux::common::Status setRadioPower (bool enable, std::shared_ptr< telux::common::ICommandResponseCallback > callback=nullptr)=0
- virtual telux::common::Status requestCellInfo (CellInfoCallback callback)=0
- virtual telux::common::Status setCellInfoListRate (uint32_t timeInterval, common::ResponseCallback callback)=0
- virtual telux::common::Status requestSignalStrength (std::shared_ptr< ISignalStrengthCallback > callback=nullptr)=0
- virtual telux::common::Status setECallOperatingMode (ECallMode eCallMode, telux::common::ResponseCallback callback)=0
- virtual telux::common::Status requestECallOperatingMode (ECallGetOperatingModeCallback callback)=0
- virtual telux::common::Status requestOperatorName (OperatorNameCallback callback)=0
- virtual ~IPhone ()

4.3.1.27.1 Constructors and Destructors

4.3.1.27.1.1 virtual telux::tel::IPhone::~IPhone ( ) [virtual]

4.3.1.27.2 Member Function Documentation

4.3.1.27.2.1 virtual telux::common::Status telux::tel::IPhone::getPhoneId (int & *phoneId*) [pure virtual]

Get the Phone ID corresponding to phone.

Parameters

out	*phoneId*	Unique identifier for the phone

Returns

Status of getPhoneId i.e. success or suitable error code.

4.3.1.27.2.2 virtual RadioState telux::tel::IPhone::getRadioState ( ) [pure virtual]

Get Radio state of device.

Returns

RadioState

Deprecated

Use IPhoneManager::requestOperatingMode() API instead

4.3.1.27.2.3 virtual telux::common::Status telux::tel::IPhone::requestVoiceRadioTechnology (VoiceRadioTechResponseCb *callback*) [pure virtual]

Request for Radio technology type (3GPP/3GPP2) used for voice.

Parameters

in	*callback*	callback pointer to get the response of radio power request telux::tel::VoiceRadioTechResponseCb

Returns

Status of requestVoiceRadioTechnology i.e. success or suitable error code telux::common::Status.

Deprecated

Use requestVoiceServiceState() API to get VoiceServiceInfo which has API to get radio technology i.e VoiceServiceInfo::getRadioTechnology()

4.3.1.27.2.4 virtual ServiceState telux::tel::IPhone::getServiceState ( ) [pure virtual]

Get service state of the phone.

Returns

ServiceState

Deprecated

Use requestVoiceServiceState() API

4.3.1.27.2.5 virtual telux::common::Status telux::tel::IPhone::requestVoiceServiceState (std::weak_ptr< IVoiceServiceStateCallback > *callback*) [pure virtual]

Request for voice service state to get the information of phone serving states

Parameters

in	*callback*	callback pointer to get the response of voice service state telux::tel::IVoiceServiceStateCallback.

Returns

Status of requestVoiceServiceState i.e. success or suitable error code telux::common::Status.

4.3.1.27.2.6 virtual telux::common::Status telux::tel::IPhone::setRadioPower (bool *enable*, std::shared_ptr< telux::common::ICommandResponseCallback > *callback* = `nullptr`) `[pure virtual]`

Set the radio power on or off.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_PHONE_MGMT permission to invoke this API successfully.

Parameters

in	*enable*	Flag that determines whether to turn radio on or off
in	*callback*	Optional callback pointer to get the response of set radio power request

Returns

Status of setRadioPower i.e. success or suitable error code.

Deprecated

Use IPhoneManager::setOperatingMode() API instead

4.3.1.27.2.7 virtual telux::common::Status telux::tel::IPhone::requestCellInfo (CellInfoCallback *callback*) `[pure virtual]`

Get the cell information about current serving cell and neighboring cells.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_PRIVATE_INFO_READ permission to invoke this API successfully.

Parameters

in	*callback*	Callback to get the response of cell info request telux::tel::CellInfoCallback

Returns

Status of requestCellInfo i.e. success or suitable error

4.3.1.27.2.8 virtual telux::common::Status telux::tel::IPhone::setCellInfoListRate (uint32_t *timeInterval,* common::ResponseCallback *callback*) [pure virtual]

Set the minimum time in milliseconds between when the cell info list should be received.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_PHONE_CONFIG permission to invoke this API successfully.

Parameters

in	*timeInterval*	Value of 0 means receive cell info list when any info changes. Value of INT_MAX means never receive cell info list even on change. Default value is 0
in	*callback*	Callback to get the response for set cell info list rate.

Returns

Status of setCellInfoListRate i.e. success or suitable error

4.3.1.27.2.9 virtual telux::common::Status telux::tel::IPhone::requestSignalStrength (std::shared_ptr< ISignalStrengthCallback > *callback = nullptr*) [pure virtual]

Get current signal strength of the associated network.

Parameters

in	*callback*	Optional callback pointer to get the response of signal strength request

Returns

Status of requestSignalStrength i.e. success or suitable error code.

4.3.1.27.2.10 virtual telux::common::Status telux::tel::IPhone::setECallOperatingMode (ECallMode *eCallMode,* telux::common::ResponseCallback *callback*) [pure virtual]

Sets the eCall operating mode

On platforms with Access control enabled, Caller needs to have TELUX_TEL_ECALL_CONFIG permission to invoke this API successfully.

Parameters

in	*eCallMode*	- ECallMode
in	*callback*	- Callback function to get the response for set eCall operating mode request.

Returns

Status of setECallOperatingMode i.e. success or suitable error

4.3.1.27.2.11 virtual telux::common::Status telux::tel::IPhone::requestECallOperatingMode (ECallGetOperatingModeCallback *callback*) [pure virtual]

Get the eCall operating mode

Parameters

in	*callback*	- Callback function to get the response of eCall operating mode request

Returns

Status of requestECallOperatingMode i.e. success or suitable error

4.3.1.27.2.12 virtual telux::common::Status telux::tel::IPhone::requestOperatorName (OperatorNameCallback *callback*) [pure virtual]

Get current registered operator name. This API returns PLMN name if available. If not then it returns the SPN configured in the SIM card.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_PRIVATE_INFO_READ permission to invoke this API successfully.

Parameters

in	*callback*	- Callback function to get the response of operator name request

Returns

Status of requestOperatorName i.e. success or suitable error

4.3.1.28 class telux::tel::ISignalStrengthCallback

Interface for Signal strength callback object. Client needs to implement this interface to get single shot responses for commands like get signal strength.

The methods in callback can be invoked from multiple different threads. The implementation should be thread safe.

Public member functions

- virtual void signalStrengthResponse (std::shared_ptr< SignalStrength > signalStrength, telux::common::ErrorCode error)
- virtual ~ISignalStrengthCallback ()

4.3.1.28.1 Constructors and Destructors

4.3.1.28.1.1 virtual telux::tel::ISignalStrengthCallback::~ISignalStrengthCallback ( ) [virtual]

4.3.1.28.2 Member Function Documentation

4.3.1.28.2.1 virtual void telux::tel::ISignalStrengthCallback::signalStrengthResponse (std::shared_ptr< SignalStrength > *signalStrength,* telux::common::ErrorCode *error*) [virtual]

This function is called with the response to requestSignalStrength API.

Parameters

in	*signalStrength*	Pointer to signal strength object
in	*error*	Return code for whether the operation succeeded or failed SUCCESS telux::common::ErrorCode::RADIO_NOT_AVAILABLE

4.3.1.29 class telux::tel::IVoiceServiceStateCallback

Interface for voice service state callback object. Client needs to implement this interface to get single shot responses for commands like request voice radio technology.

The methods in callback can be invoked from multiple different threads. The implementation should be thread safe.

Public member functions

- virtual void voiceServiceStateResponse (const std::shared_ptr< VoiceServiceInfo > &serviceInfo, telux::common::ErrorCode error)
- virtual ~IVoiceServiceStateCallback ()

4.3.1.29.1 Constructors and Destructors

4.3.1.29.1.1 virtual telux::tel::IVoiceServiceStateCallback::~IVoiceServiceStateCallback ( ) [virtual]

4.3.1.29.2 Member Function Documentation

4.3.1.29.2.1 virtual void telux::tel::IVoiceServiceStateCallback::voiceServiceStateResponse (const std::shared_ptr< VoiceServiceInfo > & *serviceInfo,* telux::common::ErrorCode *error*) [virtual]

This function is called with the response to requestVoiceServiceState API.

Parameters

in	*serviceInfo*	Pointer to voice service info object telux::tel::VoiceServiceInfo
in	*error*	Return code for whether the operation succeeded or failed • telux::common::ErrorCode::SUCCESS • telux::common::ErrorCode::RADIO_NOT_AVAILABLE • telux::common::ErrorCode::GENERIC_FAILURE

4.3.1.30 struct telux::tel::SimRatCapability

Structure contains slotID and RAT capabilities corresponding to slot.

Data fields

Type	Field	Description
int	slotId	
RAT↩ Capabilities↩ Mask	capabilities	

4.3.1.31 struct telux::tel::CellularCapabilityInfo

Structure contains information about device capability.

Data fields

Type	Field	Description
VoiceService↩ Technologies↩ Mask	voiceService↩ Techs	Indicates voice support capabilities
int	simCount	The maximum number of SIMs that can be supported simultaneously
int	maxActiveSims	The maximum number of SIMs that can be simultaneously active. If this number is less than numberofSims, it implies that any combination of the SIMs can be active and the remaining can be in standby.
vector< Sim↩ RatCapability >	simRat↩ Capabilities	A Sim inserted in a slot allows for certain rat capabilities. And the UE's HW allows for certain rat capabilities. This field lists the intersection of capabilities allowed by the Sim and the HW. The capabilities are indexed based on slotId.
vector< DeviceRat↩ Capability >	deviceRat↩ Capability	This field lists the Rat capabilities supported by the HW on a given Sim slot. The capabilities are indexed based on slotId.

4.3.1.32 class telux::tel::PhoneFactory

PhoneFactory is the central factory to create all Telephony SDK Classes and services.

Public member functions

- virtual std::shared_ptr< IPhoneManager > getPhoneManager (telux::common::InitResponseCb callback=nullptr)=0
- virtual std::shared_ptr< ISmsManager > getSmsManager (int phoneId=DEFAULT_PHONE_ID, telux::common::InitResponseCb callback=nullptr)=0
- virtual std::shared_ptr< ICallManager > getCallManager (telux::common::InitResponseCb callback=nullptr)=0
- virtual std::shared_ptr< ICardManager > getCardManager ()=0
- virtual std::shared_ptr< ISapCardManager > getSapCardManager (int slotId=DEFAULT_SLOT_ID)=0
- virtual std::shared_ptr< ISubscriptionManager > getSubscriptionManager (telux::common::InitResponseCb callback=nullptr)=0
- virtual std::shared_ptr< IServingSystemManager > getServingSystemManager (int slotId=DEFAULT_SLOT_ID, telux::common::InitResponseCb callback=nullptr)=0
- virtual std::shared_ptr< INetworkSelectionManager > getNetworkSelectionManager (int slotId=DEFAULT_SLOT_ID, telux::common::InitResponseCb callback=nullptr)=0
- virtual std::shared_ptr< IRemoteSimManager > getRemoteSimManager (int slotId=DEFAULT_SLOT_ID)=0
- virtual std::shared_ptr< IMultiSimManager > getMultiSimManager (telux::common::InitResponseCb callback=nullptr)=0
- virtual std::shared_ptr< ICellBroadcastManager > getCellBroadcastManager (SlotId slotId=DEFAULT_SLOT_ID)=0
- virtual std::shared_ptr< ISimProfileManager > getSimProfileManager ()=0
- virtual std::shared_ptr< IImsSettingsManager > getImsSettingsManager (telux::common::InitResponseCb callback=nullptr)=0
- virtual std::shared_ptr< IEcallManager > getEcallManager (telux::common::InitResponseCb callback=nullptr)=0
- virtual std::shared_ptr< IHttpTransactionManager > getHttpTransactionManager (telux::common::InitResponseCb callback=nullptr)=0
- virtual std::shared_ptr< IImsServingSystemManager > getImsServingSystemManager (SlotId slotId, telux::common::InitResponseCb callback=nullptr)=0
- virtual std::shared_ptr< ISuppServicesManager > getSuppServicesManager (SlotId slotId=DEFAULT_SLOT_ID, telux::common::InitResponseCb callback=nullptr)=0

Static Public Member Functions

- static PhoneFactory & getInstance ()

4.3.1.32.1 Member Function Documentation

4.3.1.32.1.1 static PhoneFactory& telux::tel::PhoneFactory::getInstance ( ) [static]

Get Phone Factory instance.

4.3.1.32.1.2 virtual std::shared_ptr<IPhoneManager> telux::tel::PhoneFactory::getPhoneManager (telux::common::InitResponseCb *callback* = *nullptr*) [pure virtual]

Get Phone Manager instance. Phone Manager is the main entry point into the telephony subsystem.

Parameters

in	*callback*	Optional callback pointer to get response of Phone Manager initialization. It will be invoked when initialization is either succeeded or failed. In case of failure response, the provided Phone Manager object will no more be a valid object.

Returns

Pointer of IPhoneManager object.

4.3.1.32.1.3 virtual std::shared_ptr<ISmsManager> telux::tel::PhoneFactory::getSmsManager (int *phoneId* = *DEFAULT_PHONE_ID*, telux::common::InitResponseCb *callback* = *nullptr*) [pure virtual]

Get SMS Manager instance for Phone ID. SMSManager used to send and receive SMS messages.

Parameters

in	*phoneId*	Unique identifier for the phone
in	*callback*	Optional callback pointer to get response of SMS Manager initialization. It will be invoked when initialization is either succeeded or failed. In case of failure response, the provided SMS Manager object will no more be a valid object.

Returns

Pointer of ISmsManager object or nullptr in case of failure.

4.3.1.32.1.4 virtual std::shared_ptr<ICallManager> telux::tel::PhoneFactory::getCallManager (telux::common::InitResponseCb *callback = nullptr*) [pure virtual]

Get Call Manager instance to determine state of active calls and perform other functions like dial, conference, swap call.

Parameters

in	*callback*	Optional callback pointer to get response of CallManager initialization. It will be invoked when initialization is either succeeded or failed. In case of failure response, the provided Call Manager object will no more be a valid object.

Returns

Pointer of ICallManager object or nullptr in case of failure.

4.3.1.32.1.5 virtual std::shared_ptr<ICardManager> telux::tel::PhoneFactory::getCardManager ( ) [pure virtual]

Get Card Manager instance to handle services such as transmitting APDU, SIM IO and more.

Returns

Pointer of ICardManager object.

4.3.1.32.1.6 virtual std::shared_ptr<ISapCardManager> telux::tel::PhoneFactory::getSapCardManager (int *slotId = DEFAULT_SLOT_ID*) [pure virtual]

Get Sap Card Manager instance associated with the provided slot id. This object will handle services in SAP mode such as APDU, SIM Power On/Off and SIM reset.

On platforms with access control enabled, caller needs to have TELUX_TEL_SAP permission to invoke this API successfully.

Parameters

in	*slotId*	Unique identifier for the SIM slot

Returns

Pointer of ISapCardManager object.

4.3.1.32.1.7 virtual std::shared_ptr<ISubscriptionManager> telux::tel::PhoneFactory::getSubscriptionManager (telux::common::InitResponseCb *callback = nullptr*) [pure virtual]

Get Subscription Manager instance to get device subscription details

Parameters

in	*callback*	Optional callback pointer to get response of Phone Manager initialization. It will be invoked when initialization is either succeeded or failed. In case of failure response, the provided SubscriptionManager object will no more be a valid object.

Returns

Pointer of ISubscriptionManager object.

4.3.1.32.1.8 virtual std::shared_ptr<IServingSystemManager> telux::tel::PhoneFactory::getServing↩ SystemManager (int *slotId = DEFAULT_SLOT_ID,* telux::common::InitResponseCb *callback = nullptr*) [pure virtual]

Get Serving System Manager instance to get and set preferred network type.

Parameters

in	*slotId*	Unique identifier for the SIM slot
in	*callback*	Optional callback pointer to get the response of the manager initialisation.

Returns

Pointer of IServingSystemManager object.

4.3.1.32.1.9 virtual std::shared_ptr<INetworkSelectionManager> telux::tel::PhoneFactory::get↩ NetworkSelectionManager (int *slotId = DEFAULT_SLOT_ID,* telux::common::Init↩ ResponseCb *callback = nullptr*) [pure virtual]

Get Network Selection Manager instance to get and set selection mode, get and set preferred networks and scan available networks.

Parameters

in	*slotId*	Unique identifier for the SIM slot
in	*callback*	Optional callback pointer to get the response of the manager initialisation.

Returns

Pointer of INetworkSelectionManager object.

4.3.1.32.1.10 virtual std::shared_ptr<IRemoteSimManager> telux::tel::PhoneFactory::getRemoteSimManager (int *slotId = DEFAULT_SLOT_ID*) [pure virtual]

Get Remote SIM Manager instance to handle services like exchanging APDU, SIM Power On/Off, etc.

On platforms with access control enabled, caller needs to have TELUX_TEL_REMOTE_SIM permission to invoke this API successfully.

Parameters

in	*slotId*	Unique identifier for the SIM slot

Returns

Pointer of IRemoteSimManager object.

4.3.1.32.1.11 virtual std::shared_ptr<IMultiSimManager> telux::tel::PhoneFactory::getMultiSimManager (telux::common::InitResponseCb *callback = nullptr*) [pure virtual]

Get Multi SIM Manager instance to handle operations like high capabilty switch.

Parameters

in	*callback*	Optional callback pointer to get response of MultiSimManager initialization. It will be invoked when initialization is either succeeded or failed. In case of failure response, the provided MultiSimManager object will no more be a valid object.

Returns

Pointer of IMultiSimManager object.

4.3.1.32.1.12 virtual std::shared_ptr<ICellBroadcastManager> telux::tel::PhoneFactory::getCellBroadcastManager (SlotId *slotId = DEFAULT_SLOT_ID*) [pure virtual]

Get CellBroadcast Manager instance for Slot ID. CellBroadcast manager used to receive broacast messages and configure broadcast messages.

Parameters

in	*slotId*	telux::common::SlotId

Returns

Pointer of ICellBroadcastManager object or nullptr in case of failure.

4.3.1.32.1.13 virtual std::shared_ptr<ISimProfileManager> telux::tel::PhoneFactory::getSimProfileManager ( ) [pure virtual]

Get SimProfileManager. SimProfileManager is a primary interface for remote eUICC(eSIM) provisioning and local profile assistance.

Returns

Pointer of ISimProfileManager object or nullptr in case of failure.

4.3.1.32.1.14 virtual std::shared_ptr<IImsSettingsManager> telux::tel::PhoneFactory::getImsSettingsManager (telux::common::InitResponseCb *callback* = *nullptr*) [pure virtual]

Get Ims Settings Manager instance to handle IMS service enable configuation parameters like enable/disable voIMS.

Parameters

in	*callback*	Optional callback pointer to get the response of the manager initialisation.

Returns

Pointer of IImsSettingsManager object.

4.3.1.32.1.15 virtual std::shared_ptr<IEcallManager> telux::tel::PhoneFactory::getEcallManager (telux::common::InitResponseCb *callback* = *nullptr*) [pure virtual]

4.3.1.32.1.16 virtual std::shared_ptr<IHttpTransactionManager> telux::tel::PhoneFactory::getHttpTransactionManager (telux::common::InitResponseCb *callback* = *nullptr*) [pure virtual]

Get HttpTransactionManager instance to handle HTTP related requests from the modem for SIM profile update related operations.

Parameters

in	*callback*	Optional callback pointer to get the response of the manager initialisation.

Returns

Pointer of IHttpTransactionManager object or nullptr in case of failure.

4.3.1.32.1.17 virtual std::shared_ptr<IImsServingSystemManager> telux::tel::PhoneFactory::get↩ ImsServingSystemManager (SlotId *slotId,* telux::common::InitResponseCb *callback = nullptr*) [pure virtual]

Get IMS Serving System Manager instance to query IMS registration status

Returns

Pointer of IImsServingSystemManager object or nullptr in case of failure.

4.3.1.32.1.18 virtual std::shared_ptr<ISuppServicesManager> telux::tel::PhoneFactory::getSupp↩ ServicesManager (SlotId *slotId = DEFAULT_SLOT_ID,* telux::common::InitResponseCb *callback = nullptr*) [pure virtual]

Get Supplementary service manager instance to set/get preference for supplementary services like call waiting, call forwarding etc.

Parameters

in	*SlotId*	telux::common::SlotId
in	*callback*	Optional callback pointer to get the response of the manager initialisation.

Returns

Pointer of ISuppServicesManager object.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.3.1.33 class telux::tel::IPhoneListener

A listener class for monitoring changes in specific telephony states on the device, including service state and signal strength. Override the methods for the state that you wish to receive updates for.

The methods in listener can be invoked from multiple different threads. The implementation should be thread safe.

Public member functions

- virtual void onServiceStateChanged (int phoneId, ServiceState state)
- virtual void onSignalStrengthChanged (int phoneId, std::shared_ptr< SignalStrength > signalStrength)
- virtual void onCellInfoListChanged (int phoneId, std::vector< std::shared_ptr< CellInfo >> cellInfoList)
- virtual void onRadioStateChanged (int phoneId, RadioState radioState)

- virtual void onVoiceRadioTechnologyChanged (int phoneId, RadioTechnology radioTech)
- virtual void onVoiceServiceStateChanged (int phoneId, const std::shared_ptr< VoiceServiceInfo > &serviceInfo)
- virtual void onOperatingModeChanged (OperatingMode mode)
- virtual void onECallOperatingModeChange (int phoneId, telux::tel::ECallModeInfo info)
- virtual ~IPhoneListener ()

4.3.1.33.1 Constructors and Destructors

4.3.1.33.1.1 virtual telux::tel::IPhoneListener::~IPhoneListener ( ) [virtual]

4.3.1.33.2 Member Function Documentation

4.3.1.33.2.1 virtual void telux::tel::IPhoneListener::onServiceStateChanged (int *phoneId,* ServiceState *state*) [virtual]

This function is called when device service state changes.

Parameters

in	*phoneId*	Unique id of the phone on which service state changed.
in	*state*	Service state of the phone ServiceState

Deprecated

Use onVoiceServiceStateChanged() listener

4.3.1.33.2.2 virtual void telux::tel::IPhoneListener::onSignalStrengthChanged (int *phoneId,* std::shared_ptr< SignalStrength > *signalStrength*) [virtual]

This function is called when network signal strength changes.

Parameters

in	*phoneId*	Unique id of the phone on which signal strength state changed.
in	*signalStrength*	Pointer to signal strength object

4.3.1.33.2.3 virtual void telux::tel::IPhoneListener::onCellInfoListChanged (int *phoneId,* std::vector< std::shared_ptr< CellInfo >> *cellInfoList*) [virtual]

This function is called when info pertaining to current or neighboring cells change.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_PRIVATE_INFO_READ permission to receive this notification.

Parameters

in	*phoneId*	Unique id of the phone on which cell info changed.
in	*cellInfoList*	vector of shared pointers to cell info object

4.3.1.33.2.4 virtual void telux::tel::IPhoneListener::onRadioStateChanged (int *phoneId,* RadioState *radioState*) [virtual]

This function is called when radio state changes on phone

Parameters

in	*phoneId*	Unique id of the phone on which radio state changed
in	*radioState*	Radio state of the phone RadioState

Deprecated

Use onOperatingModeChanged() API instead

4.3.1.33.2.5 virtual void telux::tel::IPhoneListener::onVoiceRadioTechnologyChanged (int *phoneId,* RadioTechnology *radioTech*) [virtual]

This function is called when the radio technology for voice service changes

Parameters

in	*phoneId*	Unique id of the phone on which radio technology changed
in	*radioTech*	Radio state of the phone telux::tel::RadioTechnology

Deprecated

Use onVoiceServiceStateChanged() API instead

4.3.1.33.2.6 virtual void telux::tel::IPhoneListener::onVoiceServiceStateChanged (int *phoneId,* const std::shared_ptr< VoiceServiceInfo > & *serviceInfo*) [virtual]

This function is called when the service state for voice service changes

Parameters

in	*phoneId*	Unique id of the phone on which radio technology changed
in	*serviceInfo*	pointer of voice service state info object telux::tel::VoiceServiceInfo

4.3.1.33.2.7 virtual void telux::tel::IPhoneListener::onOperatingModeChanged (OperatingMode *mode*) [virtual]

This function is called when the operating mode changes

Parameters

in	*mode*	Operating mode OperatingMode.

4.3.1.33.2.8 virtual void telux::tel::IPhoneListener::onECallOperatingModeChange (int *phoneId,* telux::tel::ECallModeInfo *info*) [virtual]

This function is called when eCall operating mode changes.

Parameters

in	*phoneId*	- Unique Id of phone for which eCall operating mode changed
in	*info*	- Indicates eCall operating mode change reason ECallModeInfo

4.3.1.34 class telux::tel::IPhoneManager

Phone Manager creates one or more phones based on SIM slot count, it allows clients to register for notification of system events. Clients should check if the subsystem is ready before invoking any of the APIs.

Public member functions

- virtual bool isSubsystemReady ()=0
- virtual std::future< bool > onSubsystemReady ()=0
- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual telux::common::Status getPhoneIds (std::vector< int > &phoneIds)=0
- virtual int getPhoneIdFromSlotId (int slotId)=0
- virtual int getSlotIdFromPhoneId (int phoneId)=0
- virtual std::shared_ptr< IPhone > getPhone (int phoneId=DEFAULT_PHONE_ID)=0
- virtual telux::common::Status requestCellularCapabilityInfo (std::shared_ptr< ICellularCapabilityCallback > callback=nullptr)=0
- virtual telux::common::Status requestOperatingMode (std::shared_ptr< IOperatingModeCallback > callback=nullptr)=0
- virtual telux::common::Status setOperatingMode (OperatingMode operatingMode, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status resetWwan (telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status registerListener (std::weak_ptr< IPhoneListener > listener)=0
- virtual telux::common::Status removeListener (std::weak_ptr< IPhoneListener > listener)=0
- virtual ~IPhoneManager ()

4.3.1.34.1 Constructors and Destructors

4.3.1.34.1.1 virtual telux::tel::IPhoneManager::~IPhoneManager ( ) [virtual]

4.3.1.34.2 Member Function Documentation

4.3.1.34.2.1 virtual bool telux::tel::IPhoneManager::isSubsystemReady ( ) [pure virtual]

Checks the status of telephony subsystems and returns the result.

Returns

If true PhoneManager is ready for service (i.e Phone, Sms and Card).

Deprecated

Use IPhoneManager::getServiceStatus() API.

4.3.1.34.2.2 virtual std::future<bool> telux::tel::IPhoneManager::onSubsystemReady ( ) [pure virtual]

Wait for telephony subsystem to be ready.

Returns

A future that caller can wait on to be notified when telephony subsystem is ready.

Deprecated

Use InitResponseCb in PhoneFactory::getPhoneManager instead, to get notified about subsystem readiness.

4.3.1.34.2.3 virtual telux::common::ServiceStatus telux::tel::IPhoneManager::getServiceStatus ( ) [pure virtual]

This status indicates whether the IServingSystemManager object is in a usable state.

Returns

SERVICE_AVAILABLE - If Serving System manager is ready for service.
SERVICE_UNAVAILABLE - If Serving System manager is temporarily unavailable.
SERVICE_FAILED - If Serving System manager encountered an irrecoverable failure.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.3.1.34.2.4 virtual telux::common::Status telux::tel::IPhoneManager::getPhoneIds (std::vector< int > & *phoneIds*) [pure virtual]

Retrieves a list of Phone Ids. Each id is unique per phone. For example: on a dual SIM device, there would be 2 Phones.

Parameters

out	*phoneIds*	List of phone ids

Returns

Status of getPhoneIds i.e. success or suitable error code.

4.3.1.34.2.5 virtual int telux::tel::IPhoneManager::getPhoneIdFromSlotId (int *slotId*) [pure virtual]

Get the Phone Id for a given Slot Id.

Parameters

in	*slotId*	SIM Card Slot Id

Returns

Phone Id corresponding to the Slot Id.

4.3.1.34.2.6 virtual int telux::tel::IPhoneManager::getSlotIdFromPhoneId (int *phoneId*) [pure virtual]

Get the SIM Slot Id for a given Phone Id.

Parameters

in	*phoneId*	Phone Id of the phone

Returns

Slot Id corresponding to the Phone Id.

4.3.1.34.2.7 virtual std::shared_ptr<IPhone> telux::tel::IPhoneManager::getPhone (int *phoneId = DEFAULT_PHONE_ID*) [pure virtual]

Get the phone instance for a given phone identifier.

Parameters

in	*phoneId*	Identifier for phone instance, retrieved from getPhoneIds API

Returns

Pointer to Phone object corresponding to phoneId.

4.3.1.34.2.8 virtual telux::common::Status telux::tel::IPhoneManager::requestCellularCapabilityInfo (std::shared_ptr< ICellularCapabilityCallback > *callback = nullptr*) [pure virtual]

Get the information about cellular capability.

Parameters

in	*callback*	Optional callback pointer to get the response of cellular capability.

Returns

Status of requestCellularCapabilityInfo i.e. success or suitable error code.

4.3.1.34.2.9 virtual telux::common::Status telux::tel::IPhoneManager::requestOperatingMode (std::shared_ptr< IOperatingModeCallback > *callback = nullptr*) [pure virtual]

Get current operating mode of the device.

Parameters

in	*callback*	Optional callback pointer to get the response of operating mode request

Returns

Status of requestOperatingMode i.e. success or suitable error code.

4.3.1.34.2.10 virtual telux::common::Status telux::tel::IPhoneManager::setOperatingMode (OperatingMode *operatingMode,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

Set the operating mode of the device. Only valid transitions allowed from one mode to another.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_PHONE_MGMT permission to invoke this API successfully.

Parameters

in	*operatingMode*	Operating Mode to be set
in	*callback*	Optional callback pointer to get the response of set operatingmode request. In callback following error is returned. • telux::common::ErrorCode::INVALID_TRANSITION • telux::common::ErrorCode::INVALID_ARGUMENTS • telux::common::ErrorCode::DEVICE_IN_USE • telux::common::ErrorCode::NO_MEMORY

Returns

Status of setOperatingMode i.e. success or suitable error code.

4.3.1.34.2.11 virtual telux::common::Status telux::tel::IPhoneManager::resetWwan (telux::common::↩ ResponseCallback *callback* = `nullptr`) `[pure virtual]`

Reset the WWAN stack on the modem without impacting the CV2x stack. This does a soft-reset of some of the subsystems on the modem. Some subsystems like data services is not impacted by this trigger, so it is recommended to stop any WWAN data calls using telux::data::IDataConnectionManager::stopDataCall before invoking this API. This API will be rejected when under the scenarios of voice call, emergency call, emergency SMS, and emergency call back mode.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_PHONE_MGMT permission to invoke this API successfully.

Parameters

in	*callback*	Optional callback pointer to get the response of WWAN reset request.

Returns

Status of resetWwan i.e. success or suitable error code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.3.1.34.2.12 virtual telux::common::Status telux::tel::IPhoneManager::registerListener (std::weak_↩ ptr< IPhoneListener > *listener*) `[pure virtual]`

Register a listener for specific events in the telephony subsystem.

Parameters

in	*listener*	Pointer to Phone Listener object that processes the notification

Returns

Status of registerListener i.e. success or suitable error code.

4.3.1.34.2.13 virtual telux::common::Status telux::tel::IPhoneManager::removeListener (std::weak_ptr< IPhoneListener > *listener*) [pure virtual]

Remove a previously added listener.

Parameters

in	*listener*	Pointer to Phone Listener object that needs to be removed

Returns

Status of removeListener i.e. success or suitable error code.

4.3.1.35 class telux::tel::ICellularCapabilityCallback

Interface for callback corresponding to cellular capability request. Client needs to implement this interface to get single shot responses for commands like get cellular capability.

The methods in callback can be invoked from multiple different threads. The implementation should be thread safe.

Public member functions

- virtual void cellularCapabilityResponse (CellularCapabilityInfo capabilityInfo, telux::common::ErrorCode error)
- virtual ~ICellularCapabilityCallback ()

4.3.1.35.1 Constructors and Destructors

4.3.1.35.1.1 virtual telux::tel::ICellularCapabilityCallback::~ICellularCapabilityCallback ( ) [virtual]

4.3.1.35.2 Member Function Documentation

4.3.1.35.2.1 virtual void telux::tel::ICellularCapabilityCallback::cellularCapabilityResponse (CellularCapabilityInfo *capabilityInfo,* telux::common::ErrorCode *error*) [virtual]

This function is called with the response to requestCellularCapabilityInfo API.

Parameters

in	*capabilityInfo*	Cellular capability information.
in	*error*	Return code for whether the operation succeeded or failed • telux::common::ErrorCode::SUCCESS • telux::common::ErrorCode::INTERNAL • telux::common::ErrorCode::NO_MEMORY

4.3.1.36 class telux::tel::IOperatingModeCallback

Interface for operating mode callback object. Client needs to implement this interface to get single shot responses for commands like request current operating mode.

The methods in callback can be invoked from multiple different threads. The implementation should be thread safe.

Public member functions

- virtual void operatingModeResponse (OperatingMode operatingMode, telux::common::ErrorCode error)
- virtual ~IOperatingModeCallback ()

4.3.1.36.1 Constructors and Destructors

4.3.1.36.1.1 virtual telux::tel::IOperatingModeCallback::~IOperatingModeCallback ( ) [virtual]

4.3.1.36.2 Member Function Documentation

4.3.1.36.2.1 virtual void telux::tel::IOperatingModeCallback::operatingModeResponse (OperatingMode *operatingMode*, telux::common::ErrorCode *error*) [virtual]

This function is called with the response to requestOperatingMode API.

Parameters

in	*operatingMode*	OperatingMode
in	*error*	Return code for whether the operation succeeded or failed • telux::common::ErrorCode::SUCCESS • telux::common::ErrorCode::INTERNAL_ERR • telux::common::ErrorCode::NO_MEMORY

4.3.1.37 class telux::tel::SignalStrength

SignalStrength class provides access to LTE, GSM, CDMA, WCDMA, TDSCDMA signal strengths.

Public member functions

- SignalStrength (std::shared_ptr< LteSignalStrengthInfo > lteSignalStrengthInfo, std::shared_ptr< GsmSignalStrengthInfo > gsmSignalStrengthInfo, std::shared_ptr< CdmaSignalStrengthInfo > cdmaSignalStrengthInfo, std::shared_ptr< WcdmaSignalStrengthInfo > wcdmaSignalStrengthInfo, std::shared_ptr< TdscdmaSignalStrengthInfo > tdscdmaSignalStrengthInfo, std::shared_ptr< Nr5gSignalStrengthInfo > nr5gSignalStrengthInfo)
- std::shared_ptr< LteSignalStrengthInfo > getLteSignalStrength ()
- std::shared_ptr< GsmSignalStrengthInfo > getGsmSignalStrength ()
- std::shared_ptr< CdmaSignalStrengthInfo > getCdmaSignalStrength ()
- std::shared_ptr< WcdmaSignalStrengthInfo > getWcdmaSignalStrength ()
- std::shared_ptr< TdscdmaSignalStrengthInfo > getTdscdmaSignalStrength ()
- std::shared_ptr< Nr5gSignalStrengthInfo > getNr5gSignalStrength ()

4.3.1.37.1 Constructors and Destructors

4.3.1.37.1.1 telux::tel::SignalStrength::SignalStrength (std::shared_ptr< LteSignalStrengthInfo > *lteSignalStrengthInfo*, std::shared_ptr< GsmSignalStrengthInfo > *gsmSignalStrengthInfo*, std::shared_ptr< CdmaSignalStrengthInfo > *cdmaSignalStrengthInfo*, std::shared_ptr< WcdmaSignalStrengthInfo > *wcdmaSignalStrengthInfo*, std::shared_ptr< TdscdmaSignalStrengthInfo > *tdscdmaSignalStrengthInfo*, std::shared_ptr< Nr5gSignalStrengthInfo > *nr5gSignalStrengthInfo*)

4.3.1.37.2 Member Function Documentation

4.3.1.37.2.1 std::shared_ptr<LteSignalStrengthInfo> telux::tel::SignalStrength::getLteSignalStrength ( )

Gives LTE signal strength instance.

Returns

Pointer to LTE signal strength instance that can be used to get lte dbm, signal level values.

4.3.1.37.2.2 std::shared_ptr<GsmSignalStrengthInfo> telux::tel::SignalStrength::getGsmSignalStrength ( )

Gives GSM signal strength instance.

Returns

Pointer to GSM signal strength instance that can be used to get GSM dbm, signal level values.

4.3.1.37.2.3 std::shared_ptr<CdmaSignalStrengthInfo> telux::tel::SignalStrength::getCdmaSignal↩ Strength ( )

Gives CDMA signal strength instance.

Returns

Pointer to CDMA signal strength instance that can be used to get cdma/evdo dbm, signal level values.

Deprecated

As of version 1.53.0 this API is no longer supported.

4.3.1.37.2.4 std::shared_ptr<WcdmaSignalStrengthInfo> telux::tel::SignalStrength::getWcdmaSignal↩ Strength ( )

Gives WCDMA signal strength instance.

Returns

Pointer to WCDMA signal strength instance that can be used to get WCDMA dbm, signal level values.

4.3.1.37.2.5 std::shared_ptr<TdscdmaSignalStrengthInfo> telux::tel::SignalStrength::getTdscdma↩ SignalStrength ( )

Gives TDSWCDMA signal strength instance.

Returns

Pointer to TDSWCDMA signal strength instance that can be used to get TDSCDMA RSCP value.

Deprecated

As of version 1.53.0 this API is no longer supported.

4.3.1.37.2.6 std::shared_ptr<Nr5gSignalStrengthInfo> telux::tel::SignalStrength::getNr5gSignal↩ Strength ( )

Gives 5G NR signal strength instance.

Returns

Pointer to 5G NR signal strength instance that can be used to get 5G NR dbm and snr values.

4.3.1.38 class telux::tel::LteSignalStrengthInfo

LTE signal strength class provides methods to get details of lte signals like dbm, signal level, reference signal-to-noise ratio, channel quality indicator and signal strength.

Public member functions

- LteSignalStrengthInfo (int lteSignalStrength, int lteRsrp, int lteRsrq, int lteRssnr, int lteCqi, int timingAdvance)
- const SignalStrengthLevel getLevel () const
- const int getDbm () const
- const int getLteSignalStrength () const
- const int getLteReferenceSignalReceiveQuality () const
- const int getLteReferenceSignalSnr () const
- const int getLteChannelQualityIndicator () const
- const int getTimingAdvance () const

4.3.1.38.1 Constructors and Destructors

4.3.1.38.1.1 telux::tel::LteSignalStrengthInfo::LteSignalStrengthInfo (int *lteSignalStrength,* int *lteRsrp,* int *lteRsrq,* int *lteRssnr,* int *lteCqi,* int *timingAdvance*)

4.3.1.38.2 Member Function Documentation

4.3.1.38.2.1 const SignalStrengthLevel telux::tel::LteSignalStrengthInfo::getLevel ( ) const

Get signal level in the range.

Returns

Signal levels indicates the quality of signal being received by the device.

4.3.1.38.2.2 const int telux::tel::LteSignalStrengthInfo::getDbm ( ) const

Get the signal strength in dBm. (Valid value range [-140, -44] and INVALID_SIGNAL_STRENGTH_VALUE i.e. unavailable).

Returns

LTE dBm value.

4.3.1.38.2.3 const int telux::tel::LteSignalStrengthInfo::getLteSignalStrength ( ) const

Get the LTE signal strength. (Valid value range [0, 31] and INVALID_SIGNAL_STRENGTH_VALUE i.e. unavailable).

Returns

LTE signal strength.

4.3.1.38.2.4 const int telux::tel::LteSignalStrengthInfo::getLteReferenceSignalReceiveQuality ( ) const

Get LTE reference signal receive quality in dB. (Valid value range [-20, -3] and INVALID_SIGNAL_STRENGTH_VALUE i.e. unavailable).

Returns

LteRsrq.

4.3.1.38.2.5 const int telux::tel::LteSignalStrengthInfo::getLteReferenceSignalSnr ( ) const

Get LTE reference signal signal-to-noise ratio, multiply by 0.1 to get SNR in dB. (Valid value range [-200, +300] and INVALID_SIGNAL_STRENGTH_VALUE i.e. unavailable). (-200 = -20.0 dB, +300 = 30dB).

Returns

LteSnr.

4.3.1.38.2.6 const int telux::tel::LteSignalStrengthInfo::getLteChannelQualityIndicator ( ) const

Get LTE channel quality indicator. (Valid value range [0, 15] and INVALID_SIGNAL_STRENGTH_VALUE i.e. unavailable).

Deprecated

This API not being supported

Returns

LteCqI.

Deprecated

As of version 1.54.0 this API is no longer supported.

4.3.1.38.2.7 const int telux::tel::LteSignalStrengthInfo::getTimingAdvance ( ) const

Get the timing advance in micro seconds. (Valid value range [0, 0x7FFFFFFE] and INVALID_SIGNAL_STRENGTH_VALUE i.e. unavailable).

Deprecated

This API not being supported

Returns

Timing advance value.

4.3.1.39 class telux::tel::GsmSignalStrengthInfo

GSM signal strength provides methods to get GSM signal strength in dBm and GSM signal level.

Public member functions

- GsmSignalStrengthInfo (int gsmSignalStrength, int gsmBitErrorRate, int timingAdvance)
- const SignalStrengthLevel getLevel () const
- const int getDbm () const
- const int getGsmSignalStrength () const
- const int getGsmBitErrorRate () const
- const int getTimingAdvance ()

4.3.1.39.1 Constructors and Destructors

4.3.1.39.1.1 telux::tel::GsmSignalStrengthInfo::GsmSignalStrengthInfo (int *gsmSignalStrength,* int *gsmBitErrorRate,* int *timingAdvance*)

4.3.1.39.2 Member Function Documentation

4.3.1.39.2.1 const SignalStrengthLevel telux::tel::GsmSignalStrengthInfo::getLevel ( ) const

Get signal level in the range.

Returns

Signal levels indicates the quality of signal being received by the device.

4.3.1.39.2.2 const int telux::tel::GsmSignalStrengthInfo::getDbm ( ) const

Get the signal strength in dBm. (Valid value range [-113, -51] and
INVALID_SIGNAL_STRENGTH_VALUE i.e. unavailable).

Returns

GSM signal strength in dBm.

4.3.1.39.2.3 const int telux::tel::GsmSignalStrengthInfo::getGsmSignalStrength ( ) const

Get the GSM signal strength. (Valid value range [0, 31] and INVALID_SIGNAL_STRENGTH_VALUE i.e. unavailable).

Returns

GSM signal strength.

4.3.1.39.2.4 const int telux::tel::GsmSignalStrengthInfo::getGsmBitErrorRate ( ) const

Get the GSM bit error rate. (Valid value range [0, 7] and INVALID_SIGNAL_STRENGTH_VALUE i.e. unavailable).

Deprecated

This API not being supported

Returns

GSM bit error rate.

4.3.1.39.2.5 const int telux::tel::GsmSignalStrengthInfo::getTimingAdvance ( )

Get the timing advance in bit periods . 1 bit period = 48/13 us (Valid value range [0, 219] and INVALID_SIGNAL_STRENGTH_VALUE i.e. unavailable).

Deprecated

This API not being supported

Returns

timing advance.

4.3.1.40 class telux::tel::CdmaSignalStrengthInfo

CDMA signal strength provides methods to get details of CDMA and EVDO like signal strength in dBm and signal level.

Deprecated

As of version 1.53.0 this API is no longer supported.

Public member functions

- CdmaSignalStrengthInfo (int cdmaDbm, int cdmaEcio, int evdoDbm, int evdoEcio, int evdoSignalNoiseRatio)
- const SignalStrengthLevel getLevel () const
- const int getDbm () const
- const int getCdmaEcio () const
- const int getEvdoEcio () const
- const int getEvdoSignalNoiseRatio () const

4.3.1.40.1 Constructors and Destructors

4.3.1.40.1.1 telux::tel::CdmaSignalStrengthInfo::CdmaSignalStrengthInfo (int *cdmaDbm,* int *cdmaEcio,* int *evdoDbm,* int *evdoEcio,* int *evdoSignalNoiseRatio*)

4.3.1.40.2 Member Function Documentation

4.3.1.40.2.1 const SignalStrengthLevel telux::tel::CdmaSignalStrengthInfo::getLevel ( ) const

Get signal level in the range.

Returns

Signal levels indicates the quality of signal being received by the device.

4.3.1.40.2.2 const int telux::tel::CdmaSignalStrengthInfo::getDbm ( ) const

Get the signal strength in dBm.

Returns

Minimum value of Evdo dBm and Cdma dBm.

4.3.1.40.2.3 const int telux::tel::CdmaSignalStrengthInfo::getCdmaEcio ( ) const

Get the CDMA Ec/Io in dB.

Returns

CDMA Ec/Io.

4.3.1.40.2.4 const int telux::tel::CdmaSignalStrengthInfo::getEvdoEcio ( ) const

Get the EVDO Ec/Io in dB.

Returns

EVDO Ec/Io.

4.3.1.40.2.5 const int telux::tel::CdmaSignalStrengthInfo::getEvdoSignalNoiseRatio ( ) const

Get the EVDO signal noise ratio. (Valid value range [0, 8] and 8 is the highest signal to noise ratio.

Returns

EVDO SNR.

4.3.1.41 class telux::tel::WcdmaSignalStrengthInfo

WCDMA signal strength provides methods to get WCDMA signal strength in dBm and WCDMA signal level.

Public member functions

- WcdmaSignalStrengthInfo (int signalStrength, int bitErrorRate)
- const SignalStrengthLevel getLevel () const
- const int getDbm () const
- const int getSignalStrength () const
- const int getBitErrorRate () const

4.3.1.41.1 Constructors and Destructors

4.3.1.41.1.1 telux::tel::WcdmaSignalStrengthInfo::WcdmaSignalStrengthInfo (int *signalStrength,* int *bitErrorRate*)

4.3.1.41.2 Member Function Documentation

4.3.1.41.2.1 const SignalStrengthLevel telux::tel::WcdmaSignalStrengthInfo::getLevel ( ) const

Get signal level in the range.

Returns

Signal levels indicates the quality of signal being received by the device.

4.3.1.41.2.2 const int telux::tel::WcdmaSignalStrengthInfo::getDbm ( ) const

Get the signal strength in dBm. (Valid value range [-113, -51] and INVALID_SIGNAL_STRENGTH_VALUE i.e. unavailable).

Returns

WCDMA signal strength in dBm.

4.3.1.41.2.3 const int telux::tel::WcdmaSignalStrengthInfo::getSignalStrength ( ) const

Get the WCDMA signal strength. (Valid value range [0, 31] and INVALID_SIGNAL_STRENGTH_VALUE i.e. unavailable).

Returns

WCDMA signal strength.

4.3.1.41.2.4 const int telux::tel::WcdmaSignalStrengthInfo::getBitErrorRate ( ) const

Get the WCDMA bit error rate. (Valid value range [0, 7] and INVALID_SIGNAL_STRENGTH_VALUE i.e. unavailable).

Deprecated

This API not being supported

Returns

WCDMA bit error rate.

4.3.1.42 class telux::tel::TdscdmaSignalStrengthInfo

Tdscdma signal strength provides methods to get received signal code power.

Deprecated

As of version 1.53.0 this API is no longer supported.

Public member functions

- TdscdmaSignalStrengthInfo (int rscp)
- const int getRscp () const

4.3.1.42.1 Constructors and Destructors

4.3.1.42.1.1 telux::tel::TdscdmaSignalStrengthInfo::TdscdmaSignalStrengthInfo (int *rscp*)

4.3.1.42.2 Member Function Documentation

4.3.1.42.2.1 const int telux::tel::TdscdmaSignalStrengthInfo::getRscp ( ) const

Get TdScdma received signal code power in dBm. (Valid Range [-120,-25], and INVALID_SIGNAL_STRENGTH_VALUE i.e. unavailable).

Returns

TdScdma signal code power.

4.3.1.43 class telux::tel::Nr5gSignalStrengthInfo

5G NR signal strength provides methods to get signal strength and signal-to-noise ratio.

Public member functions

- Nr5gSignalStrengthInfo (int rsrp, int rsrq, int rssnr)
- const SignalStrengthLevel getLevel () const
- const int getDbm () const
- const int getReferenceSignalReceiveQuality () const
- const int getReferenceSignalSnr () const

4.3.1.43.1 Constructors and Destructors

4.3.1.43.1.1 telux::tel::Nr5gSignalStrengthInfo::Nr5gSignalStrengthInfo (int *rsrp,* int *rsrq,* int *rssnr*)

4.3.1.43.2 Member Function Documentation

4.3.1.43.2.1 const SignalStrengthLevel telux::tel::Nr5gSignalStrengthInfo::getLevel ( ) const

Get signal level in the range.

Returns

Signal levels indicates the quality of signal being received by the device.

4.3.1.43.2.2 const int telux::tel::Nr5gSignalStrengthInfo::getDbm ( ) const

Get the signal strength in dBm. (Valid value range [-140, -44] and INVALID_SIGNAL_STRENGTH_VALUE i.e. unavailable). INVALID_SIGNAL_STRENGTH_VALUE indicates that modem is not in ENDC connected mode.

Returns

5G NR dBm value.

4.3.1.43.2.3 const int telux::tel::Nr5gSignalStrengthInfo::getReferenceSignalReceiveQuality ( ) const

Get 5G NR reference signal receive quality in dB. (Valid value range [-20, -3] and INVALID_SIGNAL_STRENGTH_VALUE i.e. unavailable). INVALID_SIGNAL_STRENGTH_VALUE indicates that modem is not in ENDC connected mode.

Returns

5G NR rsrq.

4.3.1.43.2.4 const int telux::tel::Nr5gSignalStrengthInfo::getReferenceSignalSnr ( ) const

Get 5G NR reference signal signal-to-noise ratio, multiply by 0.1 to get SNR in dB. (Valid value range [-200, +300] and INVALID_SIGNAL_STRENGTH_VALUE i.e. unavailable). (-200 = -20.0 dB, +300 = 30dB). INVALID_SIGNAL_STRENGTH_VALUE indicates that modem is not in ENDC connected mode.

Returns

5G NR signal-to-noise.

4.3.1.44 class telux::tel::VoiceServiceInfo

VoiceServiceInfo is a container class for obtaining serving state details like phone is registered to home network, roaming, in service, out of service or only emergency calls allowed.

Public member functions

- VoiceServiceInfo (VoiceServiceState voiceServiceState, VoiceServiceDenialCause denialCause, RadioTechnology radioTech)
- VoiceServiceState getVoiceServiceState ()
- VoiceServiceDenialCause getVoiceServiceDenialCause ()
- bool isEmergency ()
- bool isInService ()
- bool isOutOfService ()
- RadioTechnology getRadioTechnology ()

4.3.1.44.1 Constructors and Destructors

4.3.1.44.1.1 telux::tel::VoiceServiceInfo::VoiceServiceInfo (VoiceServiceState *voiceServiceState,* VoiceServiceDenialCause *denialCause,* RadioTechnology *radioTech*)

4.3.1.44.2 Member Function Documentation

4.3.1.44.2.1 VoiceServiceState telux::tel::VoiceServiceInfo::getVoiceServiceState ( )

Get voice service state.

Returns

VoiceServiceState

4.3.1.44.2.2 VoiceServiceDenialCause telux::tel::VoiceServiceInfo::getVoiceServiceDenialCause ( )

Get Voice service denial cause

Returns

VoiceServiceDenialCause

4.3.1.44.2.3 bool telux::tel::VoiceServiceInfo::isEmergency ( )

Check if phone service is in emergency mode (i.e Only emergency numbers are allowed)

4.3.1.44.2.4 bool telux::tel::VoiceServiceInfo::isInService ( )

Check if phone is registered to home network or roaming network, phone is in service mode

4.3.1.44.2.5 bool telux::tel::VoiceServiceInfo::isOutOfService ( )

check if phone not registered, phone is in out of service mode

4.3.1.44.2.6 RadioTechnology telux::tel::VoiceServiceInfo::getRadioTechnology ( )

Get voice radio technology

Returns

RadioTechnology

4.3.2 Enumeration Type Documentation

4.3.2.1 enum telux::tel::CellType [strong]

Defines all the cell info types.

Enumerator

GSM
CDMA
LTE
WCDMA
TDSCDMA
NR5G

4.3.2.2 enum telux::tel::ECallVariant [strong]

ECall Variant

Enumerator

ECALL_TEST Initiate a test voice eCall with a configured telephone number stored in the USIM.
ECALL_EMERGENCY Initiate an emergency eCall. The trigger can be a manually initiated eCall or automatically initiated eCall.
ECALL_VOICE Initiate a regular voice call with capability to transfer an MSD.

4.3.2.3 enum telux::tel::EmergencyCallType [strong]

Emergency Call Type

Enumerator

CALL_TYPE_ECALL eCall (0x0C)

4.3.2.4 enum telux::tel::ECallMsdTransmissionStatus [strong]

MSD Transmission Status

Enumerator

SUCCESS In-band MSD transmission is successful
FAILURE In-band MSD transmission failed
MSD_TRANSMISSION_STARTED In-band MSD transmission started
NACK_OUT_OF_ORDER Out of order NACK message detected during in-band MSD transmission
ACK_OUT_OF_ORDER Out of order ACK message detected during in-band MSD transmission
START_RECEIVED SEND-MSD(START) is received and SYNC is locked during in-band MSD transmission
LL_ACK_RECEIVED Link-Layer Acknowledgement(LL-ACK) is received during in-band MSD transmission
OUTBAND_MSD_TRANSMISSION_STARTED Outband MSD transmission started in NG eCall
OUTBAND_MSD_TRANSMISSION_SUCCESS Outband MSD transmission succeeded in NG eCall or Third Party Service (TPS) eCall
OUTBAND_MSD_TRANSMISSION_FAILURE Outband MSD transmission failed in NG eCall or Third Party Service (TPS) eCall

4.3.2.5 enum telux::tel::ECallCategory [strong]

ECall category

Enumerator

VOICE_EMER_CAT_AUTO_ECALL Automatic emergency call
VOICE_EMER_CAT_MANUAL Manual emergency call

4.3.2.6 enum telux::tel::ECallVehicleType

Represents a vehicle class as per European eCall MSD standard. i.e. EN 15722:2020. Some of these values are only supported in certain MSD versions, so ensure to use supported values in an MSD. For example, TRAILERS_CLASS_O is not supported in MSD version-2 (as per A.1 in EN 15722:2015(E)), but supported in in MSD version-3 (as per A.1 in EN 15722:2020).

Enumerator

PASSENGER_VEHICLE_CLASS_M1
BUSES_AND_COACHES_CLASS_M2
BUSES_AND_COACHES_CLASS_M3
LIGHT_COMMERCIAL_VEHICLES_CLASS_N1
HEAVY_DUTY_VEHICLES_CLASS_N2

HEAVY_DUTY_VEHICLES_CLASS_N3
MOTOR_CYCLES_CLASS_L1E
MOTOR_CYCLES_CLASS_L2E
MOTOR_CYCLES_CLASS_L3E
MOTOR_CYCLES_CLASS_L4E
MOTOR_CYCLES_CLASS_L5E
MOTOR_CYCLES_CLASS_L6E
MOTOR_CYCLES_CLASS_L7E
TRAILERS_CLASS_O
AGRI_VEHICLES_CLASS_R
AGRI_VEHICLES_CLASS_S
AGRI_VEHICLES_CLASS_T
OFF_ROAD_VEHICLES_G
SPECIAL_PURPOSE_MOTOR_CARAVAN_CLASS_SA
SPECIAL_PURPOSE_ARMOURED_VEHICLE_CLASS_SB
SPECIAL_PURPOSE_AMBULANCE_CLASS_SC
SPECIAL_PURPOSE_HEARCE_CLASS_SD
OTHER_VEHICLE_CLASS

4.3.2.7 enum telux::tel::ECallOptionalDataType [strong]

Represents OptionalDataType class as per European eCall MSD standard. i.e. EN 15722.

Enumerator

ECALL_DEFAULT

4.3.2.8 enum telux::tel::ECallMode [strong]

Represents eCall operating mode

Enumerator

NORMAL eCall and normal voice calls are allowed
ECALL_ONLY Only eCall is allowed
NONE Invalid mode

4.3.2.9 enum telux::tel::ECallModeReason [strong]

Represents eCall operating mode change reason

Enumerator

NORMAL eCall operating mode changed due to normal operation like setting of eCall mode
ERA_GLONASS eCall operating mode changed due to ERA-GLONASS operation

4.3.2.10 enum telux::tel::HlapTimerStatus [strong]

Represents the status of an eCall High Level Application Protocol(HLAP) timer that is maintained by the UE state machine.

Enumerator

UNKNOWN Unknown
INACTIVE eCall Timer is Inactive i.e it has not started or it has stopped/expired
ACTIVE eCall Timer is Active i.e it has started but not yet stopped/expired

4.3.2.11 enum telux::tel::HlapTimerEvent [strong]

Represents an event causing a change in the the status of eCall High Level Application Protocol (HLAP) timer that is maintained by the UE state machine.

Timer STARTED notification is provided when the timer moves from INACTIVE to ACTIVE state. Timer STOPPED notification is provided when the timer moves from ACTIVE to INACTIVE state, after its underlying condition is satisfied. Timer EXPIRED notification is provided when the timer moves from ACTIVE to INACTIVE state, after its underlying condition not satisfied until its timeout.

Enumerator

UNKNOWN Unknown
UNCHANGED No change in timer status
STARTED eCall Timer is Started
STOPPED eCall Timer is Stopped
EXPIRED eCall Timer is expired

4.3.2.12 enum telux::tel::HlapTimerType [strong]

Represents the type of an eCall High Level Application Protocol(HLAP) timer that is maintained by the UE state machine. The timers are represented according to EN 16062:2015 standard.

Enumerator

UNKNOWN_TIMER eCall unknown timer
T2_TIMER eCall T2 timer
T5_TIMER eCall T5 timer
T6_TIMER eCall T6 timer
T7_TIMER eCall T7 timer
T9_TIMER eCall T9 timer
T10_TIMER eCall T10 timer

4.3.2.13 enum telux::tel::ECallNumType [strong]

Configuration that represents the type of the number to be dialed when an automotive emergency call is initiated.

Enumerator

DEFAULT
OVERRIDDEN

4.3.2.14 enum telux::tel::EcallConfigType

Defines the supported ECall configuration parameters

Enumerator

ECALL_CONFIG_MUTE_RX_AUDIO Mute the local audio device during MSD transmission
ECALL_CONFIG_NUM_TYPE Decides which number needs to be dialed when an eCall is initiated
ECALL_CONFIG_OVERRIDDEN_NUM User configured/overridden number that will be dialed for eCall
ECALL_CONFIG_USE_CANNED_MSD Use the pre-defined MSD in modem for eCall
ECALL_CONFIG_GNSS_UPDATE_INTERVAL Time interval in milliseconds, at which modem updates the GNSS information in its internally generated MSD
ECALL_CONFIG_T2_TIMER T2 timer value
ECALL_CONFIG_T7_TIMER T7 timer value
ECALL_CONFIG_T9_TIMER T9 timer value
ECALL_CONFIG_MSD_VERSION MSD version to be used by modem when it internally generates MSD i.e when MSD is not sent by application and also canned MSD is not used
ECALL_CONFIG_COUNT

4.3.2.15 enum telux::tel::RadioState [strong]

Defines the radio state

Enumerator

RADIO_STATE_OFF Radio is explicitly powered off
RADIO_STATE_UNAVAILABLE Radio unavailable (eg, resetting or not booted)
RADIO_STATE_ON Radio is on

4.3.2.16 enum telux::tel::ServiceState [strong]

Defines the service states

Deprecated

Use requestVoiceServiceState() API or to know the status of phone

Enumerator

EMERGENCY_ONLY Only emergency calls allowed
IN_SERVICE Normal operation, device is registered with a carrier and online
OUT_OF_SERVICE Device is not registered with any carrier
RADIO_OFF Device radio is off - Airplane mode for example

4.3.2.17 enum telux::tel::RadioTechnology [strong]

Defines all available radio access technologies

Enumerator

RADIO_TECH_UNKNOWN Network type is unknown
RADIO_TECH_GPRS Network type is GPRS
RADIO_TECH_EDGE Network type is EDGE
RADIO_TECH_UMTS Network type is UMTS
RADIO_TECH_IS95A Network type is IS95A
RADIO_TECH_IS95B Network type is IS95B
RADIO_TECH_1xRTT Network type is 1xRTT
RADIO_TECH_EVDO_0 Network type is EVDO revision 0
RADIO_TECH_EVDO_A Network type is EVDO revision A
RADIO_TECH_HSDPA Network type is HSDPA
RADIO_TECH_HSUPA Network type is HSUPA
RADIO_TECH_HSPA Network type is HSPA
RADIO_TECH_EVDO_B Network type is EVDO revision B
RADIO_TECH_EHRPD Network type is eHRPD
RADIO_TECH_LTE Network type is LTE
RADIO_TECH_HSPAP Network type is HSPA+
RADIO_TECH_GSM Network type is GSM, Only supports voice
RADIO_TECH_TD_SCDMA Network type is TD SCDMA
RADIO_TECH_IWLAN Network type is TD IWLAN
RADIO_TECH_LTE_CA Network type is LTE CA
RADIO_TECH_NR5G Network type is NR5G

4.3.2.18 enum telux::tel::RATCapability `[strong]`

Defines all available RAT capabilities for each subscription

Enumerator

AMPS
CDMA
HDR
GSM
WCDMA
LTE
TDS
NR5G NR5G NSA mode
NR5GSA NR5G SA mode

4.3.2.19 enum telux::tel::VoiceServiceTechnology `[strong]`

Defines all voice support available on device

Enumerator

VOICE_TECH_GW_CSFB
VOICE_TECH_1x_CSFB
VOICE_TECH_VOLTE

4.3.2.20 enum telux::tel::OperatingMode [strong]

Defines operating modes of the device.

Enumerator

ONLINE Online mode
AIRPLANE Low Power mode i.e temporarily disabled RF
FACTORY_TEST Special mode for manufacturer use
OFFLINE Device has deactivated RF and partially shutdown
RESETTING Device is in process of power cycling
SHUTTING_DOWN Device is in process of shutting down
PERSISTENT_LOW_POWER Persists low power mode even on reset

4.3.2.21 enum telux::tel::EcbMode [strong]

Emergency callback mode

Enumerator

NORMAL Device is not in emergency callback mode(ECBM)
EMERGENCY Device is in emergency callback mode(ECBM)

4.3.2.22 enum telux::tel::SignalStrengthLevel [strong]

Defines all the signal levels that SignalStrength class can return where level 1 is low and level 5 is high.

Enumerator

LEVEL_1
LEVEL_2
LEVEL_3
LEVEL_4
LEVEL_5
LEVEL_UNKNOWN

4.3.2.23 enum telux::tel::VoiceServiceState [strong]

Defines the voice service states

Enumerator

NOT_REG_AND_NOT_SEARCHING Not registered, MT is not currently searching a new operator to register
REG_HOME Registered, home network
NOT_REG_AND_SEARCHING Not registered, but MT is currently searching a new operator to register
REG_DENIED Registration denied
UNKNOWN Unknown
REG_ROAMING Registered, roaming
NOT_REG_AND_EMERGENCY_AVAILABLE_AND_NOT_SEARCHING Same as NOT_REG_AND_NOT_SEARCHING but indicates that emergency calls are enabled

NOT_REG_AND_EMERGENCY_AVAILABLE_AND_SEARCHING Same as NOT_REG_AND_SEARCHING but indicates that emergency calls are enabled

REG_DENIED_AND_EMERGENCY_AVAILABLE Same as REG_DENIED but indicates that emergency calls are enabled

UNKNOWN_AND_EMERGENCY_AVAILABLE Same as UNKNOWN but indicates that emergency calls are enabled

4.3.2.24 enum telux::tel::VoiceServiceDenialCause `[strong]`

Defines the voice service denial cause why voice service state registration was denied See 3GPP TS 24.008, 10.5.3.6 and Annex G.

Enumerator

UNDEFINED Undefined

GENERAL General

AUTH_FAILURE Authentication Failure

IMSI_UNKNOWN IMSI unknown in HLR

ILLEGAL_MS Illegal Mobile Station (MS), network refuses service to the MS either because an identity of the MS is not acceptable to the network or because the MS does not pass the authentication check

IMSI_UNKNOWN_VLR IMSI unknown in Visitors Location Register (VLR)

IMEI_NOT_ACCEPTED Network does not accept emergency call establishment using an IMEI or not accept attach procedure for emergency services using an IMEI

ILLEGAL_ME ME used is not acceptable to the network

GPRS_SERVICES_NOT_ALLOWED Not allowed to operate GPRS services.

GPRS_NON_GPRS_NOT_ALLOWED Not allowed to operate either GPRS or non-GPRS services

MS_IDENTITY_FAILED the network cannot derive the MS's identity from the P-TMSI/GUTI.

IMPLICITLY_DETACHED network has implicitly detached the MS

GPRS_NOT_ALLOWED_IN_PLMN GPRS services not allowed in this PLMN

MSC_TEMPORARILY_NOT_REACHABLE MSC temporarily not reachable

SMS_PROVIDED_VIA_GPRS SMS provided via GPRS in this routing area

NO_PDP_CONTEXT_ACTIVATED No PDP context activated

PLMN_NOT_ALLOWED if the network initiates a detach request or UE requests a services, in a PLMN where the MS, by subscription or due to operator determined barring is not allowed to operate.

LOCATION_AREA_NOT_ALLOWED network initiates a detach request, in a location area where the HPLMN determines that the MS, by subscription, is not allowed to operate or roaming subscriber the subscriber is denied service even if other PLMNs are available on which registration was possible

ROAMING_NOT_ALLOWED Roaming not allowed in this Location Area

NO_SUITABLE_CELLS No Suitable Cells in this Location Area

NOT_AUTHORIZED Not Authorized for this CSG

NETWORK_FAILURE Network Failure

MAC_FAILURE MAC failure

SYNC_FAILURE USIM detects that the SQN in the AUTHENTICATION REQUEST or AUTHENTICATION_AND_CIPHERING REQUEST message is out of range

CONGESTION network cannot serve a request from the MS because of congestion

GSM_AUTHENTICATION_UNACCEPTABLE GSM Authentication unacceptable

SERVICE_OPTION_NOT_SUPPORTED Service option not supported

SERVICE_OPTION_NOT_SUBCRIBED Requested service option not subscribed
SERVICE_OPTION_OUT_OF_ORDER Service option temporarily out of order
CALL_NOT_IDENTIFIED Call cannot be identified
RETRY_FOR_NEW_CELL Retry upon entry into a new cell
INCORRECT_MESSAGE Semantically incorrect message
INVALID_INFO Invalid mandatory information
MSG_TYPE_NOT_IMPLEMENTED Message type non-existent or not implemented
MSG_NOT_COMPATIBLE Message not compatible with protocol state
INFO_NOT_IMPLEMENTED Information element non-existent or not implemented
CONDITIONAL_IE_ERROR Conditional IE error
PROTOCOL_ERROR_UNSPECIFIED Protocol error, unspecified

4.3.3 Variable Documentation

4.3.3.1 const std::string telux::tel::CONTENT_HEADER = "application/Emergency↩ CallData.eCall.MSD" [static]

Default value for CustomSipHeader::contentType

4.4 Call

This section contains APIs related to Call.

4.4.1 Data Structure Documentation

4.4.1.1 class telux::tel::ICall

ICall represents a call in progress. An ICall cannot be directly created by the client, rather it is returned as a result of instantiating a call or from the PhoneListener when receiving an incoming call.

Public member functions

- virtual telux::common::Status answer (std::shared_ptr< telux::common::ICommandResponseCallback > callback=nullptr)=0
- virtual telux::common::Status hold (std::shared_ptr< telux::common::ICommandResponseCallback > callback=nullptr)=0
- virtual telux::common::Status resume (std::shared_ptr< telux::common::ICommandResponseCallback > callback=nullptr)=0
- virtual telux::common::Status reject (std::shared_ptr< telux::common::ICommandResponseCallback > callback=nullptr)=0
- virtual telux::common::Status reject (const std::string &rejectSMS, std::shared_ptr< telux::common::ICommandResponseCallback > callback=nullptr)=0
- virtual telux::common::Status hangup (std::shared_ptr< telux::common::ICommandResponseCallback > callback=nullptr)=0
- virtual telux::common::Status playDtmfTone (char tone, std::shared_ptr< telux::common::ICommandResponseCallback > callback=nullptr)=0
- virtual telux::common::Status startDtmfTone (char tone, std::shared_ptr< telux::common::ICommandResponseCallback > callback=nullptr)=0
- virtual telux::common::Status stopDtmfTone (std::shared_ptr< telux::common::ICommandResponseCallback > callback=nullptr)=0
- virtual CallState getCallState ()=0
- virtual int getCallIndex ()=0
- virtual CallDirection getCallDirection ()=0
- virtual std::string getRemotePartyNumber ()=0
- virtual CallEndCause getCallEndCause ()=0
- virtual int getPhoneId ()=0
- virtual bool isMultiPartyCall ()=0
- virtual ~ICall ()

4.4.1.1.1 Constructors and Destructors

4.4.1.1.1.1 virtual telux::tel::ICall::~ICall ( ) [virtual]

4.4.1.1.2 Member Function Documentation

4.4.1.1.2.1 virtual telux::common::Status telux::tel::ICall::answer (std::shared_ptr< telux::common::ICommandResponseCallback > *callback = nullptr*) [pure virtual]

Allows the client to answer the call. This is only applicable for CallState::INCOMING and CallState::WAITING calls. If a Waiting call is being answered and the existing call is Active, then existing call will move to Hold state.If the existing call is on Hold already, then it will remain on Hold. The waiting call state transition from Waiting to Active.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_CALL_MGMT permission to invoke this API successfully.

Parameters

in	*callback*	- optional callback pointer to get the response of answer request below are possible error codes for callback response • telux::common::ErrorCode::SUCCESS • telux::common::ErrorCode::RADIO_NOT_AVAILABLE • telux::common::ErrorCode::NO_MEMORY • telux::common::ErrorCode::MODEM_ERR • telux::common::ErrorCode::INTERNAL_ERR • telux::common::ErrorCode::INVALID_STATE • telux::common::ErrorCode::INVALID_CALL_ID • telux::common::ErrorCode::INVALID_ARGUMENTS • telux::common::ErrorCode::OPERATION_NOT_ALLOWED • telux::common::ErrorCode::GENERIC_FAILURE

Returns

Status of hold function i.e. success or suitable error code.

4.4.1.1.2.2 virtual telux::common::Status telux::tel::ICall::hold (std::shared_ptr< telux::common::ICommandResponseCallback > *callback = nullptr*) [pure virtual]

Puts the ongoing call on hold.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_CALL_MGMT permission to invoke this API successfully.

Parameters

in	*callback*	- optional callback pointer to get the response of hold request below are possible error codes for callback response • telux::common::ErrorCode::SUCCESS • telux::common::ErrorCode::RADIO_NOT_AVAILABLE • telux::common::ErrorCode::NO_MEMORY • telux::common::ErrorCode::MODEM_ERR • telux::common::ErrorCode::INTERNAL_ERR • telux::common::ErrorCode::INVALID_STATE • telux::common::ErrorCode::INVALID_CALL_ID • telux::common::ErrorCode::INVALID_ARGUMENTS • telux::common::ErrorCode::OPERATION_NOT_ALLOWED • telux::common::ErrorCode::GENERIC_FAILURE

Returns

Status of hold function i.e. success or suitable error code.

4.4.1.1.2.3 virtual telux::common::Status telux::tel::ICall::resume (std::shared_ptr< telux::common::ICommandResponseCallback > *callback = nullptr*) [pure virtual]

Resumes this call from on-hold state to active state

On platforms with Access control enabled, Caller needs to have TELUX_TEL_CALL_MGMT permission to invoke this API successfully.

Parameters

in	*callback*	- optional callback pointer to get the response of resume request below are possible error codes for callback response • telux::common::ErrorCode::SUCCESS • telux::common::ErrorCode::RADIO_NOT_AVAILABLE • telux::common::ErrorCode::NO_MEMORY • telux::common::ErrorCode::MODEM_ERR • telux::common::ErrorCode::INTERNAL_ERR • telux::common::ErrorCode::INVALID_STATE • telux::common::ErrorCode::INVALID_CALL_ID • telux::common::ErrorCode::INVALID_ARGUMENTS • telux::common::ErrorCode::OPERATION_NOT_ALLOWED • telux::common::ErrorCode::GENERIC_FAILURE

Returns

Status of resume function i.e. success or suitable error code.

4.4.1.1.2.4 virtual telux::common::Status telux::tel::ICall::reject (std::shared_ptr< telux::common::ICommandResponseCallback > *callback = nullptr*) [pure virtual]

Reject the incoming/waiting call. Only applicable for CallState::INCOMING and CallState::WAITING calls.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_CALL_MGMT permission to invoke this API successfully.

Parameters

in	*callback*	- optional callback pointer to get the response of reject request below are possible error codes for callback response • telux::common::ErrorCode::SUCCESS • telux::common::ErrorCode::RADIO_NOT_AVAILABLE • telux::common::ErrorCode::NO_MEMORY • telux::common::ErrorCode::MODEM_ERR • telux::common::ErrorCode::INTERNAL_ERR • telux::common::ErrorCode::INVALID_STATE • telux::common::ErrorCode::INVALID_CALL_ID • telux::common::ErrorCode::INVALID_ARGUMENTS • telux::common::ErrorCode::OPERATION_NOT_ALLOWED • telux::common::ErrorCode::GENERIC_FAILURE

Returns

Status of reject function i.e. success or suitable error code.

4.4.1.1.2.5 virtual telux::common::Status telux::tel::ICall::reject (const std::string & *rejectSMS,* std::shared_ptr< telux::common::ICommandResponseCallback > *callback = nullptr*) [pure virtual]

Reject the call and send an SMS to caller. Only applicable for CallState::INCOMING and CallState::WAITING calls.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_CALL_MGMT permission to invoke this API successfully.

Parameters

in	*rejectSMS*	SMS string used to send in response to a call rejection.
in	*callback*	- optional callback pointer to get the response of rejectwithSMS request below are possible error codes for callback response • telux::common::ErrorCode::SUCCESS • telux::common::ErrorCode::RADIO_NOT_AVAILABLE • telux::common::ErrorCode::NO_MEMORY • telux::common::ErrorCode::MODEM_ERR • telux::common::ErrorCode::INTERNAL_ERR • telux::common::ErrorCode::INVALID_STATE • telux::common::ErrorCode::INVALID_CALL_ID • telux::common::ErrorCode::INVALID_ARGUMENTS • telux::common::ErrorCode::OPERATION_NOT_ALLOWED • telux::common::ErrorCode::GENERIC_FAILURE

Deprecated

This API not being supported

Returns

Status of success for call reject() or suitable error code.

4.4.1.1.2.6 virtual telux::common::Status telux::tel::ICall::hangup (std::shared_ptr< telux::common::ICommandResponseCallback > *callback* = `nullptr`) [pure virtual]

Hangup the call if the call state is either active, hold, dialing, waiting or alerting.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_CALL_MGMT permission to invoke this API successfully.

Parameters

in	*callback*	- optional callback pointer to get the response of hangup request below are possible error codes for callback response • telux::common::ErrorCode::SUCCESS • telux::common::ErrorCode::RADIO_NOT_AVAILABLE • telux::common::ErrorCode::NO_MEMORY • telux::common::ErrorCode::MODEM_ERR • telux::common::ErrorCode::INTERNAL_ERR • telux::common::ErrorCode::INVALID_STATE • telux::common::ErrorCode::INVALID_CALL_ID • telux::common::ErrorCode::INVALID_ARGUMENTS • telux::common::ErrorCode::OPERATION_NOT_ALLOWED • telux::common::ErrorCode::GENERIC_FAILURE

Returns

Status of hangup i.e. success or suitable error code.

4.4.1.1.2.7 virtual telux::common::Status telux::tel::ICall::playDtmfTone (char *tone,* std::shared_ptr< telux::common::ICommandResponseCallback > *callback = nullptr*) [pure virtual]

Play a DTMF tone and stop it. The interval for which the tone is played is dependent on the system implementation. If continuous DTMF tone is playing, it will be stopped. This API is used to play DTMF tone on TX path so that it is heard on far end. For DTMF playback on local device on the RX path use telux::audio::IAudioVoiceStream::playDtmfTone

On platforms with Access control enabled, Caller needs to have TELUX_TEL_CALL_MGMT permission to invoke this API successfully.

Parameters

in	*tone*	- a single character with one of 12 values: 0-9, ∗, #.
in	*callback*	- Optional callback pointer to get the result of playDtmfTones function

Returns

Status of playDtmfTones i.e. success or suitable error code.

4.4.1.1.2.8 virtual telux::common::Status telux::tel::ICall::startDtmfTone (char *tone,* std::shared_ptr< telux::common::ICommandResponseCallback > *callback = nullptr*) [pure virtual]

Starts a continuous DTMF tone. To terminate the continous DTMF tone,stopDtmfTone API needs to be invoked explicitly. This API is used to play DTMF tone on TX path so that it is heard on far end. For DTMF playback on local device on the RX path use telux::audio::IAudioVoiceStream::playDtmfTone

On platforms with Access control enabled, Caller needs to have TELUX_TEL_CALL_MGMT permission to invoke this API successfully.

Parameters

in	*tone*	- a single character with one of 12 values: 0-9, ∗, #.
in	*callback*	- Optional callback pointer to get the result of startDtmfTone function.

Returns

Status of startDtmfTone i.e. success or suitable error code.

4.4.1.1.2.9 virtual telux::common::Status telux::tel::ICall::stopDtmfTone (std::shared_ptr< telux::common::ICommandResponseCallback > *callback = nullptr*) [pure virtual]

Stop the currently playing continuous DTMF tone.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_CALL_MGMT permission to invoke this API successfully.

Parameters

in	*callback*	- Optional callback pointer to get the result of stopDtmfTone function.

Returns

Status of stopDtmfTone i.e. success or suitable error code.

4.4.1.1.2.10 virtual CallState telux::tel::ICall::getCallState ( ) [pure virtual]

Get the current state of the call, such as ringing, in progress etc.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_CALL_INFO_READ permission to invoke this API successfully.

Returns

CallState - enumeration representing call State

4.4.1.1.2.11 virtual int telux::tel::ICall::getCallIndex ( ) [pure virtual]

Get the unique index of the call assigned by Telephony subsystem

On platforms with Access control enabled, Caller needs to have TELUX_TEL_CALL_MGMT permission to invoke this API successfully.

Returns

Call Index

4.4.1.1.2.12 virtual CallDirection telux::tel::ICall::getCallDirection ( ) [pure virtual]

Get the direction of the call

On platforms with Access control enabled, Caller needs to have TELUX_TEL_CALL_MGMT permission to invoke this API successfully.

Returns

CallDirection - enumeration representing call direction i.e. INCOMING/ OUTGOING

4.4.1.1.2.13 virtual std::string telux::tel::ICall::getRemotePartyNumber ( ) [pure virtual]

Get the dailing number

On platforms with Access control enabled, Caller needs to have TELUX_TEL_CALL_PRIVATE_INFO permission to invoke this API successfully.

Returns

Phone Number to which the call was dialed out. Empty string in case of INCOMING call direction.

4.4.1.1.2.14 virtual CallEndCause telux::tel::ICall::getCallEndCause ( ) [pure virtual]

Get the cause of the termination of the call.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_CALL_INFO_READ permission to invoke this API successfully.

Returns

Enum representing call end cause.

4.4.1.1.2.15 virtual int telux::tel::ICall::getPhoneId ( ) [pure virtual]

Get id of the phone object which represents the network/SIM on which the call is in progress.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_CALL_INFO_READ permission to invoke this API successfully.

Returns

Phone Id.

4.4.1.1.2.16 virtual bool telux::tel::ICall::isMultiPartyCall ( ) [pure virtual]

To check if call is in multi party call(conference) or not

On platforms with Access control enabled, Caller needs to have TELUX_TEL_CALL_INFO_READ permission to invoke this API successfully.

Returns

True if call is in conference otherwise false.

4.4.1.2 class telux::tel::ICallListener

A listener class for monitoring changes in call, including call state change and ECall state change. Override the methods for the state that you wish to receive updates for.

The methods in listener can be invoked from multiple different threads. The implementation should be thread safe.

Public member functions

- virtual void onIncomingCall (std::shared_ptr< ICall > call)
- virtual void onCallInfoChange (std::shared_ptr< ICall > call)
- virtual void onECallMsdTransmissionStatus (int phoneId, telux::common::ErrorCode errorCode)
- virtual void onECallMsdTransmissionStatus (int phoneId, telux::tel::ECallMsdTransmissionStatus msdTransmissionStatus)
- virtual void OnTpsMsdUpdateRequest (int phoneId)
- virtual void onECallHlapTimerEvent (int phoneId, ECallHlapTimerEvents timersStatus)
- virtual void onEmergencyNetworkScanFail (int phoneId)
- virtual void onEcbmChange (telux::tel::EcbMode mode)
- virtual ~ICallListener ()

4.4.1.2.1 Constructors and Destructors

4.4.1.2.1.1 virtual telux::tel::ICallListener::~ICallListener ( ) [virtual]

4.4.1.2.2 Member Function Documentation

4.4.1.2.2.1 virtual void telux::tel::ICallListener::onIncomingCall (std::shared_ptr< ICall > *call*) [virtual]

This function is called when device receives an incoming/waiting call.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_CALL_INFO_READ permission to receive this notification.

Parameters

in	*call*	- Pointer to ICall instance

4.4.1.2.2.2 virtual void telux::tel::ICallListener::onCallInfoChange (std::shared_ptr< ICall > *call*) [virtual]

This function is called when there is a change in call attributes

On platforms with Access control enabled, Caller needs to have TELUX_TEL_CALL_INFO_READ permission to receive this notification.

Parameters

in	*call*	- Pointer to ICall instance

4.4.1.2.2.3 virtual void telux::tel::ICallListener::onECallMsdTransmissionStatus (int *phoneId*, telux::common::ErrorCode *errorCode*) [virtual]

This function is called when device completes MSD Transmission.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_ECALL_MGMT permission to receive this notification.

Parameters

in	*phoneId*	- Unique Id of phone on which MSD Transmission Status is being reported
in	*errorCode*	- Indicates MSD Transmission status i.e. success or failure

Deprecated

Use another onECallMsdTransmissionStatus() API with argument ECallMsdTransmissionStatus

4.4.1.2.2.4 virtual void telux::tel::ICallListener::onECallMsdTransmissionStatus (int *phoneId*, telux::tel::ECallMsdTransmissionStatus *msdTransmissionStatus*) [virtual]

This function is called when there is Minimum Set of Data (MSD) transmission. The MSD transmission happens at call connect and also when the modem or client responds to MSD pull request from PSAP.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_ECALL_MGMT permission to receive this notification.

Parameters

in	*phoneId*	- Unique Id of phone on which MSD Transmission Status is being reported
in	*msdTransmission↩ Status*	- Indicates MSD Transmission status ECallMsdTransmissionStatus

4.4.1.2.2.5 virtual void telux::tel::ICallListener::OnTpsMsdUpdateRequest (int *phoneId*) [virtual]

This function is called when MSD update is requested by PSAP during Third Party Service (TPS) ecall over IMS.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_ECALL_MGMT permission to receive this notification.

Parameters

in	*phoneId*	- Unique Id of phone on which MSD update request is received.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.4.1.2.2.6 virtual void telux::tel::ICallListener::onECallHlapTimerEvent (int *phoneId,* ECallHlap↩ TimerEvents *timersStatus*) [virtual]

This function is called when the eCall High Level Application Protocol(HLAP) timers status is changed.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_ECALL_MGMT permission to receive this notification.

Parameters

in	*phoneId*	- Unique Id of phone on which HLAP timer status is being reported
in	*timersStatus*	- Indicates the HLAP timer event ECallHlapTimerEvents

4.4.1.2.2.7 virtual void telux::tel::ICallListener::onEmergencyNetworkScanFail (int *phoneId*) [virtual]

This function is called whenever there is a scan failure after one round of network scan during origination of emergency call or at any time during the emergency call.

During origination of an ecall or in between an ongoing ecall, if the UE is in an area of no/poor coverage and loses service, the modem will perform network scan and try to register on any available network. If the scan completes successfully and the device finds a suitable cell, the ecall will be placed and the call state changes to the active state. If the network scan fails then this function will be invoked after one round of network scan.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_ECALL_MGMT permission to receive this notification.

Parameters

in	*phoneId*	- Unique Id of phone on which network scan failure reported.

4.4.1.2.2.8 virtual void telux::tel::ICallListener::onEcbmChange (telux::tel::EcbMode *mode*) [virtual]

This function is called whenever emergency callback mode(ECBM) changes.

Parameters

in	*mode*	- Indicates the status of the ECBM. EcbMode

4.4.1.3 class telux::tel::ICallManager

Call Manager is the primary interface for call related operations Allows to conference calls, swap calls, make normal voice call and emergency call, send and update MSD pdu.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual telux::common::Status makeCall (int phoneId, const std::string &dialNumber, std::shared_ptr< IMakeCallCallback > callback=nullptr)=0
- virtual telux::common::Status makeECall (int phoneId, const ECallMsdData &eCallMsdData, int category, int variant, std::shared_ptr< IMakeCallCallback > callback=nullptr)=0
- virtual telux::common::Status makeECall (int phoneId, const std::string dialNumber, const ECallMsdData &eCallMsdData, int category, std::shared_ptr< IMakeCallCallback > callback=nullptr)=0
- virtual telux::common::Status makeECall (int phoneId, const std::string dialNumber, const std::vector< uint8_t > &msdPdu, CustomSipHeader header={telux::tel::CONTENT_HEADER,""}, MakeCallCallback callback=nullptr)=0
- virtual telux::common::Status makeECall (int phoneId, const std::vector< uint8_t > &msdPdu, int category, int variant, MakeCallCallback callback=nullptr)=0
- virtual telux::common::Status makeECall (int phoneId, const std::string dialNumber, const std::vector< uint8_t > &msdPdu, int category, MakeCallCallback callback=nullptr)=0
- virtual telux::common::Status makeECall (int phoneId, int category, int variant, MakeCallCallback callback=nullptr)=0
- virtual telux::common::Status makeECall (int phoneId, const std::string dialNumber, int category, MakeCallCallback callback=nullptr)=0
- virtual telux::common::Status updateECallMsd (int phoneId, const ECallMsdData &eCallMsd, std::shared_ptr< telux::common::ICommandResponseCallback > callback=nullptr)=0
- virtual telux::common::Status updateECallMsd (int phoneId, const std::vector< uint8_t > &msdPdu, telux::common::ResponseCallback callback)=0
- virtual telux::common::Status requestECallHlapTimerStatus (int phoneId, ECallHlapTimerStatusCallback callback)=0
- virtual std::vector< std::shared_ptr< ICall > > getInProgressCalls ()=0
- virtual telux::common::Status conference (std::shared_ptr< ICall > call1, std::shared_ptr< ICall > call2, std::shared_ptr< telux::common::ICommandResponseCallback > callback=nullptr)=0
- virtual telux::common::Status swap (std::shared_ptr< ICall > callToHold, std::shared_ptr< ICall > callToActivate, std::shared_ptr< telux::common::ICommandResponseCallback > callback=nullptr)=0
- virtual telux::common::Status hangupForegroundResumeBackground (int phoneId, common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status hangupWaitingOrBackground (int phoneId, common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestEcbm (int phoneId, EcbmStatusCallback callback)=0
- virtual telux::common::Status exitEcbm (int phoneId, common::ResponseCallback callback=nullptr)=0

- virtual telux::common::Status requestNetworkDeregistration (int phoneId, common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status updateEcallHlapTimer (int phoneId, HlapTimerType type, uint32_t timeDuration, common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestEcallHlapTimer (int phoneId, HlapTimerType type, ECallHlapTimerCallback callback)=0
- virtual telux::common::Status setECallConfig (EcallConfig config)=0
- virtual telux::common::Status getECallConfig (EcallConfig &config)=0
- virtual telux::common::Status registerListener (std::shared_ptr< telux::tel::ICallListener > listener)=0
- virtual telux::common::Status removeListener (std::shared_ptr< telux::tel::ICallListener > listener)=0
- virtual ~ICallManager ()

4.4.1.3.1 Constructors and Destructors

4.4.1.3.1.1 virtual telux::tel::ICallManager::~ICallManager ( ) [virtual]

4.4.1.3.2 Member Function Documentation

4.4.1.3.2.1 virtual telux::common::ServiceStatus telux::tel::ICallManager::getServiceStatus ( ) [pure virtual]

This status indicates whether the ICallManager object is in a usable state.

Returns

SERVICE_AVAILABLE - If CallManager is ready for service. SERVICE_UNAVAILABLE - If CallManager is temporarily unavailable. SERVICE_FAILED - If CallManager encountered an irrecoverable failure.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.4.1.3.2.2 virtual telux::common::Status telux::tel::ICallManager::makeCall (int *phoneId,* const std::string & *dialNumber,* std::shared_ptr< IMakeCallCallback > *callback = nullptr*) [pure virtual]

Initiate a voice call. This API can also be used for e911/e112 type of regular emergency call. This is not meant for an automotive eCall. Regular voice call will be blocked by device while eCall is in progress.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_CALL_MGMT permission to invoke this API successfully.

Parameters

in	*phoneId*	Represents phone corresponding to which on make call operation is performed
in	*dialNumber*	String representing the dialing number
in	*callback*	Optional callback pointer to get the response of makeCall request. Possible(not exhaustive) error codes for callback response • telux::common::ErrorCode::SUCCESS • telux::common::ErrorCode::RADIO_NOT_AVAILABLE • telux::common::ErrorCode::DIAL_MODIFIED_TO_USSD • telux::common::ErrorCode::DIAL_MODIFIED_TO_SS • telux::common::ErrorCode::DIAL_MODIFIED_TO_DIAL • telux::common::ErrorCode::INVALID_ARGUMENTS • telux::common::ErrorCode::NO_MEMORY • telux::common::ErrorCode::INVALID_STATE • telux::common::ErrorCode::NO_RESOURCES • telux::common::ErrorCode::INTERNAL_ERR • telux::common::ErrorCode::FDN_CHECK_FAILURE • telux::common::ErrorCode::MODEM_ERR • telux::common::ErrorCode::NO_SUBSCRIPTION • telux::common::ErrorCode::NO_NETWORK_FOUND • telux::common::ErrorCode::INVALID_CALL_ID • telux::common::ErrorCode::DEVICE_IN_USE • telux::common::ErrorCode::MODE_NOT_SUPPORTED • telux::common::ErrorCode::ABORTED • telux::common::ErrorCode::GENERIC_FAILURE

Returns

Status of makeCall i.e. success or suitable status code.

4.4.1.3.2.3 virtual telux::common::Status telux::tel::ICallManager::makeECall (int *phoneId,* const E↩ CallMsdData & *eCallMsdData,* int *category,* int *variant,* std::shared_ptr< IMakeCallCallback > *callback = nullptr*) [pure virtual]

Initiate an automotive eCall. Regular voice calls will be blocked by device while eCall is in progress.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_ECALL_MGMT permission to invoke this API successfully.

Parameters

in	*phoneId*	Represents phone corresponding to which make eCall operation is performed
in	*eCallMsdData*	The structure containing required fields to create eCall Minimum Set of Data (MSD)
in	*category*	ECallCategory
in	*variant*	ECallVariant

in	*callback*	Optional callback pointer to get the response of makeECall request. Possible(not exhaustive) error codes for callback response • telux::common::ErrorCode::SUCCESS • telux::common::ErrorCode::RADIO_NOT_AVAILABLE • telux::common::ErrorCode::NO_MEMORY • telux::common::ErrorCode::MODEM_ERR • telux::common::ErrorCode::INTERNAL_ERR • telux::common::ErrorCode::INVALID_STATE • telux::common::ErrorCode::INVALID_CALL_ID • telux::common::ErrorCode::INVALID_ARGUMENTS • telux::common::ErrorCode::OPERATION_NOT_ALLOWED • telux::common::ErrorCode::GENERIC_FAILURE

Returns

Status of makeECall i.e. success or suitable status code.

4.4.1.3.2.4 virtual telux::common::Status telux::tel::ICallManager::makeECall (int *phoneId*, const std::string *dialNumber*, const ECallMsdData & *eCallMsdData*, int *category*, std::shared_ptr< IMakeCallCallback > *callback* = `nullptr`) [pure virtual]

Initiate an automotive Third Party Service(TPS) eCall over CS technologies only (i.e. not IMS) to the specified phone number with Minimum Set of Data(MSD) at call connect. It will be treated like a regular voice call by the UE and the network.

It is the responsibility of application to make sure that another call is not dialed while Third Party Service eCall is in progress.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_ECALL_MGMT permission to invoke this API successfully.

Parameters

in	*phoneId*	Represents phone corresponding to which make eCall operation is performed
in	*dialNumber*	String representing the dialing number
in	*eCallMsdData*	The structure containing required fields to create eCall Minimum Set of Data (MSD)
in	*category*	ECallCategory
in	*callback*	Optional callback pointer to get the response of makeECall request. Possible(not exhaustive) error codes for callback response • telux::common::ErrorCode::SUCCESS • telux::common::ErrorCode::RADIO_NOT_AVAILABLE • telux::common::ErrorCode::NO_MEMORY • telux::common::ErrorCode::MODEM_ERR • telux::common::ErrorCode::INTERNAL_ERR • telux::common::ErrorCode::INVALID_STATE • telux::common::ErrorCode::INVALID_CALL_ID • telux::common::ErrorCode::INVALID_ARGUMENTS • telux::common::ErrorCode::OPERATION_NOT_ALLOW↩ ED • telux::common::ErrorCode::GENERIC_FAILURE

Returns

Status of makeECall i.e. success or suitable status code.

4.4.1.3.2.5 virtual telux::common::Status telux::tel::ICallManager::makeECall (int *phoneId*, const std::string *dialNumber*, const std::vector< uint8_t > & *msdPdu*, CustomSipHeader *header = {*telux::tel::CONTENT_HEADER*, ""}*, MakeCallCallback *callback = nullptr*) [pure virtual]

Initiate an automotive Third Party Service(TPS) eCall over IMS to the specified phone number with Minimum Set of Data(MSD) at call connect. It will be treated like a regular voice call over IMS by the UE and the network.

Application is expected to dial only one Third Party Service eCall per subscription. It is the responsibility of application to make sure that another call is not dialed while Third Party Service eCall is in progress.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_ECALL_MGMT permissionto invoke this API successfully.

Parameters

in	*phoneId*	Represents phone corresponding to which make eCall operation is performed
in	*dialNumber*	String representing the dialing number
in	*msdPdu*	Encoded MSD(Minimum Set of Data) PDU as per spec EN 15722 2015 or GOST R 54620-2011/33464-2015 Max size 255 bytes

in	*header*	Optional SIP headers intended to be sent in the SIP invite message to the network for PSAP • telux::tel::CustomSipHeader
in	*callback*	Optional callback function to get the response of makeECall request.

Returns

Status of makeECall i.e. success or suitable status code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.4.1.3.2.6 virtual telux::common::Status telux::tel::ICallManager::makeECall (int *phoneId,* const std::vector< uint8_t > & *msdPdu,* int *category,* int *variant,* MakeCallCallback *callback = nullptr*) [pure virtual]

Initiate an automotive eCall with raw MSD pdu. Regular voice calls will be blocked by device while eCall is in progress.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_ECALL_MGMT permission to invoke this API successfully.

Parameters

in	*phoneId*	Represents phone corresponding to which on make eCall operation is performed
in	*msdPdu*	Encoded MSD(Minimum Set of Data) PDU as per spec EN 15722 2015 or GOST R 54620-2011/33464-2015
in	*category*	ECallCategory
in	*variant*	ECallVariant
in	*callback*	Callback function to get the response of makeECall request. Possible(not exhaustive) error codes for callback response • telux::common::ErrorCode::SUCCESS • telux::common::ErrorCode::RADIO_NOT_AVAILABLE • telux::common::ErrorCode::NO_MEMORY • telux::common::ErrorCode::MODEM_ERR • telux::common::ErrorCode::INTERNAL_ERR • telux::common::ErrorCode::INVALID_STATE • telux::common::ErrorCode::INVALID_CALL_ID • telux::common::ErrorCode::INVALID_ARGUMENTS • telux::common::ErrorCode::OPERATION_NOT_ALLOW↩ ED • telux::common::ErrorCode::GENERIC_FAILURE

Returns

Status of makeECall i.e. success or suitable status code.

4.4.1.3.2.7 virtual telux::common::Status telux::tel::ICallManager::makeECall (int *phoneId*, const std::string *dialNumber*, const std::vector< uint8_t > & *msdPdu*, int *category*, MakeCallCallback *callback* = `nullptr`) [pure virtual]

Initiate an automotive eCall with raw MSD pdu, to the specified phone number for TPS eCall over CS Technologies only (i.e. not IMS). It will be treated like a regular voice call by the UE and the network.

It is the responsibility of application to make sure that another call is not dialed while Third Party Service eCall is in progress.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_ECALL_MGMT permission to invoke this API successfully.

Parameters

in	*phoneId*	Represents phone corresponding to which on make eCall operation is performed
in	*dialNumber*	String representing the dialing number
in	*msdPdu*	Encoded MSD(Minimum Set of Data) PDU as per spec EN 15722 2015 or GOST R 54620-2011/33464-2015
in	*category*	ECallCategory
in	*callback*	Callback function to get the response of makeECall request. Possible(not exhaustive) error codes for callback response • telux::common::ErrorCode::SUCCESS • telux::common::ErrorCode::RADIO_NOT_AVAILABLE • telux::common::ErrorCode::NO_MEMORY • telux::common::ErrorCode::MODEM_ERR • telux::common::ErrorCode::INTERNAL_ERR • telux::common::ErrorCode::INVALID_STATE • telux::common::ErrorCode::INVALID_CALL_ID • telux::common::ErrorCode::INVALID_ARGUMENTS • telux::common::ErrorCode::OPERATION_NOT_ALLOWED • telux::common::ErrorCode::GENERIC_FAILURE

Returns

Status of makeECall i.e. success or suitable status code.

4.4.1.3.2.8 virtual telux::common::Status telux::tel::ICallManager::makeECall (int *phoneId*, int *category*, int *variant*, MakeCallCallback *callback* = `nullptr`) [pure virtual]

Initiate an automotive eCall without transmitting Minimum Set of Data (MSD) at call connect. Regular voice calls will be blocked by device while eCall is in progress.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_ECALL_MGMT

permission to invoke this API successfully.

Parameters

in	*phoneId*	Represents phone corresponding to which make eCall operation is performed
in	*category*	ECallCategory
in	*variant*	ECallVariant
in	*callback*	Optional callback function to get the response of makeECall request. Possible(not exhaustive) error codes for callback response • telux::common::ErrorCode::SUCCESS • telux::common::ErrorCode::RADIO_NOT_AVAILABLE • telux::common::ErrorCode::NO_MEMORY • telux::common::ErrorCode::MODEM_ERR • telux::common::ErrorCode::INTERNAL_ERR • telux::common::ErrorCode::INVALID_STATE • telux::common::ErrorCode::INVALID_CALL_ID • telux::common::ErrorCode::INVALID_ARGUMENTS • telux::common::ErrorCode::OPERATION_NOT_ALLOWED • telux::common::ErrorCode::GENERIC_FAILURE

Returns

Status of makeECall i.e. success or suitable status code.

4.4.1.3.2.9 virtual telux::common::Status telux::tel::ICallManager::makeECall (int *phoneId,* const std::string *dialNumber,* int *category,* MakeCallCallback *callback* = `nullptr`) [pure virtual]

Initiate an automotive eCall to the specified phone number for TPS eCall over CS technologies only (i.e. not IMS), without transmitting Minimum Set of Data(MSD) at call connect. It will be treated like a regular voice call by the UE and the network.

It is the responsibility of application to make sure that another call is not dialed while Third Party Service eCall is in progress.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_ECALL_MGMT permission to invoke this API successfully.

Parameters

in	*phoneId*	Represents phone corresponding to which make eCall operation is performed
in	*dialNumber*	String representing the dialing number
in	*category*	ECallCategory
in	*callback*	Optional callback function to get the response of makeECall request. Possible(not exhaustive) error codes for callback response • telux::common::ErrorCode::SUCCESS • telux::common::ErrorCode::RADIO_NOT_AVAILABLE • telux::common::ErrorCode::NO_MEMORY • telux::common::ErrorCode::MODEM_ERR • telux::common::ErrorCode::INTERNAL_ERR • telux::common::ErrorCode::INVALID_STATE • telux::common::ErrorCode::INVALID_CALL_ID • telux::common::ErrorCode::INVALID_ARGUMENTS • telux::common::ErrorCode::OPERATION_NOT_ALLOWED • telux::common::ErrorCode::GENERIC_FAILURE

Returns

Status of makeECall i.e. success or suitable status code.

4.4.1.3.2.10 virtual telux::common::Status telux::tel::ICallManager::updateECallMsd (int *phoneId*, const ECallMsdData & *eCallMsd*, std::shared_ptr< telux::common::ICommandResponseCallback > *callback* = `nullptr`) [pure virtual]

Update the eCall MSD in modem to be sent to Public Safety Answering Point (PSAP) when requested.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_ECALL_MGMT permission to invoke this API successfully.

Parameters

in	*phoneId*	Represents phone corresponding to which updateECallMsd operation is performed
in	*eCallMsd*	The data structure represents the Minimum Set of Data (MSD)
in	*callback*	Optional callback pointer to get the response of updateECallMsd.

Returns

Status of updateECallMsd i.e. success or suitable error code.

4.4.1.3.2.11 virtual telux::common::Status telux::tel::ICallManager::updateECallMsd (int *phoneId,* const std::vector< uint8_t > & *msdPdu,* telux::common::ResponseCallback *callback*) [pure virtual]

For Third Party Service(TPS) eCall over IMS technology: This API could be used to explicitly send MSD to PSAP in respnse to MSD pull request from the PSAP.The modem will not automatically update MSD to the Public Safety Answering Point(PSAP) telux::tel::ICallListener::OnTpsMsdUpdateRequest.

For all other types of eCall: This API will update the eCall MSD in modem's cache.The modem automatically transmits MSD from this cache whenever there is an MSD pull request from Public Safety Answering Point (PSAP).

On platforms with Access control enabled, Caller needs to have TELUX_TEL_ECALL_MGMT permission to invoke this API successfully.

Parameters

in	*phoneId*	Represents phone corresponding to which updateECallMsd operation is performed
in	*msdPdu*	Encoded MSD(Minimum Set of Data) PDU as per spec EN 15722 2015 or GOST R 54620-2011/33464-2015 For Third Party Service(TPS) eCall over IMS technology: Maximum length allowed for MSD is 255 bytes For all other types of eCall: Maximum length allowed for MSD is 140 bytes
in	*callback*	Callback function to get the response of updateECallMsd.

Returns

Status of updateECallMsd i.e. success or suitable error code.

4.4.1.3.2.12 virtual telux::common::Status telux::tel::ICallManager::requestECallHlapTimerStatus (int *phoneId,* ECallHlapTimerStatusCallback *callback*) [pure virtual]

Request for status of eCall High Level Application Protocol(HLAP) timers that are maintained by the UE state machine. This does not retrieve status of timers maintained by the PSAP. The provided timers are as per EN 16062:2015 standard.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_ECALL_MGMT permission to invoke this API successfully.

Parameters

in	*phoneId*	Represents phone corresponding on which requestECallHlapTimerStatus operation is performed
in	*callback*	Callback function to get the response of requestECallHlapTimerStatus

Returns

Status of requestECallHlapTimerStatus i.e. success or suitable error code.

4.4.1.3.2.13 virtual std::vector<std::shared_ptr<ICall> > telux::tel::ICallManager::getInProgressCalls ( ) [pure virtual]

Get in-progress calls.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_CALL_INFO_READ permission to invoke this API successfully.

Returns

List of active calls.

4.4.1.3.2.14 virtual telux::common::Status telux::tel::ICallManager::conference (std::shared_ptr< ICall > *call1*, std::shared_ptr< ICall > *call2*, std::shared_ptr< telux::common::ICommandResponseCallback > *callback = nullptr*) [pure virtual]

Merge two calls in a conference.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_CALL_MGMT permission to invoke this API successfully.

Parameters

in	*call1*	Call object to conference.
in	*call2*	Call object to conference.
in	*callback*	Optional callback pointer to get the result of conference function

Returns

Status of conference i.e. success or suitable error code.

4.4.1.3.2.15 virtual telux::common::Status telux::tel::ICallManager::swap (std::shared_ptr< ICall > *callToHold*, std::shared_ptr< ICall > *callToActivate*, std::shared_ptr< telux::common::ICommandResponseCallback > *callback = nullptr*) [pure virtual]

Swap calls to make one active and put the another on hold.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_CALL_MGMT permission to invoke this API successfully.

Parameters

in	*callToHold*	Active call object to swap to hold state.
in	*callToActivate*	Hold call object to swap to active state.
in	*callback*	Optional callback pointer to get the result of swap function

Returns

Status of swap i.e. success or suitable error code.

4.4.1.3.2.16 virtual telux::common::Status telux::tel::ICallManager::hangupForegroundResume↩ Background (int *phoneId,* common::ResponseCallback *callback = nullptr*) [pure virtual]

Hangup all the foreground call(s) if any and accept the background call as the active call. The foreground call here could be active call, incoming call or multiple active calls in case of conference and background call could be held call or waiting call.

If a call(s) is active, the active call(s) will be terminated or if a call is waiting, the waiting call will be accepted and becomes active. Otherwise, if a held call is present, the held call becomes active. In case of hold and waiting calls, the hold call will still be on hold and waiting call will be accepted. In case of hold, active and waiting scenario, the hold call will still be on hold, active call will be ended and waiting call will be accepted.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_CALL_MGMT permission to invoke this API successfully.

Parameters

in	*phoneId*	- Represents phone corresponding to which this operation is performed.
in	*callback*	- optional callback pointer to get the response of hangup request below are possible error codes for callback response • telux::common::ErrorCode::SUCCESS • telux::common::ErrorCode::RADIO_NOT_AVAILABLE • telux::common::ErrorCode::NO_MEMORY • telux::common::ErrorCode::MODEM_ERR • telux::common::ErrorCode::INTERNAL_ERR • telux::common::ErrorCode::INVALID_STATE • telux::common::ErrorCode::INVALID_CALL_ID • telux::common::ErrorCode::INVALID_ARGUMENTS • telux::common::ErrorCode::OPERATION_NOT_ALLOW↩ ED • telux::common::ErrorCode::GENERIC_FAILURE

Returns

Status of hangupForegroundResumeBackground i.e. success or suitable error code.

4.4.1.3.2.17 virtual telux::common::Status telux::tel::ICallManager::hangupWaitingOrBackground (int *phoneId,* common::ResponseCallback *callback = nullptr*) [pure virtual]

Hangup all the waiting or background call(s). The background call here could be waiting call, hold call or multiple hold calls in case of conference.

If a call(s) is hold, the hold call(s) will be terminated or if a call is waiting, the waiting call will be be terminated as well. In case of hold, active and waiting scenario, the active call will still be on active, hold and waiting call will be ended.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_CALL_MGMT permission

to invoke this API successfully.

Parameters

in	*phoneId*	- Represents phone corresponding to which this operation is performed.
in	*callback*	- optional callback pointer to get the response of hangup request below are possible error codes for callback response • telux::common::ErrorCode::SUCCESS • telux::common::ErrorCode::RADIO_NOT_AVAILABLE • telux::common::ErrorCode::NO_MEMORY • telux::common::ErrorCode::MODEM_ERR • telux::common::ErrorCode::INTERNAL_ERR • telux::common::ErrorCode::INVALID_STATE • telux::common::ErrorCode::INVALID_CALL_ID • telux::common::ErrorCode::INVALID_ARGUMENTS • telux::common::ErrorCode::OPERATION_NOT_ALLOWED • telux::common::ErrorCode::GENERIC_FAILURE

Returns

Status of hangupWaitingOrBackground i.e. success or suitable error code.

Note

Eval: This is a new API and is being evaluated.It is subject to change and could break backwards compatibility.

4.4.1.3.2.18 virtual telux::common::Status telux::tel::ICallManager::requestEcbm (int *phoneId*, EcbmStatusCallback *callback*) [pure virtual]

Request for emergency callback mode

Parameters

in	*phoneId*	Represents the phone corresponding to which the emergency callback mode(ECBM) status is requested.
in	*callback*	Callback pointer to get the result of ECBM status request

Returns

Status of requestEcbm i.e. success or suitable error code.

4.4.1.3.2.19 virtual telux::common::Status telux::tel::ICallManager::exitEcbm (int *phoneId,* common::ResponseCallback *callback = nullptr*) [pure virtual]

Exit emergency callback mode.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_EMERGENCY_OPS permission to invoke this API successfully.

Parameters

in	*phoneId*	Represents the phone corresponding to which the emergency callback mode(ECBM) exit is requested.
in	*callback*	Optional callback pointer to get the result of exit ECBM request

Returns

Status of exitEcbm i.e. success or suitable error code.

4.4.1.3.2.20 virtual telux::common::Status telux::tel::ICallManager::requestNetworkDeregistration (int *phoneId,* common::ResponseCallback *callback = nullptr*) [pure virtual]

Deregister from the network after an eCall when the modem is in eCall-only mode. This is typically done after the T9 eCall HLAP timer has expired to stop the T10 eCall HLAP timer and deregister from the serving network.

To invoke this API on platforms with access control enabled, the caller needs to have TELUX_TEL_ECALL_MGMT permission.

Parameters

in	*phoneId*	Represents the phone corresponding to which the network deregistration will be performed.
in	*callback*	Callback function to get the response of the request. The response is sent after the operation is complete.

Returns

Status of requestNetworkDeregistration request, i.e., success or suitable error code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.4.1.3.2.21 virtual telux::common::Status telux::tel::ICallManager::updateEcallHlapTimer (int *phoneId,* HlapTimerType *type,* uint32_t *timeDuration,* common::ResponseCallback *callback = nullptr*) [pure virtual]

Set the value of an eCall HLAP timer. Only the T10 Timer is supported currently.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_ECALL_MGMT permission to invoke this API successfully.

Parameters

in	*phoneId*	Represents the phone corresponding to which the value of T10 eCall HLAP timer updated will be performed.
in	*type*	HlapTimerType
in	*timeDuration*	Represents the time duration for the HLAP timer. T10 timer is in units of minutes, and the supported range is from 60 to 720.
in	*callback*	Callback function to get the response of the request. The response is sent after the operation is complete.

Returns

Status of updateEcallHlapTimer i.e., success or suitable error code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.4.1.3.2.22 virtual telux::common::Status telux::tel::ICallManager::requestEcallHlapTimer (int *phoneId*, HlapTimerType *type*, ECallHlapTimerCallback *callback*) [pure virtual]

Get the value of an eCall HLAP timer. Only the T10 Timer is supported currently.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_ECALL_MGMT permission to invoke this API successfully.

Parameters

in	*phoneId*	Represents the phone corresponding to which the value of eCall HLAP timer query will be performed.
in	*type*	HlapTimerType
in	*callback*	Callback function to get the response of the request. The response is sent after the operation is complete.

Returns

Status of requestEcallHlapTimer i.e., success or suitable error code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.4.1.3.2.23 virtual telux::common::Status telux::tel::ICallManager::setECallConfig (EcallConfig *config*) [pure virtual]

Set the configuration related to emergency call. The configuration is persistent and takes effect when the next emergency call is dialed.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_ECALL_MGMT permissionto invoke this API successfully.

Parameters

in	*config*	eCall configuration to be set EcallConfig

Returns

Status of setECallConfig i.e. success or suitable error code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.4.1.3.2.24 virtual telux::common::Status telux::tel::ICallManager::getECallConfig (EcallConfig & *config*) [pure virtual]

Get the configuration related to emergency call.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_ECALL_MGMT permissionto invoke this API successfully.

Parameters

out	*config*	Parameter to hold the fetched eCall configuration EcallConfig

Returns

Status of getECallConfig i.e. success or suitable error code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.4.1.3.2.25 virtual telux::common::Status telux::tel::ICallManager::registerListener (std::shared_ptr< telux::tel::ICallListener > *listener*) [pure virtual]

Add a listener to listen for incoming call, call info change and eCall MSD transmission status change.

Parameters

in	*listener*	Pointer to ICallListener object which receives event corresponding to phone

Returns

Status of registerListener i.e. success or suitable error code.

4.4.1.3.2.26 virtual telux::common::Status telux::tel::ICallManager::removeListener (std::shared_ptr< telux::tel::ICallListener > *listener*) [pure virtual]

Remove a previously added listener.

Parameters

in	*listener*	Listener to be removed.

Returns

Status of removeListener i.e. success or suitable error code.

4.4.1.4 class telux::tel::IMakeCallCallback

Interface for Make Call callback object. Client needs to implement this interface to get single shot responses for commands like make call.

The methods in callback can be invoked from multiple different threads. The implementation should be thread safe.

Public member functions

- virtual void makeCallResponse (telux::common::ErrorCode error, std::shared_ptr< ICall > call=nullptr)
- virtual ~IMakeCallCallback ()

4.4.1.4.1 Constructors and Destructors

4.4.1.4.1.1 virtual telux::tel::IMakeCallCallback::~IMakeCallCallback ( ) [virtual]

4.4.1.4.2 Member Function Documentation

4.4.1.4.2.1 virtual void telux::tel::IMakeCallCallback::makeCallResponse (telux::common::ErrorCode *error,* std::shared_ptr< ICall > *call = nullptr*) [virtual]

This function is called with the response to makeCall API.

Parameters

out	*error*	telux::common::ErrorCode
out	*call*	Pointer to Call object or nullptr in case of failure

4.4.2 Enumeration Type Documentation

4.4.2.1 enum telux::tel::CallDirection [strong]

Defines type of call like incoming, outgoing and none.

Enumerator

INCOMING
OUTGOING
NONE

4.4.2.2 enum telux::tel::CallState [strong]

Defines the states a call can be in

Enumerator

CALL_IDLE idle call, default state of a newly created call object
CALL_ACTIVE active call
CALL_ON_HOLD on hold call
CALL_DIALING out going call, in dialing state and not yet connected, MO Call only
CALL_INCOMING incoming call, not yet answered
CALL_WAITING waiting call
CALL_ALERTING alerting call, MO Call only
CALL_ENDED call ended / disconnected

4.4.2.3 enum telux::tel::CallEndCause [strong]

Reason for the recently terminated call (either normally ended or failed)

Enumerator

UNOBTAINABLE_NUMBER
NO_ROUTE_TO_DESTINATION
CHANNEL_UNACCEPTABLE
OPERATOR_DETERMINED_BARRING
NORMAL
BUSY
NO_USER_RESPONDING
NO_ANSWER_FROM_USER
NOT_REACHABLE
CALL_REJECTED
NUMBER_CHANGED
PREEMPTION
DESTINATION_OUT_OF_ORDER

INVALID_NUMBER_FORMAT
FACILITY_REJECTED
RESP_TO_STATUS_ENQUIRY
NORMAL_UNSPECIFIED
CONGESTION
NETWORK_OUT_OF_ORDER
TEMPORARY_FAILURE
SWITCHING_EQUIPMENT_CONGESTION
ACCESS_INFORMATION_DISCARDED
REQUESTED_CIRCUIT_OR_CHANNEL_NOT_AVAILABLE
RESOURCES_UNAVAILABLE_OR_UNSPECIFIED
QOS_UNAVAILABLE
REQUESTED_FACILITY_NOT_SUBSCRIBED
INCOMING_CALLS_BARRED_WITHIN_CUG
BEARER_CAPABILITY_NOT_AUTHORIZED
BEARER_CAPABILITY_UNAVAILABLE
SERVICE_OPTION_NOT_AVAILABLE
BEARER_SERVICE_NOT_IMPLEMENTED
ACM_LIMIT_EXCEEDED
REQUESTED_FACILITY_NOT_IMPLEMENTED
ONLY_DIGITAL_INFORMATION_BEARER_AVAILABLE
SERVICE_OR_OPTION_NOT_IMPLEMENTED
INVALID_TRANSACTION_IDENTIFIER
USER_NOT_MEMBER_OF_CUG
INCOMPATIBLE_DESTINATION
INVALID_TRANSIT_NW_SELECTION
SEMANTICALLY_INCORRECT_MESSAGE
INVALID_MANDATORY_INFORMATION
MESSAGE_TYPE_NON_IMPLEMENTED
MESSAGE_TYPE_NOT_COMPATIBLE_WITH_PROTOCOL_STATE
INFORMATION_ELEMENT_NON_EXISTENT
CONDITIONAL_IE_ERROR
MESSAGE_NOT_COMPATIBLE_WITH_PROTOCOL_STATE
RECOVERY_ON_TIMER_EXPIRED
PROTOCOL_ERROR_UNSPECIFIED
INTERWORKING_UNSPECIFIED
CALL_BARRED
FDN_BLOCKED
IMSI_UNKNOWN_IN_VLR
IMEI_NOT_ACCEPTED
DIAL_MODIFIED_TO_USSD
DIAL_MODIFIED_TO_SS
DIAL_MODIFIED_TO_DIAL
CDMA_LOCKED_UNTIL_POWER_CYCLE
CDMA_DROP
CDMA_INTERCEPT
CDMA_REORDER
CDMA_SO_REJECT
CDMA_RETRY_ORDER
CDMA_ACCESS_FAILURE

CDMA_PREEMPTED
CDMA_NOT_EMERGENCY
CDMA_ACCESS_BLOCKED
ERROR_UNSPECIFIED

4.5 SMS

This section contains APIs related to Sending and Receiving SMS.

4.5.1 Data Structure Documentation

4.5.1.1 struct telux::tel::DeleteInfo

Specify delete information used for deleting message on storage.

Data fields

Type	Field	Description
DeleteType	delType	Specifies the type of delete operation to be performed
SmsTagType	tagType	1.If SMS tag type is set to telux::tel::SmsTagType::UNKNOWN and delType is set to telux::tel::DeleteType::DELETE_ALL then all messages on the storage would be deleted. 2.To delete all messages of a particular tag, set tagType to the particular tag like telux::tel::SmsTagType::MT_READ and delType to telux::tel::DeleteType::DELETE_MESSAGES_BY_TAG
uint32_t	msgIndex	To delete message at specific index, specify msgIndex and delType as telux::tel::DeleteType::DELETE_MSG_AT_INDEX

4.5.1.2 struct telux::tel::SmsMetaInfo

Provides certain attributes of an SMS message.

Data fields

Type	Field	Description
uint32_t	msgIndex	Message index on storage
SmsTagType	tagType	SMS tag type

4.5.1.3 struct telux::tel::MessageAttributes

Contains structure of message attributes like encoding type, number of segments, characters left in last segment.

Data fields

Type	Field	Description
SmsEncoding	encoding	Data encoding type
int	numberOf↩ Segments	Number of segments
int	segmentSize	Max size of each segment
int	numberOf↩ CharsLeftIn↩ LastSegment	characters left in last segment

4.5.1.4 struct telux::tel::MessagePartInfo

Structure containing information about the part of multi-part SMS such as concatenated message reference number, number of segments and segment number. During concatenation this information along with originating address helps in associating each part of the multi-part message to the corresponding multi-part message.

Data fields

Type	Field	Description
uint16_t	refNumber	Concatenated message reference number as per spec 3GPP TS 23.040 9.2.3.24.1. For each part of multipart message this message reference will be the same
uint8_t	numberOf↩ Segments	Number of segments
uint8_t	segment↩ Number	Segment Number

4.5.1.5 class telux::tel::SmsMessage

Data structure represents an incoming SMS. This is applicable for single part message or part of the multipart message.

Public member functions

- SmsMessage (std::string text, std::string sender, std::string receiver, SmsEncoding encoding, std::string pdu, PduBuffer pduBuffer, std::shared_ptr< MessagePartInfo > info)
- SmsMessage (std::string text, std::string sender, std::string receiver, SmsEncoding encoding, std::string pdu, PduBuffer pduBuffer, std::shared_ptr< MessagePartInfo > info, bool isMetaInfoValid, SmsMetaInfo metaInfo)
- const std::string & getText () const
- const std::string & getSender () const
- const std::string & getReceiver () const
- SmsEncoding getEncoding () const
- const std::string & getPdu () const
- PduBuffer getRawPdu () const
- std::shared_ptr< MessagePartInfo > getMessagePartInfo ()
- const std::string toString () const
- telux::common::Status getMetaInfo (SmsMetaInfo &metaInfo)

4.5.1.5.1 Constructors and Destructors

4.5.1.5.1.1 telux::tel::SmsMessage::SmsMessage (std::string *text,* std::string *sender,* std::string *receiver,* SmsEncoding *encoding,* std::string *pdu,* PduBuffer *pduBuffer,* std::shared_ptr< MessagePartInfo > *info*)

4.5.1.5.1.2 telux::tel::SmsMessage::SmsMessage (std::string *text,* std::string *sender,* std::string *receiver,* SmsEncoding *encoding,* std::string *pdu,* PduBuffer *pduBuffer,* std::shared_ptr< MessagePartInfo > *info,* bool *isMetaInfoValid,* SmsMetaInfo *metaInfo*)

4.5.1.5.2 Member Function Documentation

4.5.1.5.2.1 const std::string& telux::tel::SmsMessage::getText ( ) const

Get the message text for the single part message or part of the multipart message.

Returns

String containing SMS message.

4.5.1.5.2.2 const std::string& telux::tel::SmsMessage::getSender ( ) const

Get the originating address (sender address).

Returns

String containing sender address.

4.5.1.5.2.3 const std::string& telux::tel::SmsMessage::getReceiver ( ) const

Get the destination address (receiver address).

Returns

String containing receiver address

4.5.1.5.2.4 SmsEncoding telux::tel::SmsMessage::getEncoding ( ) const

Get encoding format used for the single part message or part of the multipart message.

Returns

SMS message encoding used.

4.5.1.5.2.5 const std::string& telux::tel::SmsMessage::getPdu ( ) const

Get the raw PDU for the single part message or part of the multipart message.

Returns

String containing raw PDU content.

Deprecated

Use API SmsMessage::getRawPdu

4.5.1.5.2.6 PduBuffer telux::tel::SmsMessage::getRawPdu ( ) const

Get the raw PDU buffer for the single part message or part of the multipart message.

Returns

Buffer containing raw PDU content.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backward compatibility

4.5.1.5.2.7 std::shared_ptr<MessagePartInfo> telux::tel::SmsMessage::getMessagePartInfo ( )

Applicable for multi-part SMS only. Get the information such as segment number, number of segments and concatenated reference number corresponding to the part of multi-part SMS.

Returns

If a message is single part SMS the method returns null otherwise returns message part information.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.5.1.5.2.8 const std::string telux::tel::SmsMessage::toString ( ) const

Get the text related informative representation of this object.

Returns

String containing informative string.

4.5.1.5.2.9 telux::common::Status telux::tel::SmsMessage::getMetaInfo (SmsMetaInfo & *metaInfo*)

Get meta information of SMS stored in storage. There is no meta information when storage type is none.

Parameters

out	*metaInfo*	Meta information about SMS message stored in storage.

Returns

Status of getMetaInfo i.e. success or suitable error code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.5.1.6 class telux::tel::ISmsManager

SmsManager class is the primary interface to manage SMS operations such as send and receive an SMS text and raw encoded PDU(s). This class handles single part and multi-part messages.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual telux::common::Status sendSms (const std::string &message, const std::string &receiver↩ Address, std::shared_ptr< telux::common::ICommandResponseCallback > sentCallback=nullptr, std::shared_ptr< telux::common::ICommandResponseCallback > deliveryCallback=nullptr)=0
- virtual telux::common::Status sendSms (std::string message, std::string receiverAddress, bool deliveryReportNeeded=true, SmsResponseCb sentCallback=nullptr, std::string smscAddr="")=0
- virtual telux::common::Status sendRawSms (const std::vector< PduBuffer > rawPdus, SmsResponseCb sentCallback=nullptr)=0
- virtual telux::common::Status requestSmscAddress (std::shared_ptr< ISmscAddressCallback > callback=nullptr)=0
- virtual telux::common::Status setSmscAddress (const std::string &smscAddress, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestSmsMessageList (SmsTagType type, RequestSmsInfoListCb callback)=0
- virtual telux::common::Status readMessage (uint32_t messageIndex, ReadSmsMessageCb callback)=0
- virtual telux::common::Status deleteMessage (DeleteInfo info, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestPreferredStorage (RequestPreferredStorageCb callback)=0
- virtual telux::common::Status setPreferredStorage (StorageType storageType, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status setTag (uint32_t msgIndex, SmsTagType tagType, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestStorageDetails (RequestStorageDetailsCb callback)=0
- virtual MessageAttributes calculateMessageAttributes (const std::string &message)=0
- virtual int getPhoneId ()=0
- virtual telux::common::Status registerListener (std::weak_ptr< ISmsListener > listener)=0

- virtual telux::common::Status removeListener (std::weak_ptr< ISmsListener > listener)=0
- virtual ~ISmsManager ()

4.5.1.6.1 Constructors and Destructors

4.5.1.6.1.1 virtual telux::tel::ISmsManager::~ISmsManager ( ) [virtual]

4.5.1.6.2 Member Function Documentation

4.5.1.6.2.1 virtual telux::common::ServiceStatus telux::tel::ISmsManager::getServiceStatus ( ) [pure virtual]

This status indicates whether the ISmsManager object is in a usable state.

Returns

telux::common::ServiceStatus

4.5.1.6.2.2 virtual telux::common::Status telux::tel::ISmsManager::sendSms (const std::string & *message,* const std::string & *receiverAddress,* std::shared_ptr< telux::common::ICommandResponseCallback > *sentCallback = nullptr,* std::shared_ptr< telux::common::ICommandResponseCallback > *deliveryCallback = nullptr*) [pure virtual]

Send SMS to the destination address. When registered on IMS the SMS will be attempted over IMS. If sending SMS over IMS fails, an automatic retry would be attempted to send the message over CS. Only support UCS2 format, GSM 7 bit default alphabet and does not support National language shift tables.

On platforms with access control enabled, caller needs to have TELUX_TEL_SMS_OPS permission to invoke this API successfully.

Parameters

in	*message*	Message text to be sent
in	*receiverAddress*	Receiver or destination address
in	*sentCallback*	Optional callback pointer to get the response of send SMS request.
in	*deliveryCallback*	Optional callback pointer to get message delivery status

Deprecated

Use API ISmsManager::sendSms(const std::string &message, const std::string &receiverAddress, bool deliveryReportNeeded = true, SmsResponseCb callback = nullptr, std::string smscAddr = "")

Returns

Status of sendSms i.e. success or suitable error code.

4.5.1.6.2.3 virtual telux::common::Status telux::tel::ISmsManager::sendSms (std::string *message*, std::string *receiverAddress*, bool *deliveryReportNeeded* = `true`, SmsResponseCb *sentCallback* = `nullptr`, std::string *smscAddr* = `""`) [pure virtual]

Send single or multipart SMS to the destination address. When registered on IMS the SMS will be attempted over IMS. If sending SMS over IMS fails, an automatic retry would be attempted to send the message over CS. Only support UCS2 format, GSM 7 bit default alphabet and does not support National language shift tables. The SMS is sent directly not stored on storage.

On platforms with access control enabled, caller needs to have TELUX_TEL_SMS_OPS permission to invoke this API successfully.

Parameters

in	*message*	Message text to be send.
in	*receiverAddress*	Receiver or destination address
in	*deliveryReportNeeded*	Delivery status received in the listener API telux::tel::ISmsListener if deliveryReportNeeded is true. Provided recipient responds to SMSC before the validity period expires. If deliveryReportNeeded is false delivery report will not be received.
in	*sentCallback*	Optional callback pointer to get the sent response for single part or multi-part SMS.
in	*smscAddr*	SMS is sent to SMSC address. If SMSC address is empty then pre-configured SMSC address is used.

Returns

Status of sendSms i.e. success or suitable error code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.5.1.6.2.4 virtual telux::common::Status telux::tel::ISmsManager::sendRawSms (const std::vector< PduBuffer > *rawPdus*, SmsResponseCb *sentCallback* = `nullptr`) [pure virtual]

Send an SMS that is provided as a raw encoded PDU(s). When registered on IMS the SMS will be attempted over IMS. If sending SMS over IMS fails, an automatic retry would be attempted to send the message over CS. If the SMS is a multi-part message, the API expects multiple PDU to be passed to it. The SMS is sent directly not stored on storage.

On platforms with access control enabled, caller needs to have TELUX_TEL_SMS_OPS permission to invoke this API successfully.

Parameters

in	*rawPdus*	Each element in the vector represents a part of a multipart message. For single part message the vector will have single element.

in	*sentCallback*	Optional callback to get the sent response for single part or multi-part SMS.

Returns

Status of sendRawSms i.e. success or suitable error code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.5.1.6.2.5 virtual telux::common::Status telux::tel::ISmsManager::requestSmscAddress (std::shared_ptr< ISmscAddressCallback > *callback = nullptr*) [pure virtual]

Request for Short Messaging Service Center (SMSC) Address.Purpose of SMSC is to store, forward, convert and deliver Short Message Service (SMS) messages.

On platforms with access control enabled, caller needs to have TELUX_TEL_SMS_CONFIG permission to invoke this API successfully.

Parameters

in	*callback*	Optional callback pointer to get the response of Smsc address request

Returns

Status of getSmscAddress i.e. success or suitable error code.

4.5.1.6.2.6 virtual telux::common::Status telux::tel::ISmsManager::setSmscAddress (const std::string & *smscAddress,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

Sets the Short Message Service Center(SMSC) address on the device.

On platforms with access control enabled, caller needs to have TELUX_TEL_SMS_CONFIG permission to invoke this API successfully.

This will change the SMSC address for all the SMS messages sent from any app.

Parameters

in	*smscAddress*	SMSC address
in	*callback*	Optional callback pointer to get the response of set SMSC address

Returns

Status of setSmscAddress i.e. success or suitable error code.

4.5.1.6.2.7 virtual telux::common::Status telux::tel::ISmsManager::requestSmsMessageList (SmsTagType *type,* RequestSmsInfoListCb *callback*) [pure virtual]

Requests a list of message information for the messages saved in SIM storage.

On platforms with access control enabled, the caller needs to have TELUX_TEL_SMS_STORAGE permission to invoke this API successfully.

Parameters

in	*type*	Specifies the tag type of the SMS message that should be matched when retrieving the list. Specifying telux::tel::SmsTagType::UNKNOWN will retrieve all the messages from storage.
in	*callback*	Callback to get the response of request SMS messages info.

Returns

Status of requestSmsMessageList i.e. success or suitable error code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.5.1.6.2.8 virtual telux::common::Status telux::tel::ISmsManager::readMessage (uint32_t *messageIndex,* ReadSmsMessageCb *callback*) [pure virtual]

Retrieve a particular message from SIM storage matching the index.

On platforms with access control enabled, the caller needs to have TELUX_TEL_SMS_STORAGE permission to invoke this API successfully.

Parameters

in	*messageIndex*	SMS index on storage.
in	*callback*	Callback to get the response of read SMS message from storage .

Returns

Status of readMessage i.e. success or suitable error code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.5.1.6.2.9 virtual telux::common::Status telux::tel::ISmsManager::deleteMessage (DeleteInfo *info,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

Delete specific SMS based on message index or delete messages on SIM storage based on telux::tel::SmsTagType or delete all messages from SIM storage.

On platforms with access control enabled, the caller needs to have TELUX_TEL_SMS_STORAGE permission to invoke this API successfully.

Parameters

in	*info*	Specify delete information based on which messages are deleted
in	*callback*	Optional callback to get the response of delete SMS message from storage .

Returns

Status of deleteMessage i.e. success or suitable error code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.5.1.6.2.10 virtual telux::common::Status telux::tel::ISmsManager::requestPreferredStorage (RequestPreferredStorageCb *callback*) [pure virtual]

Request preferred storage for incoming SMS.

On platforms with access control enabled, the caller needs to have TELUX_TEL_SMS_CONFIG permission to invoke this API successfully.

Parameters

in	*callback*	Callback to get the response of get preferred storage type .

Returns

Status of requestPreferredStorage i.e. success or suitable error code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.5.1.6.2.11 virtual telux::common::Status telux::tel::ISmsManager::setPreferredStorage (StorageType *storageType,* telux::common::ResponseCallback *callback* = `nullptr`) [pure virtual]

Set the preferred storage for incoming SMS. All future messages that arrive will be stored on the storage set in this API, if any. Messages in the current storage will not be moved to the new storage. If client does not require messages to be stored by the platform, then the storage could be set to telux::tel::StorageType::NONE.

On platforms with access control enabled, the caller needs to have TELUX_TEL_SMS_CONFIG permission to invoke this API successfully.

Parameters

in	*storageType*	telux::tel::StorageType
in	*callback*	Optional callback to get the response of set preferred storage.

Returns

Status of setPreferredStorage i.e. success or suitable error code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.5.1.6.2.12 virtual telux::common::Status telux::tel::ISmsManager::setTag (uint32_t *msgIndex,* SmsTagType *tagType,* telux::common::ResponseCallback *callback* = `nullptr`) [pure virtual]

Update the tag of the incoming message stored in SIM storage as read/unread

On platforms with access control enabled, the caller needs to have TELUX_TEL_SMS_OPS permission to invoke this API successfully.

Parameters

in	*msgIndex*	Message index corresponding to message in storage for which tag needs to be updated.
in	*tagType*	telux::tel::SmsTagType. The applicable tag types are MT_READ and MT_NOT_READ.
in	*callback*	Optional callback to get the response of updating the tag.

Returns

Status of setTag i.e. success or suitable error code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.5.1.6.2.13 virtual telux::common::Status telux::tel::ISmsManager::requestStorageDetails (RequestStorageDetailsCb *callback*) [pure virtual]

Request details about SIM storage like total size and available size in terms of number of messages.

On platforms with access control enabled, the caller needs to have TELUX_TEL_SMS_CONFIG permission to invoke this API successfully.

Parameters

in	*callback*	Callback to get the response of storage detail request.

Returns

Status of requestStorageDetails i.e. success or suitable error code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.5.1.6.2.14 virtual MessageAttributes telux::tel::ISmsManager::calculateMessageAttributes (const std::string & *message*) [pure virtual]

Calculate message attributes for the given message.

Parameters

in	*message*	Message to send

Returns

MessageAttributes structure containing encoding type, number of segments, max size of segment and characters left in last segment.

4.5.1.6.2.15 virtual int telux::tel::ISmsManager::getPhoneId ( ) [pure virtual]

Get associated phone id for this SMSManager.

Returns

PhoneId.

4.5.1.6.2.16 virtual telux::common::Status telux::tel::ISmsManager::registerListener (std::weak_ptr< ISmsListener > *listener*) [pure virtual]

Register a listener for Sms events

Parameters

in	*listener*	Pointer to ISmsListener object which receives event corresponding to SMS

Returns

Status of registerListener i.e. success or suitable error code.

4.5.1.6.2.17 virtual telux::common::Status telux::tel::ISmsManager::removeListener (std::weak_ptr< ISmsListener > *listener*) [pure virtual]

Remove a previously added listener.

Parameters

in	*listener*	Pointer to ISmsListener object

Returns

Status of removeListener i.e. success or suitable error code.

4.5.1.7 class telux::tel::ISmsListener

A listener class receives notification for the incoming message(s) and delivery report for sent message(s).

The methods in listener can be invoked from multiple different threads. The implementation should be thread safe.

Public member functions

- virtual void onIncomingSms (int phoneId, std::shared_ptr< SmsMessage > message)
- virtual void onIncomingSms (int phoneId, std::shared_ptr< std::vector< SmsMessage >> messages)
- virtual void onDeliveryReport (int phoneId, int msgRef, std::string receiverAddress, telux::common::ErrorCode error)
- virtual void onMemoryFull (int phoneId, StorageType type)
- virtual ~ISmsListener ()

4.5.1.7.1 Constructors and Destructors

4.5.1.7.1.1 virtual telux::tel::ISmsListener::~ISmsListener ( ) [virtual]

4.5.1.7.2 Member Function Documentation

4.5.1.7.2.1 virtual void telux::tel::ISmsListener::onIncomingSms (int *phoneId,* std::shared_ptr< SmsMessage > *message*) [virtual]

This function will be invoked when a single part message is received or when a part of a multi-part message is received. If the SMS preferred storage is to store the SMS in storage i.e SIM then the SMS will be first stored in storage and then this API will be invoked.

On platforms with access control enabled, the client needs to have TELUX_TEL_SMS_LISTEN permission to invoke this API successfully.

Parameters

in	*phoneId*	Unique identifier per SIM slot. Phone on which the message is received.
in	*message*	Pointer to SmsMessage object

4.5.1.7.2.2 virtual void telux::tel::ISmsListener::onIncomingSms (int *phoneId,* std::shared_ptr< std::vector< SmsMessage >> *messages*) [virtual]

This function will be invoked when either a single part message is received, or when all the parts of a multipart message have been received. This API is invoked only once all parts of a message are received. In case of a single part message, it will be invoked as soon as it is received. In case of multi-part, the implementation waits for all parts of the message to arrive and then invokes this API. If the SMS preferred storage is to store the SMS in storage i.e SIM then the messages will be first stored in storage and then this API will be invoked.

On platforms with access control enabled, the client needs to have TELUX_TEL_SMS_LISTEN permission to invoke this API successfully.

Parameters

in	*phoneId*	Unique identifier per SIM slot. Phone on which the message is received.
in	*messages*	Pointer to list of SmsMessage received corresponding to single part or all parts of multipart message.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.5.1.7.2.3 virtual void telux::tel::ISmsListener::onDeliveryReport (int *phoneId,* int *msgRef,* std::string *receiverAddress,* telux::common::ErrorCode *error*) [virtual]

This function will be invoked when either a delivery report for a single part message is received or when the delivery report for part of a multi-part message is received. In order to determine delivery of all parts of the

multi-part message, the client application shall compare message reference received in the delivery indications with message references received in telux::tel::SmsResponseCb.

On platforms with access control enabled, the client needs to have TELUX_TEL_SMS_OPS permission to invoke this API successfully.

Parameters

in	*phoneId*	Unique identifier per SIM slot. Phone on which the message is received.
in	*msgRef*	Message reference number (as per spec 3GPP TS 23.040 9.2.2.3) for a single part message or part of multipart message.
in	*receiverAddress*	Receiver or destination address
in	*error*	telux::common::ErrorCode

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.5.1.7.2.4 virtual void telux::tel::ISmsListener::onMemoryFull (int *phoneId,* StorageType *type*) [virtual]

This function will be invoked when SMS storage is full.

On platforms with access control enabled, the client needs to have TELUX_TEL_SMS_STORAGE permission to invoke this API successfully.

Parameters

in	*phoneId*	Unique identifier per SIM slot. Phone on which the message is received.
in	*type*	telux::tel::StorageType. Applicable storage type StorageType::SIM

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.5.1.8 class telux::tel::ISmscAddressCallback

Interface for SMS callback object. Client needs to implement this interface to get single shot responses for send SMS.

The methods in callback can be invoked from multiple different threads. The implementation should be thread safe.

Public member functions

- virtual void smscAddressResponse (const std::string &address, telux::common::ErrorCode error)=0

4.5.1.8.1 Member Function Documentation

4.5.1.8.1.1 virtual void telux::tel::ISmscAddressCallback::smscAddressResponse (const std::string & *address*, telux::common::ErrorCode *error*) [pure virtual]

This function is called with the response to the Smsc address request.

Parameters

in	*address*	Smsc address
in	*error*	telux::common::ErrorCode

4.5.2 Enumeration Type Documentation

4.5.2.1 enum telux::tel::SmsEncoding [strong]

Specifies the encoding of the SMS message.

Enumerator

GSM7 GSM 7-bit default alphabet encoding
GSM8 GSM 8-bit data encoding
UCS2 UCS-2 encoding
UNKNOWN Unknown encoding

4.5.2.2 enum telux::tel::SmsTagType [strong]

Specifies the SMS tag type. All incoming messages will be received and stored with tag as MT_NOT_READ. It is the client's responsibility to update the tag to MT_READ using telux::tel::ISmsManager::setTag whenever the message is considered read.

Enumerator

UNKNOWN Unknown tag type
MT_READ MT message marked as read
MT_NOT_READ MT message marked as not read

4.5.2.3 enum telux::tel::DeleteType [strong]

Specifies the type of delete operation to be performed.

Enumerator

UNKNOWN Unknown delete type
DELETE_ALL Delete all message from memory storage
DELETE_MESSAGES_BY_TAG Deletes all messages from the memory storage that match the

specific message tag

DELETE_MSG_AT_INDEX Deletes only the message at the specific index from the memory storage

4.5.2.4 enum telux::tel::StorageType `[strong]`

Specifies the SMS storage type for incoming message.

Enumerator

UNKNOWN Unknown storage type

NONE This indicates SMS not stored on any of storage and is directly notified to client. This is the default storage type

SIM This indicates SMS is stored on SIM

4.6 SIM Card Services

This section contains APIs related to Card Services.

4.6.1 Data Structure Documentation

4.6.1.1 class telux::tel::ICardApp

Represents a single card application.

Public member functions

- virtual AppType getAppType ()=0
- virtual AppState getAppState ()=0
- virtual std::string getAppId ()=0
- virtual telux::common::Status changeCardPassword (CardLockType lockType, std::string oldPwd, std::string newPwd, PinOperationResponseCb callback)=0
- virtual telux::common::Status unlockCardByPuk (CardLockType lockType, std::string puk, std::string newPin, PinOperationResponseCb callback)=0
- virtual telux::common::Status unlockCardByPin (CardLockType lockType, std::string pin, PinOperationResponseCb callback)=0
- virtual telux::common::Status queryPin1LockState (QueryPin1LockResponseCb callback)=0
- virtual telux::common::Status queryFdnLockState (QueryFdnLockResponseCb callback)=0
- virtual telux::common::Status setCardLock (CardLockType lockType, std::string password, bool isEnabled, PinOperationResponseCb callback)=0
- virtual ~ICardApp ()

4.6.1.1.1 Constructors and Destructors

4.6.1.1.1.1 virtual telux::tel::ICardApp::~ICardApp ( ) [virtual]

4.6.1.1.2 Member Function Documentation

4.6.1.1.2.1 virtual AppType telux::tel::ICardApp::getAppType ( ) [pure virtual]

Get Application type like SIM, USIM, RUIM, CSIM or ISIM.

Returns

AppType.

4.6.1.1.2.2 virtual AppState telux::tel::ICardApp::getAppState ( ) [pure virtual]

Get Application state like PIN1, PUK required and others.

Returns

AppState.

4.6.1.1.2.3 virtual std::string telux::tel::ICardApp::getAppId ( ) [pure virtual]

Get application identifier.

Returns

Application Id.

4.6.1.1.2.4 virtual telux::common::Status telux::tel::ICardApp::changeCardPassword (CardLockType *lockType,* std::string *oldPwd,* std::string *newPwd,* PinOperationResponseCb *callback*) [pure virtual]

Change the password used in PIN1/PIN2 lock.

On platforms with access control enabled, caller needs to have TELUX_TEL_CARD_PRIVILEGED_OPS permission to invoke this API successfully.

Parameters

in	*lockType*	CardLockType. Applicable lock types are PIN1 and PIN2.
in	*oldPwd*	Old password
in	*newPwd*	New password
in	*callback*	Callback function to get the response of change pin password.

4.6.1.1.2.5 virtual telux::common::Status telux::tel::ICardApp::unlockCardByPuk (CardLockType *lockType,* std::string *puk,* std::string *newPin,* PinOperationResponseCb *callback*) [pure virtual]

Unlock the Sim card for an app by entering PUK and new pin.

On platforms with access control enabled, caller needs to have TELUX_TEL_CARD_PRIVILEGED_OPS permission to invoke this API successfully.

Parameters

in	*lockType*	CardLockType. Applicable lock types are PUK1 and PUK2
in	*puk*	PUK1/PUK2
in	*newPin*	New PIN1/PIN2
in	*callback*	Callback function to get the response of unlock card lock.

4.6.1.1.2.6 virtual telux::common::Status telux::tel::ICardApp::unlockCardByPin (CardLockType *lockType,* std::string *pin,* PinOperationResponseCb *callback*) [pure virtual]

Unlock the Sim card for an app by entering PIN.

On platforms with access control enabled, caller needs to have TELUX_TEL_CARD_PRIVILEGED_OPS permission to invoke this API successfully.

Parameters

in	*lockType*	CardLockType. Applicable lock types are PIN1 and PIN2.
in	*pin*	New PIN1/PIN2
in	*callback*	Callback function to get the response of unlock card lock.

4.6.1.1.2.7 virtual telux::common::Status telux::tel::ICardApp::queryPin1LockState (QueryPin1Lock↩ ResponseCb *callback*) [pure virtual]

Query Pin1 lock state.

On platforms with access control enabled, caller needs to have TELUX_TEL_CARD_OPS permission to invoke this API successfully.

Parameters

in	*callback*	Callback function to get the response of query pin1 lock state.

4.6.1.1.2.8 virtual telux::common::Status telux::tel::ICardApp::queryFdnLockState (QueryFdnLock↩ ResponseCb *callback*) [pure virtual]

Query FDN lock state.

On platforms with access control enabled, caller needs to have TELUX_TEL_CARD_OPS permission to invoke this API successfully.

Parameters

in	*callback*	Callback function to get the response of query fdn lock state.

4.6.1.1.2.9 virtual telux::common::Status telux::tel::ICardApp::setCardLock (CardLockType *lockType,* std::string *password,* bool *isEnabled,* PinOperationResponseCb *callback*) [pure virtual]

Enable or disable FDN or Pin1 lock.

On platforms with access control enabled, caller needs to have TELUX_TEL_CARD_PRIVILEGED_OPS permission to invoke this API successfully.

Parameters

in	*lockType*	CardLockType. Applicable lock type such as PIN1 and FDN
in	*password*	Password of PIN1 and FDN
in	*isEnabled*	If true then enable else disable.
in	*callback*	Callback function to get the response of set card lock.

4.6.1.2 struct telux::tel::IccResult

The APDU response with status for transmit APDU operation.

Public member functions

- const std::string toString () const

Data Fields

- int sw1
- int sw2
- std::string payload
- std::vector< int > data

4.6.1.2.1 Member Function Documentation

4.6.1.2.1.1 const std::string telux::tel::IccResult::toString ( ) const

4.6.1.2.2 Field Documentation

4.6.1.2.2.1 int telux::tel::IccResult::sw1

Status word 1 for command processing status

4.6.1.2.2.2 int telux::tel::IccResult::sw2

Status word 2 for command processing qualifier

4.6.1.2.2.3 std::string telux::tel::IccResult::payload

response as a hex string

4.6.1.2.2.4 std::vector<int> telux::tel::IccResult::data

vector of raw data received as part of response to the card services request

4.6.1.3 struct telux::tel::FileAttributes

SIM Elementary file attributes.

Data fields

Type	Field	Description
uint16_t	fileSize	File size of transparent or linear fixed file.
uint16_t	recordSize	Size of the file record. Applicable only for telux::tel::EfType::LINEAR_FIXED.
uint16_t	recordCount	The number of records in a file. Applicable only for telux::tel::EfType::LINEAR_FIXED.

4.6.1.4 class telux::tel::ICardFileHandler

ICardFileHandler provides APIs for reading from an elementary file(EF) on SIM and writing to EF on SIM. Provide API to get EF attributes like file size, record size, and the number of records in EF.

Public member functions

- virtual telux::common::Status readEFLinearFixed (std::string filePath, uint16_t fileId, int recordNum, std::string aid, EfOperationCallback callback)=0
- virtual telux::common::Status readEFLinearFixedAll (std::string filePath, uint16_t fileId, std::string aid, EfReadAllRecordsCallback callback)=0
- virtual telux::common::Status readEFTransparent (std::string filePath, uint16_t fileId, int size, std::string aid, EfOperationCallback callback)=0
- virtual telux::common::Status writeEFLinearFixed (std::string filePath, uint16_t fileId, int recordNum, std::vector< uint8_t > data, std::string pin2, std::string aid, EfOperationCallback callback)=0
- virtual telux::common::Status writeEFTransparent (std::string filePath, uint16_t fileId, std::vector< uint8_t > data, std::string aid, EfOperationCallback callback)=0
- virtual telux::common::Status requestEFAttributes (EfType efType, std::string filePath, uint16_t fileId, std::string aid, EfGetFileAttributesCallback callback)=0
- virtual SlotId getSlotId ()=0

4.6.1.4.1 Member Function Documentation

4.6.1.4.1.1 virtual telux::common::Status telux::tel::ICardFileHandler::readEFLinearFixed (std::string *filePath,* uint16_t *fileId,* int *recordNum,* std::string *aid,* EfOperationCallback *callback*) [pure virtual]

Read a record from a SIM linear fixed elementary file (EF).

Parameters

in	*filePath*	File path of the elementary file to be read Refer ETSI GTS GSM 11.11 V5.3.0 6.5. For example to read EF FDN corresponding to USIM app the file path is "3F007FFF"
in	*fileId*	Elementary file identifier. For example File Id for EF FDN is 0x6F3B
in	*recordNum*	Record number is 1-based (not 0-based)
in	*aid*	Application identifier is optional for reading EF that is not part of card application
in	*callback*	Callback function to get the response of readEFLinearFixed request

Returns

- Status of readEFLinearFixed i.e. success or suitable status code

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backward compatibility.

4.6.1.4.1.2 virtual telux::common::Status telux::tel::ICardFileHandler::readEFLinearFixedAll (std::string *filePath,* uint16_t *fileId,* std::string *aid,* EfReadAllRecordsCallback *callback*) [pure virtual]

Read all records from a SIM linear fixed elementary file (EF).

Parameters

in	*filePath*	File path of the elementary file to be read Refer ETSI GTS GSM 11.11 V5.3.0 6.5. For example to read EF FDN corresponding to USIM app the file path is "3F007FFF"
in	*fileId*	Elementary file identifier. For example File Id for EF FDN is 0x6F3B
in	*aid*	Application identifier is optional for reading EF that is not part of card application
in	*callback*	Callback function to get the response of readEFLinearFixedAll request

Returns

- Status of readEFLinearFixedAll i.e. success or suitable status code

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backward compatibility.

4.6.1.4.1.3 virtual telux::common::Status telux::tel::ICardFileHandler::readEFTransparent (std::string *filePath,* uint16_t *fileId,* int *size,* std::string *aid,* EfOperationCallback *callback*) [pure virtual]

Read from a SIM transparent elementary file (EF).

Parameters

in	*filePath*	File path of the elementary file to be read Refer ETSI GTS GSM 11.11 V5.3.0 6.5. For example to read EF ICCID the file path is "3F00"
in	*fileId*	Elementary file identifier. For example File Id for EF ICCID is 0x2FE2
in	*size*	If the size is zero then read the complete file otherwise, read the first size bytes from EF.
in	*aid*	Application identifier is optional for reading EF that is not part of card application
in	*callback*	Callback function to get the response of readEFTransparent request

Returns

- Status of readEFTransparent i.e. success or suitable status code

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backward compatibility.

4.6.1.4.1.4 virtual telux::common::Status telux::tel::ICardFileHandler::writeEFLinearFixed (std::string *filePath,* uint16_t *fileId,* int *recordNum,* std::vector< uint8_t > *data,* std::string *pin2,* std::string *aid,* EfOperationCallback *callback*) [pure virtual]

Write a record in a SIM linear fixed elementary file (EF).

Parameters

in	*filePath*	File path of the elementary file to be written. Refer ETSI GTS GSM 11.11 V5.3.0 6.5. For example to update record to EF FDN corresponding to USIM app the file path is "3F007FFF"
in	*fileId*	Elementary file identifier. For example File Id for EF FDN is 0x6F3B
in	*recordNum*	Record number is 1-based (not 0-based)
in	*data*	Data represents record in the EF
in	*pin2*	Pin2 for card holder verification(CHV2) operations, otherwise must be empty.
in	*aid*	Application identifier is optional for writing to EF that is not part of card application.

in	*callback*	Callback function to get the response of writeEFLinearFixed request

Returns

- Status of writeEFLinearFixed i.e. success or suitable status code

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backward compatibility.

4.6.1.4.1.5 virtual telux::common::Status telux::tel::ICardFileHandler::writeEFTransparent (std::string *filePath,* uint16_t *fileId,* std::vector< uint8_t > *data,* std::string *aid,* EfOperationCallback *callback*) [pure virtual]

Write in a SIM transparent elementary file (EF).

Parameters

in	*filePath*	File path of the elementary file to be written Refer ETSI GTS GSM 11.11 V5.3.0 6.5. For example to write to EF ICCID the file path is "3F00"
in	*fileId*	Elementary file identifier. For example File Id for EF ICCID is 0x2FE2
in	*data*	Binary data to be written on the EF
in	*aid*	Application identifier is optional for writing to EF that is not part of card application.
in	*callback*	Callback function to get the response of writeEFTransparent request

Returns

- Status of writeEFTransparent i.e. success or suitable status code

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backward compatibility.

4.6.1.4.1.6 virtual telux::common::Status telux::tel::ICardFileHandler::requestEFAttributes (EfType *efType,* std::string *filePath,* uint16_t *fileId,* std::string *aid,* EfGetFileAttributesCallback *callback*) [pure virtual]

Get file attributes for SIM elementary file(EF).

Parameters

in	*efType*	Elementary file type i.e. telux::tel::EfType
in	*filePath*	File path of the elementary file to read file attributes Refer ETSI GTS GSM 11.11 V5.3.0 6.5. For example to read file attributes of EF ICCID the file path is "3F00"
in	*fileId*	Elementary file identifier. For example File Id for EF ICCID is 0x2FE2
in	*aid*	Application identifier is optional for EF that is not part of card application.
in	*callback*	Callback function to get the response of requestEFAttributes request

Returns

- Status of requestEFAttributes i.e. success or suitable status code

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backward compatibility.

4.6.1.4.1.7 virtual SlotId telux::tel::ICardFileHandler::getSlotId ( ) [pure virtual]

Get associated slot identifier for ICardFileHandler

Returns

telux::common::SlotId

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backward compatibility.

4.6.1.5 class telux::tel::ICardManager

ICardManager provide APIs for slot count, retrieve slot ids, get card state and get card.

Public member functions

- virtual bool isSubsystemReady ()=0
- virtual std::future< bool > onSubsystemReady ()=0
- virtual telux::common::Status getSlotCount (int &count)=0
- virtual telux::common::Status getSlotIds (std::vector< int > &slotIds)=0
- virtual std::shared_ptr< ICard > getCard (int slotId=DEFAULT_SLOT_ID, telux::common::Status *status=nullptr)=0

- virtual telux::common::Status cardPowerUp (SlotId slotId, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status cardPowerDown (SlotId slotId, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status registerListener (std::shared_ptr< ICardListener > listener)=0
- virtual telux::common::Status removeListener (std::shared_ptr< ICardListener > listener)=0
- virtual ~ICardManager ()

4.6.1.5.1 Constructors and Destructors

4.6.1.5.1.1 virtual telux::tel::ICardManager::~ICardManager ( ) [virtual]

4.6.1.5.2 Member Function Documentation

4.6.1.5.2.1 virtual bool telux::tel::ICardManager::isSubsystemReady ( ) [pure virtual]

Checks the status of telephony subsystems and returns the result.

Returns

If true then CardManager is ready for service.

4.6.1.5.2.2 virtual std::future<bool> telux::tel::ICardManager::onSubsystemReady ( ) [pure virtual]

Wait for telephony subsystem to be ready.

Returns

A future that caller can wait on to be notified when card manager is ready.

4.6.1.5.2.3 virtual telux::common::Status telux::tel::ICardManager::getSlotCount (int & *count*) [pure virtual]

Get SIM slot count.

Parameters

out	*count*	SIM slot count.

Returns

Status of getSlotCount i.e. success or suitable status code.

4.6.1.5.2.4 virtual telux::common::Status telux::tel::ICardManager::getSlotIds (std::vector< int > & *slotIds*) [pure virtual]

Get list of SIM slots.

Parameters

out	*slotIds*	List of SIM slot ids.

Returns

Status of getSlotIds i.e. success or suitable status code.

4.6.1.5.2.5 virtual std::shared_ptr<ICard> telux::tel::ICardManager::getCard (int *slotId* = *DEFAULT_SLOT_ID*, telux::common::Status * *status* = *nullptr*) [pure virtual]

Get the Card corresponding to SIM slot.

Parameters

in	*slotId*	Slot id corresponding to the card.
out	*status*	Status of getCard i.e. success or suitable status code.

Returns

Pointer to ICard object.

4.6.1.5.2.6 virtual telux::common::Status telux::tel::ICardManager::cardPowerUp (SlotId *slotId*, telux::common::ResponseCallback *callback* = *nullptr*) [pure virtual]

Power on the SIM card.

On platforms with access control enabled, caller needs to have TELUX_TEL_CARD_POWER permission to invoke this API successfully.

Parameters

in	*slotId*	Slot identifier corresponding to the card which needs to be powered up.
in	*callback*	Optional callback pointer to get the result of cardPowerUp

Returns

Status of cardPowerUp i.e. success or suitable status code.

4.6.1.5.2.7 virtual telux::common::Status telux::tel::ICardManager::cardPowerDown (SlotId *slotId*, telux::common::ResponseCallback *callback* = *nullptr*) [pure virtual]

Power off the SIM card. When the SIM card is powered down, the card state is absent and the SIM IO operations, PIN management API's like unlock card by pin, change card pin will fail.

On platforms with access control enabled, caller needs to have TELUX_TEL_CARD_POWER permission to invoke this API successfully.

Parameters

in	*slotId*	Slot identifier corresponding to the card which needs to be powered down.
in	*callback*	Optional callback pointer to get the result of CardPowerDown

Returns

Status of cardPowerDown i.e. success or suitable status code.

4.6.1.5.2.8 virtual telux::common::Status telux::tel::ICardManager::registerListener (std::shared_ptr< ICardListener > *listener*) [pure virtual]

Register a listener for card events.

Parameters

in	*listener*	Pointer to ICardListener object that processes the notification.

Returns

Status of registerListener i.e. success or suitable status code.

4.6.1.5.2.9 virtual telux::common::Status telux::tel::ICardManager::removeListener (std::shared_ptr< ICardListener > *listener*) [pure virtual]

Remove a previously added listener.

Parameters

in	*listener*	Pointer to ICardListener object that needs to be removed.

Returns

Status of removeListener i.e. success or suitable status code.

4.6.1.6 class telux::tel::ICard

ICard represents currently inserted UICC or eUICC.

Public member functions

- virtual telux::common::Status getState (CardState &cardState)=0
- virtual std::vector< std::shared_ptr< ICardApp > > getApplications (telux::common::Status *status=nullptr)=0

- virtual telux::common::Status openLogicalChannel (std::string applicationId, std::shared_ptr< ICardChannelCallback > callback=nullptr)=0
- virtual telux::common::Status closeLogicalChannel (int channelId, std::shared_ptr< telux::common::ICommandResponseCallback > callback=nullptr)=0
- virtual telux::common::Status transmitApduLogicalChannel (int channel, uint8_t cla, uint8_t instruction, uint8_t p1, uint8_t p2, uint8_t p3, std::vector< uint8_t > data, std::shared_ptr< ICardCommandCallback > callback=nullptr)=0
- virtual telux::common::Status transmitApduBasicChannel (uint8_t cla, uint8_t instruction, uint8_t p1, uint8_t p2, uint8_t p3, std::vector< uint8_t > data, std::shared_ptr< ICardCommandCallback > callback=nullptr)=0
- virtual telux::common::Status exchangeSimIO (uint16_t fileId, uint8_t command, uint8_t p1, uint8_t p2, uint8_t p3, std::string filePath, std::vector< uint8_t > data, std::string pin2, std::string aid, std::shared_ptr< ICardCommandCallback > callback=nullptr)=0
- virtual int getSlotId ()=0
- virtual telux::common::Status requestEid (EidResponseCallback callback)=0
- virtual std::shared_ptr< ICardFileHandler > getFileHandler ()=0

4.6.1.6.1 Member Function Documentation

4.6.1.6.1.1 virtual telux::common::Status telux::tel::ICard::getState (CardState & *cardState*) [pure virtual]

Get the card state for the slot id.

Parameters

out	*cardState*	CardState - state of the card.

Returns

Status of getCardState i.e. success or suitable status code.

4.6.1.6.1.2 virtual std::vector<std::shared_ptr<ICardApp> > telux::tel::ICard::getApplications (telux::common::Status * *status = nullptr*) [pure virtual]

Get card applications.

Parameters

out	*status*	Status of getApplications i.e. success or suitable status code.

Returns

List of card applications.

4.6.1.6.1.3 virtual telux::common::Status telux::tel::ICard::openLogicalChannel (std::string *applicationId,* std::shared_ptr< ICardChannelCallback > *callback = nullptr*) [pure virtual]

Open a logical channel to the SIM.

On platforms with access control enabled, caller needs to have TELUX_TEL_CARD_OPS permission to invoke this API successfully.

Parameters

in	*applicationId*	Application Id.
in	*callback*	Optional callback pointer to get the response of open logical channel request.

Returns

Status of openLogicalChannel i.e. success or suitable status code.

4.6.1.6.1.4 virtual telux::common::Status telux::tel::ICard::closeLogicalChannel (int *channelId,* std::shared_ptr< telux::common::ICommandResponseCallback > *callback = nullptr*) [pure virtual]

Close a previously opened logical channel to the SIM.

On platforms with access control enabled, caller needs to have TELUX_TEL_CARD_OPS permission to invoke this API successfully.

Parameters

in	*channelId*	The channel ID to be closed.
in	*callback*	Optional callback pointer to get the response of close logical channel request.

Returns

Status of closeLogicalChannel i.e. success or suitable status code.

4.6.1.6.1.5 virtual telux::common::Status telux::tel::ICard::transmitApduLogicalChannel (int *channel,* uint8_t *cla,* uint8_t *instruction,* uint8_t *p1,* uint8_t *p2,* uint8_t *p3,* std::vector< uint8_t > *data,* std::shared_ptr< ICardCommandCallback > *callback = nullptr*) [pure virtual]

Transmit an APDU to the ICC card over a logical channel.

On platforms with access control enabled, caller needs to have TELUX_TEL_CARD_OPS permission to invoke this API successfully.

Parameters

in	*channel*	Channel Id of the channel to use for communication. Has to be greater than zero.
in	*cla*	Class of the APDU command.
in	*instruction*	Instruction of the APDU command.
in	*p1*	Instruction Parameter 1 value of the APDU command.
in	*p2*	Instruction Parameter 2 value of the APDU command.
in	*p3*	Number of bytes present in the data field of the APDU command. If p3 is negative, a 4 byte APDU is sent to the SIM.
in	*data*	Data to be sent with the APDU.
in	*callback*	Optional callback pointer to get the response of transmit APDU request.

Returns

Status of transmitApduLogicalChannel i.e. success or suitable status code.

4.6.1.6.1.6 virtual telux::common::Status telux::tel::ICard::transmitApduBasicChannel (uint8_t *cla,* uint8_t *instruction,* uint8_t *p1,* uint8_t *p2,* uint8_t *p3,* std::vector< uint8_t > *data,* std::shared_ptr< ICardCommandCallback > *callback = nullptr*) [pure virtual]

Exchange APDUs with the SIM on a basic channel.

On platforms with access control enabled, caller needs to have TELUX_TEL_CARD_OPS permission to invoke this API successfully.

Parameters

in	*cla*	Class of the APDU command.
in	*instruction*	Instruction of the APDU command.
in	*p1*	Instruction Param1 value of the APDU command.
in	*p2*	Instruction Param1 value of the APDU command.
in	*p3*	Number of bytes present in the data field of the APDU command. If p3 is negative, a 4 byte APDU is sent to the SIM.
in	*data*	Data to be sent with the APDU.
in	*callback*	Optional callback pointer to get the response of transmit APDU request.

Returns

Status of transmitApduBasicChannel i.e. success or suitable status code.

4.6.1.6.1.7 virtual telux::common::Status telux::tel::ICard::exchangeSimIO (uint16_t *fileId,* uint8_t *command,* uint8_t *p1,* uint8_t *p2,* uint8_t *p3,* std::string *filePath,* std::vector< uint8_t > *data,* std::string *pin2,* std::string *aid,* std::shared_ptr< ICardCommandCallback > *callback = nullptr*) [pure virtual]

Performs SIM IO operation, This is similar to the TS 27.007 "restricted SIM" operation where it assumes all of the EF selection will be done by the callee.

On platforms with access control enabled, caller needs to have TELUX_TEL_CARD_OPS permission to invoke this API successfully.

Parameters

in	*fileId*	Elementary File Identifier
in	*command*	APDU Command for SIM IO operation
in	*p1*	Instruction Param1 value of the APDU command
in	*p2*	Instruction Param2 value of the APDU command
in	*p3*	Number of bytes present in the data field of APDU command. If p3 is negative, a 4 byte APDU is sent to the SIM.
in	*filePath*	Path of the file
in	*data*	Data to be sent with the APDU, send empty or null string in case no data
in	*pin2*	Pin value of the SIM. Invalid attempt of PIN2 value will lock the SIM. send empty or null string in case of no Pin2 value
in	*aid*	Application identifier, send empty or null string in case of no aid
in	*callback*	Optional callback pointer to get the response of SIM IO request

Returns

- Status of exchangeSimIO i.e. success or suitable status code

4.6.1.6.1.8 virtual int telux::tel::ICard::getSlotId ( ) [pure virtual]

Get associated slot id for ICard

Returns

SlotId

4.6.1.6.1.9 virtual telux::common::Status telux::tel::ICard::requestEid (EidResponseCallback *callback*) [pure virtual]

Request eUICC identifier (EID) of eUICC card.

On platforms with access control enabled, caller needs to have TELUX_TEL_PRIVATE_INFO_READ permission to invoke this API successfully.

Parameters

in	*callback*	Callback function to get the result of request EID.

Returns

Status of request EID i.e. success or suitable error code.

Dependencies Card should be eUICC capable

4.6.1.6.1.10 virtual std::shared_ptr<ICardFileHandler> telux::tel::ICard::getFileHandler ( ) [pure virtual]

Get file handler for reading or writing to EF on SIM.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backward compatibility.

Returns

ICardFileHandler

4.6.1.7 class telux::tel::ICardChannelCallback

Interface for Card callback object. Client needs to implement this interface to get single shot responses for commands like open logical channel and close logical channel.

The methods in callback can be invoked from multiple different threads. The implementation should be thread safe.

Public member functions

- virtual void onChannelResponse (int channel, IccResult result, telux::common::ErrorCode error)=0

4.6.1.7.1 Member Function Documentation

4.6.1.7.1.1 virtual void telux::tel::ICardChannelCallback::onChannelResponse (int *channel,* IccResult *result,* telux::common::ErrorCode *error*) [pure virtual]

This function is called with the response to the open logical channel operation.

Parameters

in	*channel*	Channel Id for the logical channel.
in	*result*	IccResult of open logical channel.
in	*error*	telux::common::ErrorCode of the request.

4.6.1.8 class telux::tel::ICardCommandCallback

Public member functions

- virtual void onResponse (IccResult result, telux::common::ErrorCode error)=0

4.6.1.8.1 Member Function Documentation

4.6.1.8.1.1 virtual void telux::tel::ICardCommandCallback::onResponse (IccResult *result,* telux::common::ErrorCode *error*) [pure virtual]

This function is called when SIM Card transmit APDU over Logical, Basic Channel and Exchange Sim IO.

Parameters

in	*result*	IccResult of transmit APDU command
in	*error*	telux::common::ErrorCode of the request, Possible error codes are • telux::common::ErrorCode::SUCCESS • telux::common::ErrorCode::INTERNAL • telux::common::ErrorCode::NO_MEMORY • telux::common::ErrorCode::INVALID_ARG • telux::common::ErrorCode::MISSING_ARG

4.6.1.9 class telux::tel::ICardListener

Interface for SIM Card Listener object. Client needs to implement this interface to get access to card services notifications on card state change.

The methods in listener can be invoked from multiple different threads. The implementation should be thread safe.

Public member functions

- virtual void onCardInfoChanged (int slotId)
- virtual ~ICardListener ()

4.6.1.9.1 Constructors and Destructors

4.6.1.9.1.1 virtual telux::tel::ICardListener::~ICardListener ( ) [virtual]

4.6.1.9.2 Member Function Documentation

4.6.1.9.2.1 virtual void telux::tel::ICardListener::onCardInfoChanged (int *slotId*) [virtual]

This function is called when info of card gets updated.

Parameters

in	*slotId*	Slot identifier.

4.6.1.10 struct telux::tel::CardReaderStatus

Structure contains identity of card reader status

Data fields

Type	Field	Description
int	id	Card Reader ID
bool	isRemovable	Card reader is removable
bool	isPresent	Card reader is present
bool	isID1size	Card reader present is ID-1 size
bool	isCardPresent	Card is present in reader
bool	isCard↩ PoweredOn	Card in reader is powered

4.6.1.11 class telux::tel::ISapCardManager

ISapCardManager provide APIs for SAP related operations.

Public member functions

- virtual bool isReady ()=0
- virtual std::future< bool > onReady ()=0
- virtual telux::common::Status getState (SapState &sapState)=0
- virtual telux::common::Status requestSapState (SapStateResponseCallback callback)=0
- virtual telux::common::Status openConnection (SapCondition sapCondition=SapCondition::SAP_CONDITION_BLOCK_VOICE_OR_DATA, std::shared_ptr< telux::common::ICommandResponseCallback > callback=nullptr)=0
- virtual telux::common::Status closeConnection (std::shared_ptr< telux::common::ICommandResponseCallback > callback=nullptr)=0
- virtual telux::common::Status requestAtr (std::shared_ptr< IAtrResponseCallback > callback=nullptr)=0
- virtual telux::common::Status transmitApdu (uint8_t cla, uint8_t instruction, uint8_t p1, uint8_t p2, uint8_t lc, std::vector< uint8_t > data, uint8_t le=0, std::shared_ptr< ISapCardCommandCallback > callback=nullptr)=0
- virtual telux::common::Status requestSimPowerOff (std::shared_ptr< telux::common::ICommandResponseCallback > callback=nullptr)=0
- virtual telux::common::Status requestSimPowerOn (std::shared_ptr< telux::common::ICommandResponseCallback > callback=nullptr)=0
- virtual telux::common::Status requestSimReset (std::shared_ptr<

telux::common::ICommandResponseCallback > callback=nullptr)=0

- virtual telux::common::Status requestCardReaderStatus (std::shared_ptr< ICardReaderCallback > callback=nullptr)=0
- virtual int getSlotId ()=0
- virtual telux::common::Status registerListener (std::shared_ptr< ISapCardListener > listener)=0
- virtual telux::common::Status removeListener (std::shared_ptr< ISapCardListener > listener)=0
- virtual ~ISapCardManager ()

4.6.1.11.1 Constructors and Destructors

4.6.1.11.1.1 virtual telux::tel::ISapCardManager::~ISapCardManager ( ) [virtual]

4.6.1.11.2 Member Function Documentation

4.6.1.11.2.1 virtual bool telux::tel::ISapCardManager::isReady ( ) [pure virtual]

Checks the status of SIM access profile(SAP) subsystem and returns the result.

Returns

If true then SapCardManager is ready for service.

4.6.1.11.2.2 virtual std::future<bool> telux::tel::ISapCardManager::onReady ( ) [pure virtual]

Wait for IM access profile(SAP) subsystem to be ready.

Returns

A future that caller can wait on to be notified when card manager is ready.

4.6.1.11.2.3 virtual telux::common::Status telux::tel::ISapCardManager::getState (SapState & *sapState*) [pure virtual]

Get SIM access profile (SAP) client connection state.

Parameters

out	*sapState*	SapState of the SIM Card

Returns

Status of getState i.e. success or suitable status code.

Deprecated

Use requestSapState() API below to get SAP state

4.6.1.11.2.4 virtual telux::common::Status telux::tel::ISapCardManager::requestSapState (SapStateResponseCallback *callback*) [pure virtual]

Get SIM access profile(SAP) client connection state.

Parameters

out	*callback*	Callback function pointer to get the response of requestSapState.

Returns

Status of requestSapState i.e. success or suitable status code.

4.6.1.11.2.5 virtual telux::common::Status telux::tel::ISapCardManager::openConnection (SapCondition *sapCondition* = SapCondition::SAP_CONDITION_BLOCK_VOICE_OR_DATA, std::shared_ptr< telux::common::ICommandResponseCallback > *callback* = *nullptr*) [pure virtual]

Establishes SIM access profile (SAP) client connection with SIM Card.

Parameters

in	*sapCondition*	Condition to enable sap connection.
in	*callback*	Optional callback to get the response of open sap connection request or possible error codes i.e. • telux::common::ErrorCode::SUCCESS • telux::common::ErrorCode::INTERNAL • telux::common::ErrorCode::NO_MEMORY • telux::common::ErrorCode::INVALID_ARG • telux::common::ErrorCode::MISSING_ARG

Returns

Status of openConnection i.e. success or suitable status code.

4.6.1.11.2.6 virtual telux::common::Status telux::tel::ISapCardManager::closeConnection (std::shared_ptr< telux::common::ICommandResponseCallback > *callback* = *nullptr*) [pure virtual]

Releases a SAP connection to SIM Card.

Parameters

in	*callback*	Optional callback to get the response of close sap connection request or possible error codes i.e. • telux::common::ErrorCode::SUCCESS • telux::common::ErrorCode::INTERNAL • telux::common::ErrorCode::NO_MEMORY • telux::common::ErrorCode::INVALID_ARG • telux::common::ErrorCode::MISSING_ARG

Returns

Status of closeConnection i.e. success or suitable status code

4.6.1.11.2.7 virtual telux::common::Status telux::tel::ISapCardManager::requestAtr (std::shared_ptr< IAtrResponseCallback > *callback = nullptr*) [pure virtual]

Request for SAP Answer To Reset command.

Parameters

in	*callback*	Optional callback to get the response of requestAtr.

Returns

Status of requestAtr i.e. success or suitable status code.

4.6.1.11.2.8 virtual telux::common::Status telux::tel::ISapCardManager::transmitApdu (uint8_t *cla,* uint8_t *instruction,* uint8_t *p1,* uint8_t *p2,* uint8_t *lc,* std::vector< uint8_t > *data,* uint8_t *le = 0,* std::shared_ptr< ISapCardCommandCallback > *callback = nullptr*) [pure virtual]

Send the Apdu on SAP mode.

Parameters

in	*cla*	Class of the APDU command.
in	*instruction*	Instruction of the APDU command.
in	*p1*	Instruction Parameter 1 value of the APDU command.
in	*p2*	Instruction Parameter 1 value of the APDU command.
in	*lc*	Number of bytes present in the data field of the APDU command. If lc is negative, a 4 byte APDU is sent to the SIM.
in	*data*	List of data to be sent with the APDU.
in	*le*	Maximum number of bytes expected in the data field of the response to the command.
in	*callback*	Optional callback to send APDU in SAP mode.

Returns

Status of transmitApdu i.e. success or suitable status code.

4.6.1.11.2.9 virtual telux::common::Status telux::tel::ISapCardManager::requestSimPowerOff (std::shared_ptr< telux::common::ICommandResponseCallback > *callback = nullptr*) [pure virtual]

Send the SAP SIM power off request.

Parameters

in	*callback*	Optional callback to get the response for SIM power off.

Returns

Status of requestSimPowerOff i.e. success or suitable status code.

4.6.1.11.2.10 virtual telux::common::Status telux::tel::ISapCardManager::requestSimPowerOn (std::shared_ptr< telux::common::ICommandResponseCallback > *callback = nullptr*) [pure virtual]

Send the SAP SIM power on request.

Parameters

in	*callback*	Optional callback to get the response for SIM power on.

Returns

Status of requestSimPowerOn i.e. success or suitable status code.

4.6.1.11.2.11 virtual telux::common::Status telux::tel::ISapCardManager::requestSimReset (std↩ ::shared_ptr< telux::common::ICommandResponseCallback > *callback = nullptr*) [pure virtual]

Send the SAP SIM reset request.

Parameters

in	*callback*	Optional callback to get the response for SIM reset

Returns

Status of requestSimReset i.e. success or suitable status code.

4.6.1.11.2.12 virtual telux::common::Status telux::tel::ISapCardManager::requestCardReaderStatus (std::shared_ptr< ICardReaderCallback > *callback = nullptr*) [pure virtual]

Send the SAP Card Reader Status request command.

Parameters

in	*callback*	Optional callback to get the response for card reader status

Returns

Status of requestCardReaderStatus i.e. success or suitable status code.

4.6.1.11.2.13 virtual int telux::tel::ISapCardManager::getSlotId ( ) [pure virtual]

Get associated slot id for the SapCardManager.

Returns

SlotId

4.6.1.11.2.14 virtual telux::common::Status telux::tel::ISapCardManager::registerListener (std::shared_ptr< ISapCardListener > *listener*) [pure virtual]

Register a listener for SAP events.

Parameters

in	*listener*	Pointer to ISapCardListener object that processes the notification.

Returns

Status of registerListener i.e. success or suitable status code.

4.6.1.11.2.15 virtual telux::common::Status telux::tel::ISapCardManager::removeListener (std::shared_ptr< ISapCardListener > *listener*) [pure virtual]

Remove a previously added listener.

Parameters

in	*listener*	Pointer to ISapCardListener object that needs to be removed.

Returns

Status of removeListener i.e. success or suitable status code.

4.6.1.12 class telux::tel::IAtrResponseCallback

Public member functions

- virtual void atrResponse (std::vector< int > responseAtr, telux::common::ErrorCode error)=0

4.6.1.12.1 Member Function Documentation

4.6.1.12.1.1 virtual void telux::tel::IAtrResponseCallback::atrResponse (std::vector< int > *responseAtr,* telux::common::ErrorCode *error*) [pure virtual]

This function is called in response to requestAtr() request.

Parameters

in	*responseAtr*	response ATR values
in	*error*	telux::common::ErrorCode of the request possible error codes are • telux::common::ErrorCode::SUCCESS • telux::common::ErrorCode::INTERNAL • telux::common::ErrorCode::NO_MEMORY • telux::common::ErrorCode::INVALID_ARG • telux::common::ErrorCode::MISSING_ARG

4.6.1.13 class telux::tel::ISapCardCommandCallback

Public member functions

- virtual void onResponse (IccResult result, telux::common::ErrorCode error)=0

4.6.1.13.1 Member Function Documentation

4.6.1.13.1.1 virtual void telux::tel::ISapCardCommandCallback::onResponse (IccResult *result,* telux::common::ErrorCode *error*) [pure virtual]

This function is called when SIM Card transmit APDU on SAP mode.

Parameters

in	*result*	IccResult of transmit APDU command
in	*error*	telux::common::ErrorCode of the request, possible error codes are • telux::common::ErrorCode::SUCCESS • telux::common::ErrorCode::INTERNAL • telux::common::ErrorCode::NO_MEMORY • telux::common::ErrorCode::INVALID_ARG • telux::common::ErrorCode::MISSING_ARG

4.6.1.14 class telux::tel::ICardReaderCallback

Public member functions

- virtual void cardReaderResponse (CardReaderStatus cardReaderStatus, telux::common::ErrorCode error)=0

4.6.1.14.1 Member Function Documentation

4.6.1.14.1.1 virtual void telux::tel::ICardReaderCallback::cardReaderResponse (CardReaderStatus *cardReaderStatus,* telux::common::ErrorCode *error*) [pure virtual]

This function is called in response to requestCardReaderStatus() method.

Parameters

in	*cardReaderStatus*	Structure contains the identity of the card reader
in	*error*	telux::common::ErrorCode of the request

4.6.1.15 class telux::tel::ISapCardListener

Interface for SAP Listener object. Client needs to implement this interface to get access to SAP service notifications like service status change.

The methods in listener can be invoked from multiple different threads. The implementation should be thread safe.

Public member functions

- virtual ~ISapCardListener ()

4.6.1.15.1 Constructors and Destructors

4.6.1.15.1.1 virtual telux::tel::ISapCardListener::~ISapCardListener ( ) [virtual]

4.6.2 Enumeration Type Documentation

4.6.2.1 enum telux::tel::CardState [strong]

Defines all state of Card like absent, present etc

Enumerator

CARDSTATE_UNKNOWN Unknown card state
CARDSTATE_ABSENT Card is absent
CARDSTATE_PRESENT Card is present
CARDSTATE_ERROR Card is having error, either card is removed and not readable
CARDSTATE_RESTRICTED Card is present but not usable due to carrier restrictions.

4.6.2.2 enum telux::tel::CardError [strong]

Defines the reasons for error in CardState

Enumerator

UNKNOWN Unknown error
POWER_DOWN Power down
POLL_ERROR Poll error
NO_ATR_RECEIVED No ATR received
VOLT_MISMATCH Volt mismatch
PARITY_ERROR Parity error
POSSIBLY_REMOVED Unknown, possibly removed
TECHNICAL_PROBLEMS Card returned technical problems
NULL_BYTES Card returned NULL bytes
SAP_CONNECTED Terminal in SAP mode
CMD_TIMEOUT Command timeout error

4.6.2.3 enum telux::tel::CardLockType [strong]

Defines all types of card locks which uses in PIN management APIs

Enumerator

PIN1 Lock type is PIN1
PIN2 Lock type is PIN2
PUK1 Lock type is Pin Unblocking Key1
PUK2 Lock type is Pin Unblocking Key2
FDN Lock type is Fixed Dialing Number

4.6.2.4 enum telux::tel::AppType

Defines all type of UICC application such as SIM, RUIM, USIM, CSIM and ISIM.

Enumerator

APPTYPE_UNKNOWN Unknown application type
APPTYPE_SIM UICC application type is SIM
APPTYPE_USIM UICC application type is USIM
APPTYPE_RUIM UICC application type is RSIM
APPTYPE_CSIM UICC application type is CSIM
APPTYPE_ISIM UICC application type is ISIM

4.6.2.5 enum telux::tel::AppState

Defines all application states.

Enumerator

APPSTATE_UNKNOWN Unknown application state
APPSTATE_DETECTED application state detected
APPSTATE_PIN If PIN1 or UPin is required

APPSTATE_PUK If PUK1 or Puk for UPin is required
APPSTATE_SUBSCRIPTION_PERSO PersoSubstate should be look at when application state is assigned to this value
APPSTATE_READY application State is ready

4.6.2.6 enum telux::tel::EfType [strong]

Defines supported elementary file(EF) types.

Enumerator

UNKNOWN Unknown EF type
TRANSPARENT Transparent EF
LINEAR_FIXED Linear Fixed EF

4.6.2.7 enum telux::tel::SapState [strong]

Defines all SIM access profile (SAP) connection states.

Enumerator

SAP_STATE_NOT_ENABLED SAP connection not enabled
SAP_STATE_CONNECTING SAP State is connecting
SAP_STATE_CONNECTED_SUCCESSFULLY SAP connection is successful
SAP_STATE_CONNECTION_ERROR SAP connection error
SAP_STATE_DISCONNECTING SAP state is disconnecting
SAP_STATE_DISCONNECTED_SUCCESSFULLY SAP state disconnection is successful

4.6.2.8 enum telux::tel::SapCondition [strong]

Indicates type of connection required, default behavior is to block a SAP connection when a voice or data call is active.

Enumerator

SAP_CONDITION_BLOCK_VOICE_OR_DATA Block a SAP connection when a voice or data call is active (Default)
SAP_CONDITION_BLOCK_DATA Block a SAP connection when a data call is active
SAP_CONDITION_BLOCK_VOICE Block a SAP connection when a voice call is active
SAP_CONDITION_BLOCK_NONE Allow Sap connection in all cases

4.7 Cell Broadcast

This section contains APIs related to configure, activate and receive 3GPP ETWS/CMAS cell broadcast messages.

4.7.1 Data Structure Documentation

4.7.1.1 struct telux::tel::CellBroadcastFilter

Defines cellbroadcast message filter. Refer spec 3GPP TS 23.041 9.4.1.2.2 for message identifier. Eg: If user want to receive from 0x1112 to 0x1116 then, startMessageId is 0x1112 and endMessageId is 0x1116. If user want to receive only 0x1112, then both startMessageId and endMessageId is 0x1112.

Data fields

Type	Field	Description
int	startMessageId	Intended to receive start from which MessageType
int	endMessageId	Intended to receive upto which MessageType

4.7.1.2 struct telux::tel::Point

Point represented by latitude and longitude.

Data fields

Type	Field	Description
double	latitude	
double	longitude	

4.7.1.3 class telux::tel::Polygon

This class represents a simple polygon with different points.

Public member functions

- Polygon (std::vector< Point > vertices)
- std::vector< Point > getVertices ()

4.7.1.3.1 Constructors and Destructors

4.7.1.3.1.1 telux::tel::Polygon::Polygon (std::vector< Point > *vertices*)

Polygon constructor.

Parameters

in	*vertices*	List of telux::tel::Point

4.7.1.3.2 Member Function Documentation

4.7.1.3.2.1 std::vector<Point> telux::tel::Polygon::getVertices ( )

Get vertices of polygon.

Returns

List of telux::tel::Point.

4.7.1.4 class telux::tel::Circle

This class represents a geometry represented as simple circle.

Public member functions

- Circle (Point center, double radius)
- Point getCenter ()
- double getRadius ()

4.7.1.4.1 Constructors and Destructors

4.7.1.4.1.1 telux::tel::Circle::Circle (Point *center,* double *radius*)

Circle constructor.

Parameters

in	*center*	Center of circle represented by telux::tel::Point
in	*radius*	Radius of circle in meters

4.7.1.4.2 Member Function Documentation

4.7.1.4.2.1 Point telux::tel::Circle::getCenter ( )

Get center point of circle.

Returns

Center of circle.

4.7.1.4.2.2 double telux::tel::Circle::getRadius ( )

Get radius of circle.

Returns

Radius of circle.

4.7.1.5 class telux::tel::Geometry

This class represents warning area geometry to perform geofencing on alert.

Public member functions

- Geometry (std::shared_ptr< Polygon > polygon)
- Geometry (std::shared_ptr< Circle > circle)
- GeometryType getType () const
- std::shared_ptr< Polygon > getPolygon () const
- std::shared_ptr< Circle > getCircle () const

4.7.1.5.1 Constructors and Destructors

4.7.1.5.1.1 telux::tel::Geometry::Geometry (std::shared_ptr< Polygon > *polygon*)

Geometry constructor.

Parameters

in	*polygon*	Polygon

4.7.1.5.1.2 telux::tel::Geometry::Geometry (std::shared_ptr< Circle > *circle*)

Geometry constructor.

Parameters

in	*circle*	Circle

4.7.1.5.2 Member Function Documentation

4.7.1.5.2.1 GeometryType telux::tel::Geometry::getType ( ) const

Get the geometry type.

Returns

GeometryType.

4.7.1.5.2.2 std::shared_ptr<Polygon> telux::tel::Geometry::getPolygon ( ) const

Get polygon goemetry as warning area to perform geofencing. This method should be called only if geometry type returned by getType() API is GeometryType::POLYGON

Returns

Polygon geometry object.

4.7.1.5.2.3 std::shared_ptr<Circle> telux::tel::Geometry::getCircle ( ) const

Get circle goemetry as warning area to perform geofencing. This method should be called only if geometry type returned by getType() API is GeometryType::CIRCLE

Returns

Circle geometry object.

4.7.1.6 class telux::tel::WarningAreaInfo

This class represents warning area information for alert.

Public member functions

- WarningAreaInfo (int maxWaitTime, std::vector< Geometry > geometries)
- int getGeoFenceMaxWaitTime ()
- std::vector< Geometry > getGeometries ()

4.7.1.6.1 Constructors and Destructors

4.7.1.6.1.1 telux::tel::WarningAreaInfo::WarningAreaInfo (int *maxWaitTime,* std::vector< Geometry > *geometries*)

Warning Area Information constructor.

Parameters

in	*maxWaitTime*	Maximum wait time allowed to determine position for alert Range is 0 to 255 where 0 means Zero wait time, 1 - 254 is Geo-Fencing Maximum Wait Time in seconds and 255 means use device default wait time.
in	*geometries*	Geometries to perform geofencing on alert

4.7.1.6.2 Member Function Documentation

4.7.1.6.2.1 int telux::tel::WarningAreaInfo::getGeoFenceMaxWaitTime ( )

Get maximum wait time allowed to determine position for alert.

Returns

Maximum wait time for alert in seconds.

4.7.1.6.2.2 std::vector<Geometry> telux::tel::WarningAreaInfo::getGeometries ( )

Get geometries to perform geofencing on alert.

Returns

List of telux::tel::Geometry.

4.7.1.7 class telux::tel::EtwsInfo

Contains information elements for a GSM/UMTS/E-UTRAN/NG-RAN ETWS warning notification. Supported values for each element are defined in 3GPP TS 23.041.

Public member functions

- EtwsInfo (GeographicalScope geographicalScope, int msgId, int serialNumber, std::string languageCode, std::string messageText, MessagePriority priority, EtwsWarningType warningType, bool emergencyUserAlert, bool activatePopup, bool primary, std::vector< uint8_t > warningSecurityInformation)
- GeographicalScope getGeographicalScope () const
- int getMessageId () const
- int getSerialNumber () const
- std::string getLanguageCode () const
- std::string getMessageBody () const
- MessagePriority getPriority () const
- int getMessageCode () const
- int getUpdateNumber () const
- EtwsWarningType getEtwsWarningType ()
- bool isEmergencyUserAlert ()
- bool isPopupAlert ()
- bool isPrimary ()
- std::vector< uint8_t > getWarningSecurityInformation ()

4.7.1.7.1 Constructors and Destructors

4.7.1.7.1.1 telux::tel::EtwsInfo::EtwsInfo (GeographicalScope *geographicalScope*, int *msgId*, int *serialNumber*, std::string *languageCode*, std::string *messageText*, MessagePriority *priority*, EtwsWarningType *warningType*, bool *emergencyUserAlert*, bool *activatePopup*, bool *primary*, std::vector< uint8_t > *warningSecurityInformation*)

EtwsInfo constructor.

Parameters

in	*geographicalScope*	GeographicalScope
in	*msgId*	Unique message identifier
in	*serialNumber*	Serial number for message
in	*languageCode*	ISO-639-1 language code for message
in	*messageText*	Message text
in	*priority*	MessagePriority
in	*warningType*	EtwsWarningType
in	*emergencyUserAlert*	If true message is emergency user alert otherwise not
in	*activatePopup*	If true message message activate popup flag is set, otherwise popup flag is false.
in	*primary*	If true ETWS message is primary notification otherwise not
in	*warningSecurity↩ Information*	Buffer containing security information about ETWS primary notification such as timestamp and digital signature

4.7.1.7.2 Member Function Documentation

4.7.1.7.2.1 GeographicalScope telux::tel::EtwsInfo::getGeographicalScope ( ) const

Get the geographicalScope of cellbroadcast message.

Returns

GeographicalScope.

4.7.1.7.2.2 int telux::tel::EtwsInfo::getMessageId ( ) const

Get cellbroadcast message identifier. The message identifier identifies the type of the cell broadcast message defined in spec 3GPP TS 23.041 9.4.1.2.2

Returns

Message identifier.

4.7.1.7.2.3 int telux::tel::EtwsInfo::getSerialNumber ( ) const

Get the serial number of broadcast (geographical scope + message code + update number for GSM/UMTS).

Returns

int containing cellbroadcast serial number.

4.7.1.7.2.4 std::string telux::tel::EtwsInfo::getLanguageCode ( ) const

Get the ISO-639-1 language code for cell broadcast message, or empty string if unspecified. This is not applicable for ETWS primary notification.

Returns

Language code

4.7.1.7.2.5 std::string telux::tel::EtwsInfo::getMessageBody ( ) const

Get the body of cell broadcast message, or empty string if no body available. For ETWS primary notification based on message identifier pre cannned message will be sent.

Returns

body or empty string

4.7.1.7.2.6 MessagePriority telux::tel::EtwsInfo::getPriority ( ) const

Get the priority for the cell broadcast message.

Returns

MessagePriority.

4.7.1.7.2.7 int telux::tel::EtwsInfo::getMessageCode ( ) const

Get the cellbroadcast message code.

Returns

int containing cellbroadcast message code.

4.7.1.7.2.8 int telux::tel::EtwsInfo::getUpdateNumber ( ) const

Get the cellbroadcast message update number.

Returns

int containing cellbroadcast message's update number.

4.7.1.7.2.9 EtwsWarningType telux::tel::EtwsInfo::getEtwsWarningType ( )

Get ETWS warning type.

Returns

EtwsWarningType.

4.7.1.7.2.10 bool telux::tel::EtwsInfo::isEmergencyUserAlert ( )

Returns the ETWS emergency user alert flag.

Returns

true to notify terminal to activate emergency user alert or false otherwise

4.7.1.7.2.11 bool telux::tel::EtwsInfo::isPopupAlert ( )

Returns the ETWS activate popup flag.

Returns

true to notify terminal to activate display popup or false otherwise

4.7.1.7.2.12 bool telux::tel::EtwsInfo::isPrimary ( )

Returns the ETWS format flag. This flag determine whether ETWS message is primary notification or not.

Returns

true if the message is primary message, otherwise secondary message

4.7.1.7.2.13 std::vector<uint8_t> telux::tel::EtwsInfo::getWarningSecurityInformation ( )

Returns security information about ETWS primary notification such as timestamp and digital signature(applicable only for GSM).

Returns

byte buffer

4.7.1.8 class telux::tel::CmasInfo

Contains information elements for a GSM/UMTS/E-UTRAN/NG-RAN CMAS warning notification. Supported values for each element are defined in 3GPP TS 23.041.

Public member functions

- CmasInfo (GeographicalScope geographicalScope, int msgId, int serialNumber, std::string languageCode, std::string messageText, MessagePriority priority, CmasMessageClass messageClass, CmasSeverity severity, CmasUrgency urgency, CmasCertainty certainty, std::shared_ptr< WarningAreaInfo > warningAreaInfo)
- GeographicalScope getGeographicalScope () const
- int getMessageId () const
- int getSerialNumber () const
- std::string getLanguageCode () const
- std::string getMessageBody () const
- MessagePriority getPriority () const
- int getMessageCode () const
- int getUpdateNumber () const
- CmasMessageClass getMessageClass ()
- CmasSeverity getSeverity ()
- CmasUrgency getUrgency ()
- CmasCertainty getCertainty ()
- std::shared_ptr< WarningAreaInfo > getWarningAreaInfo ()

4.7.1.8.1 Constructors and Destructors

4.7.1.8.1.1 telux::tel::CmasInfo::CmasInfo (GeographicalScope *geographicalScope,* int *msgId,* int *serialNumber,* std::string *languageCode,* std::string *messageText,* MessagePriority *priority,* CmasMessageClass *messageClass,* CmasSeverity *severity,* CmasUrgency *urgency,* CmasCertainty *certainty,* std::shared_ptr< WarningAreaInfo > *warningAreaInfo*)

CmasInfo constructor.

Parameters

in	*geographicalScope*	GeographicalScope
in	*msgId*	Unique message identifier
in	*serialNumber*	Serial number for message
in	*languageCode*	ISO-639-1 language code for message
in	*messageText*	Message text
in	*priority*	MessagePriority
in	*messageClass*	CmasMessageClass
in	*severity*	CmasSeverity
in	*urgency*	CmasUrgency
in	*certainty*	CmasCertainty
in	*warningAreaInfo*	WarningAreaInfo

4.7.1.8.2 Member Function Documentation

4.7.1.8.2.1 GeographicalScope telux::tel::CmasInfo::getGeographicalScope ( ) const

Get the geographicalScope of cellbroadcast message.

Returns

GeographicalScope.

4.7.1.8.2.2 int telux::tel::CmasInfo::getMessageId ( ) const

Get cellbroadcast message identifier. The message identifier identifies the type of the cell broadcast message defined in spec 3GPP TS 23.041 9.4.1.2.2

Returns

Message identifier.

4.7.1.8.2.3 int telux::tel::CmasInfo::getSerialNumber ( ) const

Get the serial number of broadcast (geographical scope + message code + update number for GSM/UMTS).

Returns

int containing cellbroadcast serial number.

4.7.1.8.2.4 std::string telux::tel::CmasInfo::getLanguageCode ( ) const

Get the ISO-639-1 language code for cell broadcast message, or empty string if unspecified. This is not applicable for ETWS primary notification.

Returns

Language code

4.7.1.8.2.5 std::string telux::tel::CmasInfo::getMessageBody ( ) const

Get the body of cell broadcast message, or empty string if no body available. For ETWS primary notification based on message identifier pre cannned message will be sent.

Returns

body or empty string

4.7.1.8.2.6 MessagePriority telux::tel::CmasInfo::getPriority ( ) const

Get the priority for the cell broadcast message.

Returns

MessagePriority.

4.7.1.8.2.7 int telux::tel::CmasInfo::getMessageCode ( ) const

Get the cellbroadcast message code.

Returns

int containing cellbroadcast message code.

4.7.1.8.2.8 int telux::tel::CmasInfo::getUpdateNumber ( ) const

Get the cellbroadcast message update number.

Returns

int containing cellbroadcast message's update number.

4.7.1.8.2.9 CmasMessageClass telux::tel::CmasInfo::getMessageClass ( )

Get CMAS message class.

Returns

CmasMessageClass.

4.7.1.8.2.10 CmasSeverity telux::tel::CmasInfo::getSeverity ( )

Get CMAS message severity.

Returns

CmasSeverity.

4.7.1.8.2.11 CmasUrgency telux::tel::CmasInfo::getUrgency ( )

Get CMAS message urgency.

Returns

CmasUrgency.

4.7.1.8.2.12 CmasCertainty telux::tel::CmasInfo::getCertainty ( )

Get CMAS message certainty.

Returns

CmasCertainty.

4.7.1.8.2.13 std::shared_ptr<WarningAreaInfo> telux::tel::CmasInfo::getWarningAreaInfo ( )

Returns warning area information for alert. This is applicable for LTE and NR5G

Returns

pointer to WarningAreaInfo or null if there is no warning area information available.

4.7.1.9 class telux::tel::CellBroadcastMessage

Cell Broadcast message.

Public member functions

- CellBroadcastMessage (std::shared_ptr< EtwsInfo > etwsInfo)
- CellBroadcastMessage (std::shared_ptr< CmasInfo > cmasInfo)
- MessageType getMessageType () const
- std::shared_ptr< EtwsInfo > getEtwsInfo () const
- std::shared_ptr< CmasInfo > getCmasInfo () const

4.7.1.9.1 Constructors and Destructors

4.7.1.9.1.1 telux::tel::CellBroadcastMessage::CellBroadcastMessage (std::shared_ptr< EtwsInfo > *etwsInfo*)

CellBroadcastMessage constructor.

Parameters

in	*etwsInfo*	EtwsInfo

4.7.1.9.1.2 telux::tel::CellBroadcastMessage::CellBroadcastMessage (std::shared_ptr< CmasInfo > *cmasInfo*)

CellBroadcastMessage constructor.

Parameters

in	*cmasInfo*	CmasInfo

4.7.1.9.2 Member Function Documentation

4.7.1.9.2.1 MessageType telux::tel::CellBroadcastMessage::getMessageType ( ) const

Get the cellbroadcast message type.

Returns

MessageType.

4.7.1.9.2.2 std::shared_ptr<EtwsInfo> telux::tel::CellBroadcastMessage::getEtwsInfo ( ) const

Get ETWS warning notification containing information about the ETWS warning type, the emergency user alert flag and the popup flag. This method should be called only if message type returned by getMessageType() API is MessageType::ETWS

Returns

pointer to EtwsInfo or null if this is not an ETWS warning notification

4.7.1.9.2.3 std::shared_ptr<CmasInfo> telux::tel::CellBroadcastMessage::getCmasInfo ( ) const

Get CMAS warning notification containing information about the CMAS message class, severity, urgency and certainty.This method should be called only if message type returned by getMessageType() API is MessageType::CMAS

Returns

pointer to CmasInfo or null if this is not a CMAS warning notification

4.7.1.10 class telux::tel::ICellBroadcastManager

CellBroadcastManager class is primary interface to configure and activate emergency broadcast messages and receive broadcast messages.

Public member functions

- virtual bool isSubsystemReady ()=0
- virtual std::future< bool > onSubsystemReady ()=0
- virtual SlotId getSlotId ()=0
- virtual telux::common::Status updateMessageFilters (std::vector< CellBroadcastFilter > filters, telux::common::ResponseCallback callback=nullptr)=0

- virtual telux::common::Status requestMessageFilters (RequestFiltersResponseCallback callback)=0
- virtual telux::common::Status setActivationStatus (bool activate, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestActivationStatus (RequestActivationStatusResponseCallback callback)=0
- virtual telux::common::Status registerListener (std::weak_ptr< ICellBroadcastListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< ICellBroadcastListener > listener)=0
- virtual ~ICellBroadcastManager ()

4.7.1.10.1 Constructors and Destructors

4.7.1.10.1.1 virtual telux::tel::ICellBroadcastManager::~ICellBroadcastManager ( ) [virtual]

4.7.1.10.2 Member Function Documentation

4.7.1.10.2.1 virtual bool telux::tel::ICellBroadcastManager::isSubsystemReady ( ) [pure virtual]

Checks the status of network subsystem and returns the result.

Returns

True if network subsystem is ready for service otherwise false.

4.7.1.10.2.2 virtual std::future<bool> telux::tel::ICellBroadcastManager::onSubsystemReady ( ) [pure virtual]

Wait for network subsystem to be ready.

Returns

A future that caller can wait on to be notified when network subsystem is ready.

4.7.1.10.2.3 virtual SlotId telux::tel::ICellBroadcastManager::getSlotId ( ) [pure virtual]

Get associated slot for this CellBroadcastManager.

Returns

SlotId

4.7.1.10.2.4 virtual telux::common::Status telux::tel::ICellBroadcastManager::updateMessageFilters (std::vector< CellBroadcastFilter > *filters,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

Configures the broadcast messages to be received.

On platforms with access control enabled, caller needs to have TELUX_TEL_CELL_BROADCAST_CONFIG permission to invoke this API successfully.

Parameters

in	*filters*	List of filtered broadcast message identifiers.
in	*callback*	Optional callback to get the response of set cell broadcast filters.

Returns

Status of updateMessageIdFilters i.e. success or suitable error code.

4.7.1.10.2.5 virtual telux::common::Status telux::tel::ICellBroadcastManager::requestMessageFilters (RequestFiltersResponseCallback *callback*) [pure virtual]

Retrieve configured message filters for which broadcast messages will be received.

On platforms with access control enabled, caller needs to have TELUX_TEL_CELL_BROADCAST_CONFIG permission to invoke this API successfully.

Parameters

in	*callback*	Callback to get the response of get cell broadcast filters.

Returns

Status of requestMessageIdFilters i.e. success or suitable error code.

4.7.1.10.2.6 virtual telux::common::Status telux::tel::ICellBroadcastManager::setActivationStatus (bool *activate,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

Allows activation and deactivation of configured broadcast messages.

On platforms with access control enabled, caller needs to have TELUX_TEL_CELL_BROADCAST_CONFIG permission to invoke this API successfully.

Parameters

in	*activate*	Activate/deactivate broadcast messages.
in	*callback*	Optional callback pointer to get the response.

Returns

Status of setActivationStatus i.e. success or suitable error code.

4.7.1.10.2.7 virtual telux::common::Status telux::tel::ICellBroadcastManager::requestActivationStatus (RequestActivationStatusResponseCallback *callback*) [pure virtual]

Get activation status for configured broadcast messages.

On platforms with access control enabled, caller needs to have TELUX_TEL_CELL_BROADCAST_CONFIG permission to invoke this API successfully.

Parameters

in	*callback*	Callback pointer to get the response.

Returns

Status of requestActivationStatus i.e. success or suitable error code.

4.7.1.10.2.8 virtual telux::common::Status telux::tel::ICellBroadcastManager::registerListener (std::weak_ptr< ICellBroadcastListener > *listener*) [pure virtual]

Register a listener for cell broadcast messages.

Parameters

in	*listener*	Pointer to ICellBroadcastListener object which receives broadcast message.

Returns

Status of registerListener i.e. success or suitable error code.

4.7.1.10.2.9 virtual telux::common::Status telux::tel::ICellBroadcastManager::deregisterListener (std::weak_ptr< ICellBroadcastListener > *listener*) [pure virtual]

De-register the listener.

Parameters

in	*listener*	Listener to be de-registered

Returns

Status of deregisterListener i.e. success or suitable error code.

4.7.1.11 class telux::tel::ICellBroadcastListener

A listener class which monitors cell broadcast messages.

The methods in listener can be invoked from multiple different threads. The implementation should be thread safe.

Public member functions

- virtual void onIncomingMessage (SlotId slotId, const std::shared_ptr< CellBroadcastMessage > cbMessage)
- virtual void onMessageFilterChange (SlotId slotId, std::vector< CellBroadcastFilter > filters)
- virtual ~ICellBroadcastListener ()

4.7.1.11.1 Constructors and Destructors

4.7.1.11.1.1 virtual telux::tel::ICellBroadcastListener::~ICellBroadcastListener ( ) [virtual]

4.7.1.11.2 Member Function Documentation

4.7.1.11.2.1 virtual void telux::tel::ICellBroadcastListener::onIncomingMessage (SlotId *slotId*, const std::shared_ptr< CellBroadcastMessage > *cbMessage*) [virtual]

This function is called when device receives an incoming cell broadcast message.

On platforms with access control enabled, the client needs to have TELUX_TEL_CELL_BROADCAST_LISTEN permission to invoke this API successfully.

Parameters

in	*slotId*	- Slot Id on which broadcast message is received.
in	*cbMessage*	- Broadcast message with information related to ETWS/CMAS notification.

4.7.1.11.2.2 virtual void telux::tel::ICellBroadcastListener::onMessageFilterChange (SlotId *slotId*, std::vector< CellBroadcastFilter > *filters*) [virtual]

This function is called when there is change in broadcast configuration like updation of message filters by the client using ICellBroadcastManager::updateMessageFilters.

On platforms with access control enabled, the client needs to have TELUX_TEL_CELL_BROADCAST_CONFIG permission to invoke this API successfully.

Parameters

in	*slotId*	- Slot Id on which change in message filters is received.
in	*filters*	- Complete list of configured broadcast message filters.

4.7.2 Enumeration Type Documentation

4.7.2.1 enum telux::tel::GeographicalScope [strong]

Defines geographical scope of cell broadcast.

Enumerator

CELL_WIDE_IMMEDIATE Cell wide geographical scope with immediate display
PLMN_WIDE PLMN wide geographical scope
LA_WIDE
CELL_WIDE Location / Service/ Tracking area wide geographical scope (GSM/UMTS/E-UTRAN/NG-RAN). Cell wide geographical scope

4.7.2.2 enum telux::tel::MessagePriority [strong]

Defines priority for cell broadcast message.

Enumerator

UNKNOWN Unknown message priority
NORMAL Normal message priority
EMERGENCY Emergency message priority

4.7.2.3 enum telux::tel::MessageType [strong]

Defines message type for cell broadcast message.

Enumerator

UNKNOWN Unknown message type
ETWS Earthquake and Tsunami Warning System
CMAS Commercial Mobile Alert System

4.7.2.4 enum telux::tel::EtwsWarningType [strong]

Defines warning type for ETWS cell broadcast message.

Enumerator

UNKNOWN Unknown ETWS warning type
EARTHQUAKE ETWS warning type for earthquake
TSUNAMI ETWS warning type for tsunami
EARTHQUAKE_AND_TSUNAMI ETWS warning type for earthquake and tsunami
TEST_MESSAGE ETWS warning type for test messages
OTHER_EMERGENCY ETWS warning type for other emergency types

4.7.2.5 enum telux::tel::CmasMessageClass [strong]

Defines message class for CMAS cell broadcast message.

Enumerator

UNKNOWN CMAS category for warning types that are reserved for future extension
PRESIDENTIAL_LEVEL_ALERT Presidential-level alert (Korean Public Alert System Class 0 message)
EXTREME_THREAT Extreme threat to life and property (Korean Public Alert System Class 1 message)

SEVERE_THREAT Severe threat to life and property (Korean Public Alert System Class 1 message).
CHILD_ABDUCTION_EMERGENCY Child abduction emergency (AMBER Alert)
REQUIRED_MONTHLY_TEST CMAS test message
CMAS_EXERCISE CMAS exercise
OPERATOR_DEFINED_USE CMAS category for operator defined use

4.7.2.6 enum telux::tel::CmasSeverity [strong]

Defines severity type for CMAS cell broadcast message.

Enumerator

UNKNOWN CMAS alert severity is unknown. The severity is available for all GSM/UMTS alerts except for the Presidential-level alert class (Korean Public Alert System Class 0).
EXTREME Extraordinary threat to life or property
SEVERE Significant threat to life or property

4.7.2.7 enum telux::tel::CmasUrgency [strong]

Defines urgency type for CMAS cell broadcast message.

Enumerator

UNKNOWN CMAS alert urgency is unknown. The urgency is available for all GSM/UMTS alerts except for the Presidential-level alert class (Korean Public Alert System Class 0).
IMMEDIATE Responsive action should be taken immediately
EXPECTED Responsive action should be taken within the next hour

4.7.2.8 enum telux::tel::CmasCertainty [strong]

Defines certainty type for CMAS cell broadcast message.

Enumerator

UNKNOWN CMAS alert certainty is unknown. The certainty is available for all GSM/UMTS alerts except for the Presidential-level alert class (Korean Public Alert System Class 0).
OBSERVED Determined to have occurred or to be ongoing.
LIKELY Likely (probability $> \sim 50\%$)

4.7.2.9 enum telux::tel::GeometryType [strong]

Defines geometry type specified in wireless emergency alert.

Enumerator

UNKNOWN Unknown geometry type
POLYGON Polygon geometry type
CIRCLE Circle geometry type

4.8 IMS Settings

This section contains APIs related to IMS Configuration.

4.8.1 Data Structure Documentation

4.8.1.1 struct telux::tel::ImsServiceConfig

Defines the selected IMS service configuration parameters and their corresponding value

Data fields

Type	Field	Description
ImsService↩ ConfigValidity	config↩ ValidityMask	Indicates the configurations type. Bit set to 1 denotes the config is valid.
bool	imsService↩ Enabled	Enable/Disable IMS service
bool	voImsEnabled	Enable/Disable VOIMS service
bool	smsEnabled	Enable/Disable SMS service

4.8.1.2 class telux::tel::IImsSettingsManager

ImsSettingsManager allows IMS settings. For example enabling or disabling IMS service, VOIMS service.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual telux::common::Status requestServiceConfig (SlotId slotId, ImsServiceConfigCb callback)=0
- virtual telux::common::Status setServiceConfig (SlotId slotId, ImsServiceConfig config, common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status registerListener (std::weak_ptr< IImsSettingsListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< IImsSettingsListener > listener)=0
- virtual ~IImsSettingsManager ()

4.8.1.2.1 Constructors and Destructors

4.8.1.2.1.1 virtual telux::tel::IImsSettingsManager::~IImsSettingsManager ( ) [virtual]

4.8.1.2.2 Member Function Documentation

4.8.1.2.2.1 virtual telux::common::ServiceStatus telux::tel::IImsSettingsManager::getServiceStatus ( ) [pure virtual]

This status indicates whether the IImsSettingsManager object is in a usable state.

Returns

SERVICE_AVAILABLE - If IMS settings manager is ready for service. SERVICE_UNAVAILABLE - If IMS settings manager is temporarily unavailable. SERVICE_FAILED - If IMS settings manager encountered an irrecoverable failure.

4.8.1.2.2.2 virtual telux::common::Status telux::tel::IImsSettingsManager::requestServiceConfig (SlotId *slotId,* ImsServiceConfigCb *callback*) [pure virtual]

Request the IMS service configurations

Parameters

in	*slotId*	Slot for which the IMS service configurations is requested.
in	*callback*	Callback function to get the response of request IMS service configurations.

Returns

Status of requestServiceConfig i.e. success or suitable error code.

4.8.1.2.2.3 virtual telux::common::Status telux::tel::IImsSettingsManager::setServiceConfig (SlotId *slotId,* ImsServiceConfig *config,* common::ResponseCallback *callback* = `nullptr`) [pure virtual]

To configure the IMS service configurations. Also specifiy whether configuration needs to be enabled or disabled.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_IMS_SETTINGS permission to invoke this API successfully.

Parameters

in	*slotId*	Slot for which the IMS service configuration is intended.
in	*config*	Indicates which configuration are configured currently and whether the config is enabled or disabled. ImsServiceConfig.
in	*callback*	Callback function to get the response of set IMS service configuration request.

Returns

Status of setServiceConfig i.e. success or suitable error code.

4.8.1.2.2.4 virtual telux::common::Status telux::tel::IImsSettingsManager::registerListener (std::weak_ptr< IImsSettingsListener > *listener*) [pure virtual]

Register a listener for specific events in the IMS settings subsystem.

Parameters

in	*listener*	Pointer to IImsSettingsListener object that processes the notification

Returns

Status of registerListener i.e. success or suitable error code.

4.8.1.2.2.5 virtual telux::common::Status telux::tel::IImsSettingsManager::deregisterListener (std::weak_ptr< IImsSettingsListener > *listener*) [pure virtual]

Deregister the previously added listener.

Parameters

in	*listener*	Pointer to IImsSettingsListener object that needs to be deregistered.

Returns

Status of deregisterListener i.e. success or suitable error code.

4.8.1.3 class telux::tel::IImsSettingsListener

Listener class for getting IMS service configuration change notifications. The listener method can be invoked from multiple different threads. Client needs to make sure that implementation is thread-safe.

Public member functions

- virtual void onImsServiceConfigsChange (SlotId slotId, ImsServiceConfig config)
- virtual void onServiceStatusChange (telux::common::ServiceStatus status)
- virtual ~IImsSettingsListener ()

4.8.1.3.1 Constructors and Destructors

4.8.1.3.1.1 virtual telux::tel::IImsSettingsListener::~IImsSettingsListener ( ) [virtual]

Destructor of IImsSettingsListener

4.8.1.3.2 Member Function Documentation

4.8.1.3.2.1 virtual void telux::tel::IImsSettingsListener::onImsServiceConfigsChange (SlotId *slotId,* ImsServiceConfig *config*) [virtual]

This function is called whenever any IMS service configuration is changed.

Parameters

in	*slotId*	SIM corresponding to slot identifier for which the IMS service configuration has changed.
in	*config*	Indicates which configuration is valid and whether the config is enabled or disabled. ImsServiceConfig.

4.8.1.3.2.2 virtual void telux::tel::IImsSettingsListener::onServiceStatusChange (telux::common::↩ ServiceStatus *status*) [virtual]

This function is called when IImsSettingsManager service status changes.

Parameters

in	*status*	- telux::common::ServiceStatus

4.8.2 Enumeration Type Documentation

4.8.2.1 enum telux::tel::ImsServiceConfigType

Defines the IMS service configuration parameters

Enumerator

IMSSETTINGS_VOIMS Voice calling support on LTE
IMSSETTINGS_IMS_SERVICE IMS Normal Registration configuration
IMSSETTINGS_SMS SMS support on IMS

4.9 Multi SIM

This section contains APIs related to Multi SIM DSDA Configuration.

4.9.1 Data Structure Documentation

4.9.1.1 struct telux::tel::SlotStatus

Represents status of a physical SIM slot

Data fields

Type	Field	Description
SlotState	slotState	State of the physical SIM slot
CardState	cardState	Status of the card in the physical slot
CardError	cardError	Indicates the reason for the card error, and is valid only when the card state is CARDSTATE_ERROR.

4.9.1.2 class telux::tel::IMultiSimManager

MultiSimManager allows to perform operation pertaining to devices which have more than one SIM/UICC card. Clients should check if the subsystem is ready before invoking any of the APIs as follows.

bool isReady = MultiSimManager->isSubsystemReady();

Public member functions

- virtual bool isSubsystemReady ()=0
- virtual std::future< bool > onSubsystemReady ()=0
- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual telux::common::Status getSlotCount (int &count)=0
- virtual telux::common::Status requestHighCapability (HighCapabilityCallback callback)=0
- virtual telux::common::Status setHighCapability (int slotId, common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status switchActiveSlot (SlotId slotId, common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestSlotStatus (SlotStatusCallback callback)=0
- virtual telux::common::Status registerListener (std::weak_ptr< IMultiSimListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< IMultiSimListener > listener)=0
- virtual ~IMultiSimManager ()

4.9.1.2.1 Constructors and Destructors

4.9.1.2.1.1 virtual telux::tel::IMultiSimManager::~IMultiSimManager ( ) [virtual]

4.9.1.2.2 Member Function Documentation

4.9.1.2.2.1 virtual bool telux::tel::IMultiSimManager::isSubsystemReady ( ) [pure virtual]

Checks the status of Multi SIM subsystem and returns the result.

Returns

If true MultiSimManager is ready.

Deprecated

Use IMultiSimManager::getServiceStatus() API.

4.9.1.2.2.2 virtual std::future<bool> telux::tel::IMultiSimManager::onSubsystemReady ( ) [pure virtual]

Wait for Multi SIM subsystem to be ready.

Returns

A future that caller can wait on to be notified when Multi SIM subsystem is ready.

Deprecated

Use InitResponseCb in PhoneFactory::getMultiSimManager instead, to get notified about subsystem readiness.

4.9.1.2.2.3 virtual telux::common::ServiceStatus telux::tel::IMultiSimManager::getServiceStatus ( ) [pure virtual]

This status indicates whether the IMultiSimManager object is in a usable state.

Returns

SERVICE_AVAILABLE - If MultiSim manager is ready for service. SERVICE_UNAVAILABLE - If MultiSim manager is temporarily unavailable. SERVICE_FAILED - If MultiSim manager encountered an irrecoverable failure.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.9.1.2.2.4 virtual telux::common::Status telux::tel::IMultiSimManager::getSlotCount (int & *count*) [pure virtual]

Get SIM slot count. The count can be used to determine whether the device supports multi SIM.

Parameters

out	*count*	Slot count.

Returns

Status of getSlotCount i.e. success or suitable error code.

4.9.1.2.2.5 virtual telux::common::Status telux::tel::IMultiSimManager::requestHighCapability (HighCapabilityCallback *callback*) [pure virtual]

Request to find out which SIM/slot is allowed to use advance Radio Technology like 5G at a time. For example SIM/slot with high capability may allowed to use RAT capabilities like 5G/4G/3G/2G while the SIM/slot with low capability may be allowed to use RAT capabilities like 4G/2G.

Parameters

in	*callback*	Callback function to get the response of request high capability.

Returns

Status of requestHighCapability i.e. success or suitable error code.

4.9.1.2.2.6 virtual telux::common::Status telux::tel::IMultiSimManager::setHighCapability (int *slotId,* common::ResponseCallback *callback = nullptr*) [pure virtual]

Set SIM/slot with high capability asynchronously. On dual SIM devices, only one SIM may be allowed to use advanced Radio technology like 5G at a time. This API sets the SIM/slot that should be allowed the highest RAT capability. The other SIM/slot will be given lower RAT capabilities. For example, SIM in slot1 will be allowed 2G/3G/4G/5G and the SIM in slot2 will be allowed only 2G/4G.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_MULTISIM_MGMT permission to invoke this API successfully.

Parameters

in	*slotId*	Slot set with high capablity.
in	*callback*	Callback function to get the response of set high capability request.

Returns

Status of setHighCapability i.e. success or suitable error code.

4.9.1.2.2.7 virtual telux::common::Status telux::tel::IMultiSimManager::switchActiveSlot (SlotId *slotId,* common::ResponseCallback *callback = nullptr*) [pure virtual]

Choose the physical SIM slot to be used by modem on Single-SIM TCU platforms. After switching the slot, only the SIM on chosen physical slot can be used for WWAN functionality.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_MULTISIM_MGMT permission to invoke this API successfully.

Parameters

in	*slotId*	physical slot to be made active
in	*callback*	Callback function to get the response of slot switch request

Returns

Status of switchActiveSlot i.e. success or suitable error code.

4.9.1.2.2.8 virtual telux::common::Status telux::tel::IMultiSimManager::requestSlotStatus (SlotStatusCallback *callback*) [pure virtual]

Request the status of physical slots.

Parameters

in	*callback*	Callback function to get the response of slot status request

Returns

Status of requestSlotStatus i.e. success or suitable error code.

4.9.1.2.2.9 virtual telux::common::Status telux::tel::IMultiSimManager::registerListener (std::weak_ptr< IMultiSimListener > *listener*) [pure virtual]

Register a listener for specific events in the Multi SIM subsystem.

Parameters

in	*listener*	Pointer to IMultiSimListener object that processes the notification

Returns

Status of registerListener i.e. success or suitable error code.

4.9.1.2.2.10 virtual telux::common::Status telux::tel::IMultiSimManager::deregisterListener (std::weak_ptr< IMultiSimListener > *listener*) [pure virtual]

Deregister the previously added listener.

Parameters

in	*listener*	Pointer to IMultiSimListener object that needs to be deregistered.

Returns

Status of deregisterListener i.e. success or suitable error code.

4.9.1.3 class telux::tel::IMultiSimListener

Listener class for getting high capability change notification. The listener method can be invoked from multiple different threads. Client needs to make sure that implementation is thread-safe.

Public member functions

- virtual void onHighCapabilityChanged (int slotId)
- virtual void onSlotStatusChanged (std::map< SlotId, SlotStatus > slotStatus)
- virtual ~IMultiSimListener ()

4.9.1.3.1 Constructors and Destructors

4.9.1.3.1.1 virtual telux::tel::IMultiSimListener::~IMultiSimListener ( ) [virtual]

Destructor of IMultiSimListener

4.9.1.3.2 Member Function Documentation

4.9.1.3.2.1 virtual void telux::tel::IMultiSimListener::onHighCapabilityChanged (int *slotId*) [virtual]

This function is called whenever there is change in high capability for SIM/slot.

Parameters

in	*slotId*	SIM corresponding to slot identifier has high capability now.

4.9.1.3.2.2 virtual void telux::tel::IMultiSimListener::onSlotStatusChanged (std::map< SlotId, SlotStatus > *slotStatus*) [virtual]

This function is called whenever there is change in physical SIM slots status.

Parameters

in	*slotStatus*	list of slots status SlotStatus

4.9.2 Enumeration Type Documentation

4.9.2.1 enum telux::tel::SlotState [strong]

Represents state of the physical SIM slot

Enumerator

UNKNOWN
INACTIVE Slot is inactive
ACTIVE Slot is active

4.10 Subscription Management

This section contains APIs related to Subscription Management.

4.10.1 Data Structure Documentation

4.10.1.1 class telux::tel::ISubscription

Subscription returns information about network operator subscription details pertaining to a SIM card.

Public member functions

- virtual std::string getCarrierName ()=0
- virtual std::string getIccId ()=0
- virtual int getMcc ()=0
- virtual int getMnc ()=0
- virtual std::string getMobileCountryCode ()=0
- virtual std::string getMobileNetworkCode ()=0
- virtual std::string getPhoneNumber ()=0
- virtual int getSlotId ()=0
- virtual std::string getImsi ()=0
- virtual std::string getGID1 ()=0
- virtual std::string getGID2 ()=0
- virtual ~ISubscription ()

4.10.1.1.1 Constructors and Destructors

4.10.1.1.1.1 virtual telux::tel::ISubscription::~ISubscription ( ) [virtual]

4.10.1.1.2 Member Function Documentation

4.10.1.1.2.1 virtual std::string telux::tel::ISubscription::getCarrierName ( ) [pure virtual]

Retrieves the name of the carrier on which this subscription is made.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SUB_PRIVATE_INFO permission to invoke this API successfully.

Returns

Name of the carrier.

4.10.1.1.2.2 virtual std::string telux::tel::ISubscription::getIccId ( ) [pure virtual]

Retrieves the SIM's ICCID (Integrated Chip ID) - i.e SIM Serial Number.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_PRIVATE_INFO_READ permission to invoke this API successfully.

Returns

Integrated Chip Id.

4.10.1.1.2.3 virtual int telux::tel::ISubscription::getMcc ( ) [pure virtual]

Retrieves the mobile country code of the carrier to which the phone is connected.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SUBSCRIPTION_READ permission to invoke this API successfully.

Returns

Mobile Country Code.

Deprecated

Use telux::tel::ISubscription::getMobileCountryCode() API instead

4.10.1.1.2.4 virtual int telux::tel::ISubscription::getMnc ( ) [pure virtual]

Retrieves the mobile network code of the carrier to which phone is connected.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SUBSCRIPTION_READ permission to invoke this API successfully.

Returns

Mobile Network Code.

Deprecated

Use telux::tel::ISubscription::getMobileNetworkCode() API instead

4.10.1.1.2.5 virtual std::string telux::tel::ISubscription::getMobileCountryCode ( ) [pure virtual]

Retrieves the mobile country code(MCC) of the carrier to which the phone is connected.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SUBSCRIPTION_READ permission to invoke this API successfully.

Returns

mcc.

4.10.1.1.2.6 virtual std::string telux::tel::ISubscription::getMobileNetworkCode ( ) [pure virtual]

Retrieves the mobile network code(MNC) of the carrier to which the phone is connected.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SUBSCRIPTION_READ permission to invoke this API successfully.

Returns

mnc.

4.10.1.1.2.7 virtual std::string telux::tel::ISubscription::getPhoneNumber ( ) [pure virtual]

Retrieves the phone number for the SIM subscription.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SUB_PRIVATE_INFO permission to invoke this API successfully.

Returns

PhoneNumber.

4.10.1.1.2.8 virtual int telux::tel::ISubscription::getSlotId ( ) [pure virtual]

Retrieves SIM Slot index for the SIM pertaining to this subscription object.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SUBSCRIPTION_READ permission to invoke this API successfully.

Returns

SIM slotId.

4.10.1.1.2.9 virtual std::string telux::tel::ISubscription::getImsi ( ) [pure virtual]

Retrieves IMSI (International Mobile Subscriber Identity) for the SIM. This will have home network MCC and MNC values.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SUB_PRIVATE_INFO permission to invoke this API successfully.

Returns

imsi.

4.10.1.1.2.10 virtual std::string telux::tel::ISubscription::getGID1 ( ) [pure virtual]

Retrieves the GID1(group identifier level1) on the SIM. It represents identifier for particular SIM and ME associations. It can be used to identify a group of SIMs for a particular application. Defined in 3GPP Spec 131.102 section 4.2.10

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SUBSCRIPTION_READ permission to invoke this API successfully.

Returns

GID1 content in hex format.

4.10.1.1.2.11 virtual std::string telux::tel::ISubscription::getGID2 ( ) [pure virtual]

Retrieves the GID2(group identifier level2) content on the SIM. It represents identifier for particular SIM and ME associations. It can be used to identify a group of SIMs for a particular application. Defined in 3GPP Spec 131.102 section 4.2.11

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SUBSCRIPTION_READ permission to invoke this API successfully.

Returns

GID2 content in hex format.

4.10.1.2 class telux::tel::ISubscriptionListener

A listener class for receiving device subscription information. The methods in listener can be invoked from multiple different threads. The implementation should be thread safe.

Public member functions

- virtual void onSubscriptionInfoChanged (std::shared_ptr< ISubscription > subscription)
- virtual void onNumberOfSubscriptionsChanged (int count)
- virtual ~ISubscriptionListener ()

4.10.1.2.1 Constructors and Destructors

4.10.1.2.1.1 virtual telux::tel::ISubscriptionListener::~ISubscriptionListener ( ) [virtual]

4.10.1.2.2 Member Function Documentation

4.10.1.2.2.1 virtual void telux::tel::ISubscriptionListener::onSubscriptionInfoChanged (std::shared_ptr< ISubscription > *subscription*) [virtual]

This function is called whenever there is a change in Subscription details.

Parameters

in	*subscription*	Pointer to ISubscription Object.

4.10.1.2.2.2 virtual void telux::tel::ISubscriptionListener::onNumberOfSubscriptionsChanged (int *count*) [virtual]

This function called whenever there is a change in the subscription count. for example when a new subscription is discovered or an existing subscription goes away when SIM is inserted or removed respectively.

Parameters

in	*count*	count of subscription

4.10.1.3 class telux::tel::ISubscriptionManager

Public member functions

- virtual bool isSubsystemReady ()=0
- virtual std::future< bool > onSubsystemReady ()=0
- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual std::shared_ptr< ISubscription > getSubscription (int slotId=DEFAULT_SLOT_ID, telux::common::Status *status=nullptr)=0
- virtual std::vector< std::shared_ptr< ISubscription > > getAllSubscriptions (telux::common::Status *status=nullptr)=0
- virtual telux::common::Status registerListener (std::weak_ptr< ISubscriptionListener > listener)=0
- virtual telux::common::Status removeListener (std::weak_ptr< ISubscriptionListener > listener)=0
- virtual ~ISubscriptionManager ()

4.10.1.3.1 Constructors and Destructors

4.10.1.3.1.1 virtual telux::tel::ISubscriptionManager::~ISubscriptionManager ( ) [virtual]

4.10.1.3.2 Member Function Documentation

4.10.1.3.2.1 virtual bool telux::tel::ISubscriptionManager::isSubsystemReady ( ) [pure virtual]

Checks the status of SubscriptionManager and returns the result.

Returns

If true then SubscriptionManager is ready for service.

Deprecated

Use ISubscriptionManager::getServiceStatus() API.

4.10.1.3.2.2 virtual std::future<bool> telux::tel::ISubscriptionManager::onSubsystemReady ( ) [pure virtual]

Wait for Subscription subsystem to be ready.

Returns

A future that caller can wait on to be notified when SubscriptionManager is ready.

Deprecated

Use InitResponseCb in PhoneFactory::getSubscriptionManager instead, to get notified about subsystem readiness.

4.10.1.3.2.3 virtual telux::common::ServiceStatus telux::tel::ISubscriptionManager::getServiceStatus ( ) [pure virtual]

This status indicates whether the ISubscriptionManager object is in a usable state.

Returns

SERVICE_AVAILABLE - If Subscription manager is ready for service. SERVICE_UNAVAILABLE - If Subscription manager is temporarily unavailable. SERVICE_FAILED - If Subscription manager encountered an irrecoverable failure.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.10.1.3.2.4 virtual std::shared_ptr<ISubscription> telux::tel::ISubscriptionManager::getSubscription (int *slotId = DEFAULT_SLOT_ID,* telux::common::Status * *status = nullptr*) [pure virtual]

Get Subscription details of the SIM in the given SIM slot.

Parameters

in	*slotId*	Slot id corresponding to the subscription.
out	*status*	Status of getSubscription i.e. success or suitable status code.

Returns

Pointer to ISubscription object.

4.10.1.3.2.5 virtual std::vector<std::shared_ptr<ISubscription> > telux::tel::ISubscriptionManager::getAllSubscriptions (telux::common::Status * *status = nullptr*) [pure virtual]

Get all the subscription details of the device.

Parameters

out	*status*	Status of getAllSubscriptions i.e. success or suitable status code.

Returns

list of ISubscription objects.

4.10.1.3.2.6 virtual telux::common::Status telux::tel::ISubscriptionManager::registerListener (std::weak_ptr< ISubscriptionListener > *listener*) [pure virtual]

Register a listener for Subscription events.

Parameters

in	*listener*	Pointer to ISubscriptionListener object that processes the notification.

Returns

Status of registerListener i.e. success or suitable status code.

4.10.1.3.2.7 virtual telux::common::Status telux::tel::ISubscriptionManager::removeListener (std::weak_ptr< ISubscriptionListener > *listener*) [pure virtual]

Remove a previously added listener.

Parameters

in	*listener*	Pointer to ISubscriptionListener object that needs to be removed.

Returns

Status of removeListener i.e. success or suitable status code.

4.11 Network Selection

Network Selection Manager provides the interface to get and set network selection mode (Manual or Automatic), scan available networks and set and get preferred networks list.

4.11.1 Data Structure Documentation

4.11.1.1 struct telux::tel::PreferredNetworkInfo

Defines the preferred network information

Data fields

Type	Field	Description
uint16_t	mcc	mobile country code
uint16_t	mnc	mobile network code
RatMask	ratMask	bit mask denotes which of the radio access technologies are set

4.11.1.2 struct telux::tel::OperatorStatus

Defines status of network operator

Data fields

Type	Field	Description
InUseStatus	inUse	In-use status of network operator
RoamingStatus	roaming	Roaming status of network operator
Forbidden↩ Status	forbidden	Forbidden status of network operator
PreferredStatus	preferred	Preferred status of network operator

4.11.1.3 struct telux::tel::NetworkScanInfo

Defines Network scan information

Data fields

Type	Field	Description
NetworkScan↩ Type	scanType	Network scan type
RatMask	ratMask	Bit mask denotes which of the radio access technologies are set. ratMask is valid/set only when scanType is provided as NetworkScanType::USER_SPECIFIED_RAT

4.11.1.4 class telux::tel::INetworkSelectionManager

Network Selection Manager class provides the interface to get and set network selection mode, preferred network list and scan available networks.

Public member functions

- virtual bool isSubsystemReady ()=0
- virtual std::future< bool > onSubsystemReady ()=0
- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual telux::common::Status requestNetworkSelectionMode (SelectionModeResponseCallback callback)=0
- virtual telux::common::Status setNetworkSelectionMode (NetworkSelectionMode selectMode, std::string mcc, std::string mnc, common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestPreferredNetworks (PreferredNetworksCallback callback)=0
- virtual telux::common::Status setPreferredNetworks (std::vector< PreferredNetworkInfo > preferredNetworksInfo, bool clearPrevious, common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status performNetworkScan (NetworkScanCallback callback)=0
- virtual telux::common::Status performNetworkScan (NetworkScanInfo info, common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status registerListener (std::weak_ptr< INetworkSelectionListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< INetworkSelectionListener > listener)=0
- virtual ~INetworkSelectionManager ()

4.11.1.4.1 Constructors and Destructors

4.11.1.4.1.1 virtual telux::tel::INetworkSelectionManager::~INetworkSelectionManager ( ) [virtual]

4.11.1.4.2 Member Function Documentation

4.11.1.4.2.1 virtual bool telux::tel::INetworkSelectionManager::isSubsystemReady ( ) [pure virtual]

Checks the status of network subsystem and returns the result.

Returns

True if network subsystem is ready for service otherwise false.

Deprecated

Use INetworkSelectionManager::getServiceStatus() API.

4.11.1.4.2.2 virtual std::future<bool> telux::tel::INetworkSelectionManager::onSubsystemReady ( ) [pure virtual]

Wait for network subsystem to be ready.

Returns

A future that caller can wait on to be notified when network subsystem is ready.

Deprecated

Use InitResponseCb in PhoneFactory::getNetworkSelectionManager instead, to get notified about subsystem readiness.

4.11.1.4.2.3 virtual telux::common::ServiceStatus telux::tel::INetworkSelectionManager::getService↩ Status ( ) [pure virtual]

This status indicates whether the INetworkSelectionManager object is in a usable state.

Returns

SERVICE_AVAILABLE - If Serving System manager is ready for service.
SERVICE_UNAVAILABLE - If Serving System manager is temporarily unavailable.
SERVICE_FAILED - If Serving System manager encountered an irrecoverable failure.

4.11.1.4.2.4 virtual telux::common::Status telux::tel::INetworkSelectionManager::request↩ NetworkSelectionMode (SelectionModeResponseCallback *callback*) [pure virtual]

Get current network selection mode (i.e Manual or Automatic) asynchronously.

On platforms with Access control enabled, Caller needs to have
TELUX_TEL_NETWORK_SELECTION_READ permission to invoke this API successfully.

Parameters

in	*callback*	Callback function to get the response of get network selection mode request.

Returns

Status of requestNetworkSelectionMode i.e. success or suitable error code.

4.11.1.4.2.5 virtual telux::common::Status telux::tel::INetworkSelectionManager::setNetwork↩ SelectionMode (NetworkSelectionMode *selectMode,* std::string *mcc,* std::string *mnc,* common::ResponseCallback *callback = nullptr*) [pure virtual]

Set current network selection mode and receive the response asynchronously.

On platforms with Access control enabled, Caller needs to have

TELUX_TEL_NETWORK_SELECTION_OPS permission to invoke this API successfully.

Parameters

in	*selectMode*	Selection mode for a network i.e. automatic or manual. If selection mode is automatic then MCC and MNC are ignored. If it is manual, client has to explicitly pass MCC and MNC as arguments.
in	*callback*	Optional callback function to get the response of set network selection mode request.
in	*mcc*	Mobile Country Code (Applicable only for MANUAL selection mode).
in	*mnc*	Mobile Network Code (Applicable only for MANUAL selection mode).

Returns

Status of setNetworkSelectionMode i.e. success or suitable error code.

4.11.1.4.2.6 virtual telux::common::Status telux::tel::INetworkSelectionManager::requestPreferredNetworks (PreferredNetworksCallback *callback*) [pure virtual]

Get 3GPP preferred network list and static 3GPP preferred network list asynchronously. Higher priority networks appear first in the list. The networks that appear in the 3GPP Preferred Networks list get higher priority than the networks in the static 3GPP preferred networks list.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_NETWORK_SELECTION_READ permission to invoke this API successfully.

Parameters

in	*callback*	Callback function to get the response of get preferred networks request.

Returns

Status of requestPreferredNetworks i.e. success or suitable error code.

4.11.1.4.2.7 virtual telux::common::Status telux::tel::INetworkSelectionManager::setPreferredNetworks (std::vector< PreferredNetworkInfo > *preferredNetworksInfo,* bool *clearPrevious,* common::ResponseCallback *callback = nullptr*) [pure virtual]

Set 3GPP preferred network list and receive the response asynchronously. It overrides the existing preferred network list. The preferred network list affects network selection selection when automatic registration is performed by the device. Higher priority networks should appear first in the list.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_NETWORK_SELECTION_OPS permission to invoke this API successfully.

Parameters

in	*preferredNetworks↩ Info*	List of 3GPP preferred networks.
in	*clearPrevious*	If flag is false then new 3GPP preferred network list is appended to existing preferred network list. If flag is true then old list is flushed and new 3GPP preferred network list is added.
in	*callback*	Callback function to get the response of set preferred network list request.

Returns

Status of setPreferredNetworks i.e. success or suitable error code.

4.11.1.4.2.8 virtual telux::common::Status telux::tel::INetworkSelectionManager::performNetworkScan (NetworkScanCallback *callback*) [pure virtual]

Perform the network scan and returns a list of available networks.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_NETWORK_SELECTION_OPS permission to invoke this API successfully.

Parameters

in	*callback*	Callback function to get the response of perform network scan request

Returns

Status of performNetworkScan i.e. success or suitable error code.

Deprecated

Use INetworkSelectionManager::performNetworkScan(common::ResponseCallback callback) API instead

4.11.1.4.2.9 virtual telux::common::Status telux::tel::INetworkSelectionManager::performNetworkScan (NetworkScanInfo *info,* common::ResponseCallback *callback = nullptr*) [pure virtual]

Perform the network scan. The available networks list is returned incrementally as they become available, without waiting for the entire scan to complete through the indication API (INetworkSelectionListener::onNetworkScanResults). The scan status in indication will indicate if its a partial result or complete result.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_NETWORK_SELECTION_OPS permission to invoke this API successfully.

Parameters

in	*info*	Provides network scan type and if the network scan type is user prefered RAT, includes RAT(s) information. NetworkScanInfo
in	*callback*	Callback function to get the response of network scan request

Returns

Status of performNetworkScan i.e. success or suitable error code.

4.11.1.4.2.10 virtual telux::common::Status telux::tel::INetworkSelectionManager::registerListener (std::weak_ptr< INetworkSelectionListener > *listener*) [pure virtual]

Register a listener for specific updates from network access service.

Parameters

in	*listener*	Pointer of INetworkSelectionListener object that processes the notification

Returns

Status of registerListener i.e success or suitable status code.

4.11.1.4.2.11 virtual telux::common::Status telux::tel::INetworkSelectionManager::deregisterListener (std::weak_ptr< INetworkSelectionListener > *listener*) [pure virtual]

Deregister the previously added listener.

Parameters

in	*listener*	Previously registered INetworkSelectionListener that needs to be removed

Returns

Status of removeListener success or suitable status code

4.11.1.5 class telux::tel::OperatorInfo

Operator Info class provides operator name, MCC, MNC and network status.

Public member functions

- OperatorInfo (std::string networkName, std::string mcc, std::string mnc, OperatorStatus operatorStatus)
- OperatorInfo (std::string networkName, std::string mcc, std::string mnc, RadioTechnology rat, OperatorStatus operatorStatus)

- std::string getName ()
- std::string getMcc ()
- std::string getMnc ()
- RadioTechnology getRat ()
- OperatorStatus getStatus ()

4.11.1.5.1 Constructors and Destructors

4.11.1.5.1.1 telux::tel::OperatorInfo::OperatorInfo (std::string *networkName,* std::string *mcc,* std::string *mnc,* OperatorStatus *operatorStatus*)

4.11.1.5.1.2 telux::tel::OperatorInfo::OperatorInfo (std::string *networkName,* std::string *mcc,* std::string *mnc,* RadioTechnology *rat,* OperatorStatus *operatorStatus*)

4.11.1.5.2 Member Function Documentation

4.11.1.5.2.1 std::string telux::tel::OperatorInfo::getName ( )

Get Operator name or description

Returns

Operator name.

4.11.1.5.2.2 std::string telux::tel::OperatorInfo::getMcc ( )

Get mcc from the operator numeric.

Returns

MCC.

4.11.1.5.2.3 std::string telux::tel::OperatorInfo::getMnc ( )

Get mnc from operator numeric.

Returns

MNC.

4.11.1.5.2.4 RadioTechnology telux::tel::OperatorInfo::getRat ( )

Get radio access technology.

Returns

Radio access technology(RAT) RadioTechnology.

4.11.1.5.2.5 OperatorStatus telux::tel::OperatorInfo::getStatus ( )

Get status of operator.

Returns

status of the operator OperatorStatus.

4.11.1.6 class telux::tel::INetworkSelectionListener

Listener class for getting network selection mode change notification.

The methods in listener can be invoked from multiple different threads. Client needs to make sure that implementation is thread-safe.

Public member functions

- virtual void onSelectionModeChanged (NetworkSelectionMode mode)
- virtual void onNetworkScanResults (NetworkScanStatus scanStatus, std::vector< telux::tel::OperatorInfo > operatorInfos)
- virtual ~INetworkSelectionListener ()

4.11.1.6.1 Constructors and Destructors

4.11.1.6.1.1 virtual telux::tel::INetworkSelectionListener::~INetworkSelectionListener ( ) [virtual]

Destructor of INetworkSelectionListener

4.11.1.6.2 Member Function Documentation

4.11.1.6.2.1 virtual void telux::tel::INetworkSelectionListener::onSelectionModeChanged (NetworkSelectionMode *mode*) [virtual]

This function is called whenever network selection mode is changed.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_NETWORK_SELECTION_READ permission to receive this notification.

Parameters

in	*mode*	Network selection mode NetworkSelectionMode

4.11.1.6.2.2 virtual void telux::tel::INetworkSelectionListener::onNetworkScanResults (NetworkScanStatus *scanStatus*, std::vector< telux::tel::OperatorInfo > *operatorInfos*) [virtual]

This function is called in response to performNetworkScan API. This API will be invoked multiple times in case of partial network scan results. In case of network scan failure and network scan completed this API

will not be invoked further.

Parameters

in	*scanStatus*	Status of the network scan results NetworkScanStatus
in	*operatorInfos*	Operators info with details of network operator name, MCC, MNC, etc. In case of partial network scan results, the operator info will have the information of the new set of operator info along with previous partial network scan results.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibilty.

4.11.2 Enumeration Type Documentation

4.11.2.1 enum telux::tel::RatType

Defines network RAT type for preferred networks. Each value represents corresponding bit for RatMask bitset.

Enumerator

UMTS UMTS
LTE LTE
LTE
GSM GSM
GSM
NR5G NR5G
NR5G

4.11.2.2 enum telux::tel::NetworkScanStatus [strong]

Defines the status of the network scan results

Enumerator

COMPLETE Network scan is successful and completed. No more indications are expected for the scan request
PARTIAL Network scan results are partial, further results are expected in subsequent indication
FAILED Network scan failed either due to radio link failure or it is aborted or due to problem in performing incremental search.

4.11.2.3 enum telux::tel::NetworkSelectionMode [strong]

Defines network selection mode

Enumerator

UNKNOWN Unknown
AUTOMATIC Device registers according to provisioned mcc and mnc

MANUAL Device registers to specified network as per provided mcc and mnc

4.11.2.4 enum telux::tel::InUseStatus [strong]

Defines in-use status of network operator

Enumerator

UNKNOWN Unknown
CURRENT_SERVING Current serving
AVAILABLE Available

4.11.2.5 enum telux::tel::RoamingStatus [strong]

Defines roaming status of network operator

Enumerator

UNKNOWN Unknown
HOME Home
ROAM Roaming

4.11.2.6 enum telux::tel::ForbiddenStatus [strong]

Defines forbidden status of network operator

Enumerator

UNKNOWN Unknown
FORBIDDEN Forbidden
NOT_FORBIDDEN Not forbidden

4.11.2.7 enum telux::tel::PreferredStatus [strong]

Defines preferred status of network operator

Enumerator

UNKNOWN Unknown
PREFERRED Preferred
NOT_PREFERRED Not preferred

4.11.2.8 enum telux::tel::NetworkScanType [strong]

Defines Network scan type

Enumerator

CURRENT_RAT_PREFERENCE Network scan based on current RAT preference
USER_SPECIFIED_RAT Network scan based on user specified RAT(s)
ALL_RATS Network scan on GSM/WCDMA/LTE/NR5G

4.12 Serving System

Serving System Manager class provides the interface to request and set service domain preference and radio access technology mode preference for searching and registering (CS/PS domain, RAT and operation mode)

4.12.1 Data Structure Documentation

4.12.1.1 struct telux::tel::ServingSystemInfo

Defines current serving system information

Data fields

Type	Field	Description
RadioTechnology	rat	Current serving RAT
ServiceDomain	domain	Current service domain registered on system for the serving RAT

4.12.1.2 struct telux::tel::RFBandInfo

Defines information of RF bands.

Data fields

Type	Field	Description
RFBand	band	Currently active band
uint32_t	channel	Currently active channel
RFBandWidth	bandWidth	Bandwidth information

4.12.1.3 struct telux::tel::DcStatus

Defines Dual Connectivity status

Data fields

Type	Field	Description
EndcAvailability	endcAvailability	ENDC availability
DcnrRestriction	dcnrRestriction	DCNR restriction

4.12.1.4 struct telux::tel::NetworkTimeInfo

Defines Network time information

Data fields

Type	Field	Description
uint16_t	year	Year.
uint8_t	month	Month. 1 is January and 12 is December.

Type	Field	Description
uint8_t	day	Day. Range: 1 to 31.
uint8_t	hour	Hour. Range: 0 to 23.
uint8_t	minute	Minute. Range: 0 to 59.
uint8_t	second	Second. Range: 0 to 59.
uint8_t	dayOfWeek	Day of the week. 0 is Monday and 6 is Sunday.
int8_t	timeZone	Offset between UTC and local time in units of 15 minutes (signed value). Actual value = field value ∗ 15 minutes.
uint8_t	dstAdj	Daylight saving adjustment in hours to obtain local time. Possible values: 0, 1, and 2.
string	nitzTime	Network Identity and Time Zone(NITZ) information in the form "yyyy/mm/dd,hh:mm:ss(+/-)tzh:tzm,dt

4.12.1.5 class telux::tel::IServingSystemManager

Public member functions

- virtual bool isSubsystemReady ()=0
- virtual std::future< bool > onSubsystemReady ()=0
- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual telux::common::Status setRatPreference (RatPreference ratPref, common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestRatPreference (RatPreferenceCallback callback)=0
- virtual telux::common::Status setServiceDomainPreference (ServiceDomainPreference serviceDomain, common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestServiceDomainPreference (ServiceDomainPreferenceCallback callback)=0
- virtual telux::common::Status getSystemInfo (ServingSystemInfo &sysInfo)=0
- virtual telux::tel::DcStatus getDcStatus ()=0
- virtual telux::common::Status requestNetworkTime (NetworkTimeResponseCallback callback)=0
- virtual telux::common::Status requestRFBandInfo (RFBandInfoCallback callback)=0
- virtual telux::common::Status registerListener (std::weak_ptr< IServingSystemListener > listener, ServingSystemNotificationMask mask=ALL_NOTIFICATIONS)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< IServingSystemListener > listener, ServingSystemNotificationMask mask=ALL_NOTIFICATIONS)=0
- virtual ∼IServingSystemManager ()

Static Public Attributes

- static const uint32_t ALL_NOTIFICATIONS = 0xFFFFFFFF

4.12.1.5.1 Constructors and Destructors

4.12.1.5.1.1 virtual telux::tel::IServingSystemManager::~IServingSystemManager ( ) [virtual]

Destructor of IServingSystemManager

4.12.1.5.2 Member Function Documentation

4.12.1.5.2.1 virtual bool telux::tel::IServingSystemManager::isSubsystemReady ( ) [pure virtual]

Checks the status of serving subsystem and returns the result.

Returns

True if serving subsystem is ready for service otherwise false.

Deprecated

Use IServingSystemManager::getServiceStatus() API.

4.12.1.5.2.2 virtual std::future<bool> telux::tel::IServingSystemManager::onSubsystemReady ( ) [pure virtual]

Wait for serving subsystem to be ready.

Returns

A future that caller can wait on to be notified when serving subsystem is ready.

Deprecated

Use InitResponseCb in PhoneFactory::getServingSystemManager instead, to get notified about subsystem readiness.

4.12.1.5.2.3 virtual telux::common::ServiceStatus telux::tel::IServingSystemManager::getServiceStatus ( ) [pure virtual]

This status indicates whether the IServingSystemManager object is in a usable state.

Returns

SERVICE_AVAILABLE - If Serving System manager is ready for service.
SERVICE_UNAVAILABLE - If Serving System manager is temporarily unavailable.
SERVICE_FAILED - If Serving System manager encountered an irrecoverable failure.

4.12.1.5.2.4 virtual telux::common::Status telux::tel::IServingSystemManager::setRatPreference (RatPreference *ratPref,* common::ResponseCallback *callback* = `nullptr`) [pure virtual]

Set the preferred radio access technology mode that the device should use to acquire service.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SRV_SYSTEM_CONFIG permission to invoke this API successfully.

Parameters

in	*ratPref*	Radio access technology mode preference.
in	*callback*	Callback function to get the response of set RAT mode preference.

Returns

Status of setRatPreference i.e. success or suitable error code.

4.12.1.5.2.5 virtual telux::common::Status telux::tel::IServingSystemManager::requestRatPreference (RatPreferenceCallback *callback*) [pure virtual]

Request for preferred radio access technology mode.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SRV_SYSTEM_READ permission to invoke this API successfully.

Parameters

in	*callback*	Callback function to get the response of request preferred RAT mode.

Returns

Status of requestRatPreference i.e. success or suitable error code.

4.12.1.5.2.6 virtual telux::common::Status telux::tel::IServingSystemManager::setServiceDomain↩ Preference (ServiceDomainPreference *serviceDomain,* common::ResponseCallback *callback* = `nullptr`) [pure virtual]

Initiate service domain preference like CS, PS or CS_PS and receive the response asynchronously.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SRV_SYSTEM_CONFIG permission to invoke this API successfully.

Parameters

in	*serviceDomain*	ServiceDomainPreference.
in	*callback*	Callback function to get the response of set service domain preference request.

Returns

Status of setServiceDomainPreference i.e. success or suitable error code.

4.12.1.5.2.7 virtual telux::common::Status telux::tel::IServingSystemManager::requestServiceDomain↩ Preference (ServiceDomainPreferenceCallback *callback*) [pure virtual]

Request for Service Domain Preference asynchronously.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SRV_SYSTEM_READ permission to invoke this API successfully.

Parameters

in	*callback*	Callback function to get the response of request service domain preference.

Returns

Status of requestServiceDomainPreference i.e. success or suitable error code.

4.12.1.5.2.8 virtual telux::common::Status telux::tel::IServingSystemManager::getSystemInfo (ServingSystemInfo & *sysInfo*) [pure virtual]

Get the Serving system information. Supports only 3GPP RATs.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SRV_SYSTEM_READ permission to invoke this API successfully.

Parameters

out	*sysInfo*	Serving system information ServingSystemInfo

Returns

Status of getServingSystemInfo i.e. success or suitable error code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.12.1.5.2.9 virtual telux::tel::DcStatus telux::tel::IServingSystemManager::getDcStatus ( ) [pure virtual]

Request for Dual Connectivity status on 5G NR.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SRV_SYSTEM_READ permission to invoke this API successfully.

Returns

DcStatus

4.12.1.5.2.10 virtual telux::common::Status telux::tel::IServingSystemManager::requestNetworkTime (NetworkTimeResponseCallback *callback*) [pure virtual]

Get network time information asynchronously.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SRV_SYSTEM_READ permission to invoke this API successfully.

Parameters

in	*callback*	Callback function to get the response of get network time information request.

Returns

Status of requestNetworkTime i.e. success or suitable error code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.12.1.5.2.11 virtual telux::common::Status telux::tel::IServingSystemManager::requestRFBandInfo (RFBandInfoCallback *callback*) [pure virtual]

Get the information about the band that the device is currently using.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SRV_SYSTEM_READ permission to invoke this API successfully.

Parameters

in	*callback*	Callback function to get the response of get RF band information request.

Returns

Status of requestRFBandInfo i.e. success or suitable error code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.12.1.5.2.12 virtual telux::common::Status telux::tel::IServingSystemManager::registerListener (std::weak_ptr< IServingSystemListener > *listener,* ServingSystemNotificationMask *mask* = ALL_NOTIFICATIONS) [pure virtual]

Register a listener for specific updates from serving system.

Parameters

in	*listener*	Pointer of IServingSystemListener object that processes the notification
in	*mask*	Bit mask representing a set of notifications that needs to be registered - ServingSystemNotificationMask Notifications under IServingSystemListener that are not listed in in ServingSystemNotificationType would always be registered by default. All the notifications will be registered when the client provides ALL_NOTIFICATIONS as input. The bits that are not set in the mask are ignored and do not have any effect on registration. To deregister, the API deregisterListener should be used.

Returns

Status of registerListener i.e success or suitable status code.

4.12.1.5.2.13 virtual telux::common::Status telux::tel::IServingSystemManager::deregisterListener (std::weak_ptr< IServingSystemListener > *listener,* ServingSystemNotificationMask *mask* = ALL_NOTIFICATIONS) [pure virtual]

Deregister the previously added listener.

Parameters

in	*listener*	Previously registered IServingSystemListener that needs to be removed
in	*mask*	Bit mask that denotes a set of notifications that needs to be de-registered - ServingSystemNotificationMask Notifications under IServingSystemListener that are not listed in ServingSystemNotificationType will be de-registered only when ALL_NOTIFICATIONS is provided as input. The bits that are not set in the mask are ignored and does not have any effect on de-registration. However, providing an empty mask is an invalid operation. To register again, the API deregisterListener should be used.

Returns

Status of removeListener i.e. success or suitable status code

4.12.1.5.3 Field Documentation

4.12.1.5.3.1 const uint32_t telux::tel::IServingSystemManager::ALL_NOTIFICATIONS = 0xFFFFFFFF [static]

Represents the set of all notifications defined in ServingSystemNotificationType. When this constant value is provided for registration or deregistration, all notifications will be registered or deregistered.

4.12.1.6 class telux::tel::IServingSystemListener

Listener class for getting notifications related to updates in radio access technology mode preference, service domain preference, serving system information, etc. Some notifications in this listener could be frequent in nature. When the system is in a suspended/low power state, those indications will wake the system up. This could result in increased power consumption by the system. If those notifications are not required in the suspended/low power state, it is recommended for the client to de-register specific notifications using the deregisterListener API.

The listener method can be invoked from multiple different threads. Client needs to make sure that implementation is thread-safe.

Public member functions

- virtual void onRatPreferenceChanged (RatPreference preference)
- virtual void onServiceDomainPreferenceChanged (ServiceDomainPreference preference)
- virtual void onSystemInfoChanged (ServingSystemInfo sysInfo)
- virtual void onDcStatusChanged (DcStatus dcStatus)
- virtual void onNetworkTimeChanged (NetworkTimeInfo info)
- virtual void onRFBandInfoChanged (RFBandInfo bandInfo)
- virtual ~IServingSystemListener ()

4.12.1.6.1 Constructors and Destructors

4.12.1.6.1.1 virtual telux::tel::IServingSystemListener::~IServingSystemListener ( ) [virtual]

Destructor of IServingSystemListener

4.12.1.6.2 Member Function Documentation

4.12.1.6.2.1 virtual void telux::tel::IServingSystemListener::onRatPreferenceChanged (RatPreference *preference*) [virtual]

This function is called whenever RAT mode preference is changed.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SRV_SYSTEM_READ permission to receive this notification.

Parameters

in	*preference*	RatPreference

4.12.1.6.2.2 virtual void telux::tel::IServingSystemListener::onServiceDomainPreferenceChanged (ServiceDomainPreference *preference*) [virtual]

This function is called whenever service domain preference is changed.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SRV_SYSTEM_READ permission to receive this notification.

Parameters

in	*preference*	ServiceDomainPreference

4.12.1.6.2.3 virtual void telux::tel::IServingSystemListener::onSystemInfoChanged (ServingSystem↩ Info *sysInfo*) [virtual]

This function is called whenever the Serving System information is changed. Supports only 3GPP RATs.

To receive this notification, client needs to register a listener using registerListener API by setting the ServingSystemNotificationType::SYSTEM_INFO bit in the bitmask.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SRV_SYSTEM_READ permission to receive this notification.

Parameters

in	*sysInfo*	ServingSystemInfo

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.12.1.6.2.4 virtual void telux::tel::IServingSystemListener::onDcStatusChanged (DcStatus *dcStatus*) [virtual]

This function is called whenever the Dual Connnectivity status is changed on 5G NR.

To receive this notification, client needs to register a listener using registerListener API by setting the ServingSystemNotificationType::SYSTEM_INFO bit in the bitmask.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SRV_SYSTEM_READ permission to receive this notification.

Parameters

in	*dcStatus*	DcStatus

4.12.1.6.2.5 virtual void telux::tel::IServingSystemListener::onNetworkTimeChanged (NetworkTimeInfo *info*) [virtual]

This function is called whenever network time information is changed.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SRV_SYSTEM_READ permission to receive this notification.

Parameters

in	*info*	Network time information NetworkTimeInfo

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.12.1.6.2.6 virtual void telux::tel::IServingSystemListener::onRFBandInfoChanged (RFBandInfo *bandInfo*) [virtual]

This function is called whenever the RF band information changes.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SRV_SYSTEM_READ permission to receive this notification.

Parameters

in	*bandInfo*	RFBandInfo

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.12.2 Enumeration Type Documentation

4.12.2.1 enum telux::tel::ServiceDomainPreference [strong]

Defines service domain preference

Enumerator

UNKNOWN
CS_ONLY Circuit-switched only

PS_ONLY Packet-switched only
CS_PS Circuit-switched and packet-switched

4.12.2.2 enum telux::tel::ServiceDomain `[strong]`

Defines service domain

Enumerator

UNKNOWN Unknown, when the information is not available
NO_SRV No Service
CS_ONLY Circuit-switched only
PS_ONLY Packet-switched only
CS_PS Circuit-switched and packet-switched
CAMPED Device camped on the network according to its provisioning, but not registered

4.12.2.3 enum telux::tel::RFBand `[strong]`

Defines RF Bands.

Enumerator

INVALID
BC_0
BC_1
BC_3
BC_4
BC_5
BC_6
BC_7
BC_8
BC_9
BC_10
BC_11
BC_12
BC_13
BC_14
BC_15
BC_16
BC_17
BC_18
BC_19
GSM_450
GSM_480
GSM_750
GSM_850
GSM_900_EXTENDED
GSM_900_PRIMARY
GSM_900_RAILWAYS
GSM_1800
GSM_1900

WCDMA_2100
WCDMA_PCS_1900
WCDMA_DCS_1800
WCDMA_1700_US
WCDMA_850
WCDMA_800
WCDMA_2600
WCDMA_900
WCDMA_1700_JAPAN
WCDMA_1500_JAPAN
WCDMA_850_JAPAN
E_UTRA_OPERATING_BAND_1
E_UTRA_OPERATING_BAND_2
E_UTRA_OPERATING_BAND_3
E_UTRA_OPERATING_BAND_4
E_UTRA_OPERATING_BAND_5
E_UTRA_OPERATING_BAND_6
E_UTRA_OPERATING_BAND_7
E_UTRA_OPERATING_BAND_8
E_UTRA_OPERATING_BAND_9
E_UTRA_OPERATING_BAND_10
E_UTRA_OPERATING_BAND_11
E_UTRA_OPERATING_BAND_12
E_UTRA_OPERATING_BAND_13
E_UTRA_OPERATING_BAND_14
E_UTRA_OPERATING_BAND_17
E_UTRA_OPERATING_BAND_33
E_UTRA_OPERATING_BAND_34
E_UTRA_OPERATING_BAND_35
E_UTRA_OPERATING_BAND_36
E_UTRA_OPERATING_BAND_37
E_UTRA_OPERATING_BAND_38
E_UTRA_OPERATING_BAND_39
E_UTRA_OPERATING_BAND_40
E_UTRA_OPERATING_BAND_18
E_UTRA_OPERATING_BAND_19
E_UTRA_OPERATING_BAND_20
E_UTRA_OPERATING_BAND_21
E_UTRA_OPERATING_BAND_24
E_UTRA_OPERATING_BAND_25
E_UTRA_OPERATING_BAND_41
E_UTRA_OPERATING_BAND_42
E_UTRA_OPERATING_BAND_43
E_UTRA_OPERATING_BAND_23
E_UTRA_OPERATING_BAND_26
E_UTRA_OPERATING_BAND_32
E_UTRA_OPERATING_BAND_125
E_UTRA_OPERATING_BAND_126
E_UTRA_OPERATING_BAND_127
E_UTRA_OPERATING_BAND_28

E_UTRA_OPERATING_BAND_29
E_UTRA_OPERATING_BAND_30
E_UTRA_OPERATING_BAND_66
E_UTRA_OPERATING_BAND_250
E_UTRA_OPERATING_BAND_46
E_UTRA_OPERATING_BAND_27
E_UTRA_OPERATING_BAND_31
E_UTRA_OPERATING_BAND_71
E_UTRA_OPERATING_BAND_47
E_UTRA_OPERATING_BAND_48
E_UTRA_OPERATING_BAND_67
E_UTRA_OPERATING_BAND_68
E_UTRA_OPERATING_BAND_49
E_UTRA_OPERATING_BAND_85
E_UTRA_OPERATING_BAND_72
E_UTRA_OPERATING_BAND_73
E_UTRA_OPERATING_BAND_86
E_UTRA_OPERATING_BAND_53
E_UTRA_OPERATING_BAND_87
E_UTRA_OPERATING_BAND_88
E_UTRA_OPERATING_BAND_70
TDSCDMA_BAND_A
TDSCDMA_BAND_B
TDSCDMA_BAND_C
TDSCDMA_BAND_D
TDSCDMA_BAND_E
TDSCDMA_BAND_F
NR5G_BAND_1
NR5G_BAND_2
NR5G_BAND_3
NR5G_BAND_5
NR5G_BAND_7
NR5G_BAND_8
NR5G_BAND_20
NR5G_BAND_28
NR5G_BAND_38
NR5G_BAND_41
NR5G_BAND_50
NR5G_BAND_51
NR5G_BAND_66
NR5G_BAND_70
NR5G_BAND_71
NR5G_BAND_74
NR5G_BAND_75
NR5G_BAND_76
NR5G_BAND_77
NR5G_BAND_78
NR5G_BAND_79
NR5G_BAND_80
NR5G_BAND_81

NR5G_BAND_82
NR5G_BAND_83
NR5G_BAND_84
NR5G_BAND_85
NR5G_BAND_257
NR5G_BAND_258
NR5G_BAND_259
NR5G_BAND_260
NR5G_BAND_261
NR5G_BAND_12
NR5G_BAND_25
NR5G_BAND_34
NR5G_BAND_39
NR5G_BAND_40
NR5G_BAND_65
NR5G_BAND_86
NR5G_BAND_48
NR5G_BAND_14
NR5G_BAND_13
NR5G_BAND_18
NR5G_BAND_26
NR5G_BAND_30
NR5G_BAND_29
NR5G_BAND_53
NR5G_BAND_46
NR5G_BAND_91
NR5G_BAND_92
NR5G_BAND_93
NR5G_BAND_94

4.12.2.4 enum telux::tel::RFBandWidth `[strong]`

Defines RF Bandwidth Information.

Enumerator

INVALID_BANDWIDTH Invalid Value
LTE_BW_NRB_6 LTE 1.4
LTE_BW_NRB_15 LTE 3
LTE_BW_NRB_25 LTE 5
LTE_BW_NRB_50 LTE 10
LTE_BW_NRB_75 LTE 15
LTE_BW_NRB_100 LTE 20
NR5G_BW_NRB_5 NR5G 5
NR5G_BW_NRB_10 NR5G 10
NR5G_BW_NRB_15 NR5G 15
NR5G_BW_NRB_20 NR5G 20
NR5G_BW_NRB_25 NR5G 25
NR5G_BW_NRB_30 NR5G 30
NR5G_BW_NRB_40 NR5G 40

NR5G_BW_NRB_50 NR5G 50
NR5G_BW_NRB_60 NR5G 60
NR5G_BW_NRB_80 NR5G 80
NR5G_BW_NRB_90 NR5G 90
NR5G_BW_NRB_100 NR5G 100
NR5G_BW_NRB_200 NR5G 200
NR5G_BW_NRB_400 NR5G 400
GSM_BW_NRB_2 GSM 0.2
TDSCDMA_BW_NRB_2 TDSCDMA 1.6
WCDMA_BW_NRB_5 WCDMA 5
WCDMA_BW_NRB_10 WCDMA 10
NR5G_BW_NRB_70 NR5G 70

4.12.2.5 enum telux::tel::RatPrefType

Defines the radio access technology mode preference.

Enumerator

PREF_CDMA_1X CDMA_1X
PREF_CDMA_EVDO CDMA_EVDO
PREF_GSM GSM
PREF_WCDMA WCDMA
PREF_LTE LTE
PREF_TDSCDMA TDSCDMA
PREF_NR5G NR5G

4.12.2.6 enum telux::tel::EndcAvailability `[strong]`

Defines ENDC(E-UTRAN New Radio-Dual Connectivity) Availability status on 5G NR

Enumerator

UNKNOWN Status unknown
AVAILABLE ENDC is Available
UNAVAILABLE ENDC is not Available

4.12.2.7 enum telux::tel::DcnrRestriction `[strong]`

Defines DCNR(Dual Connectivity with NR) Restriction status on 5G NR

Enumerator

UNKNOWN Status unknown
RESTRICTED DCNR is Rescticted
UNRESTRICTED DCNR is not Restricted

4.12.2.8 enum telux::tel::ServingSystemNotificationType

Defines some of the notifications supported by IServingSystemListener which can be dynamically disabled/enabled. Each entry represents one or more listener callbacks in IServingSystemListener

Enumerator

SYSTEM_INFO

4.13 Remote SIM Provisioning

This section contains APIs related to Remote SIM provisioning.

4.13.1 Data Structure Documentation

4.13.1.1 struct telux::tel::CustomHeader

Header information to be sent along with HTTP post request.

Data fields

Type	Field	Description
string	name	Header name
string	value	Header value

4.13.1.2 class telux::tel::IHttpTransactionListener

The interface listens for indication to perform HTTP request and send back the response for HTTP request to modem.

The methods in the listener can be invoked from multiple threads. It is client's responsibility to make sure the implementation is thread safe.

Public member functions

- virtual void onNewHttpRequest (const std::string &url, uint32_t tokenId, const std::vector< CustomHeader > &headers, const std::vector< uint8_t > &reqPayload)
- virtual ~IHttpTransactionListener ()

4.13.1.2.1 Constructors and Destructors

4.13.1.2.1.1 virtual telux::tel::IHttpTransactionListener::~IHttpTransactionListener ( ) [virtual]

Destructor of IHttpTransactionListener

4.13.1.2.2 Member Function Documentation

4.13.1.2.2.1 virtual void telux::tel::IHttpTransactionListener::onNewHttpRequest (const std::string & *url,* uint32_t *tokenId,* const std::vector< CustomHeader > & *headers,* const std::vector< uint8_t > & *reqPayload*) [virtual]

An application handling this indication should perform the HTTP request and call the IHttpTransactionManager::sendHttpTransactionReq to provide the result of the HTTP transaction.

On platforms with access control enabled, the client needs to have TELUX_TEL_SIM_PROFILE_HTTP_PROXY permission to invoke this API successfully.

Parameters

in	*url*	URL to sent HTTP post request.
in	*tokenId*	Token identifier.
in	*headers*	Header information to be sent along with HTTP post request.
in	*reqPayload*	Request payload.

4.13.1.3 class telux::tel::IHttpTransactionManager

IHttpTransactionManager is the interface to service HTTP related requests from the modem for SIM profile update related operations.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual telux::common::Status sendHttpTransactionResult (uint32_t token, HttpResult result, const std::vector< CustomHeader > &headers, const std::vector< uint8_t > &response, common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status registerListener (std::weak_ptr< IHttpTransactionListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< IHttpTransactionListener > listener)=0
- virtual ~IHttpTransactionManager ()

4.13.1.3.1 Constructors and Destructors

4.13.1.3.1.1 virtual telux::tel::IHttpTransactionManager::~IHttpTransactionManager ( ) [virtual]

Destructor for IHttpTransactionManager

4.13.1.3.2 Member Function Documentation

4.13.1.3.2.1 virtual telux::common::ServiceStatus telux::tel::IHttpTransactionManager::getServiceStatus ( ) [pure virtual]

This status indicates whether the IHttpTransactionManager object is in a usable state.

Returns

SERVICE_AVAILABLE - If HTTP transaction manager is ready for service.
SERVICE_UNAVAILABLE - If HTTP transaction manager is temporarily unavailable.
SERVICE_FAILED - If HTTP transaction manager encountered an irrecoverable failure.

4.13.1.3.2.2 virtual telux::common::Status telux::tel::IHttpTransactionManager::sendHttpTransaction↩ Result (uint32_t *token*, HttpResult *result*, const std::vector< CustomHeader > & *headers*, const std::vector< uint8_t > & *response*, common::ResponseCallback *callback* = *nullptr*) [pure virtual]

Send the result of HTTP Post request transaction to modem.

On platforms with access control enabled, caller needs to have
TELUX_TEL_SIM_PROFILE_HTTP_PROXY permission to invoke this API successfully.

Parameters

in	*token*	Token identifier for request and response pair.
in	*result*	HTTP transaction request result.
in	*headers*	Custom Headers in HTTP Response.
in	*response*	HTTP response payload.
in	*callback*	Callback function to get the result of send HTTP transaction request.

Returns

Status of send HTTP transaction result i.e. success or suitable error code.

4.13.1.3.2.3 virtual telux::common::Status telux::tel::IHttpTransactionManager::registerListener (std::weak_ptr< IHttpTransactionListener > *listener*) [pure virtual]

Register a listener for specific events like perform HTTP Post request.

Parameters

in	*listener*	Pointer of IHttpTransactionListener object that processes the notification.

Returns

Status of registerHttpListener success or suitable status code.

4.13.1.3.2.4 virtual telux::common::Status telux::tel::IHttpTransactionManager::deregisterListener (std::weak_ptr< IHttpTransactionListener > *listener*) [pure virtual]

De-register the listener.

Parameters

in	*listener*	Pointer of IHttpTransactionListener object that needs to be deregistered.

Returns

Status of deregisterHttpListener success or suitable status code.

4.13.1.4 class telux::tel::SimProfile

SimProfile class represents single eUICC profile on the card.

Public member functions

- SimProfile (int profileId, ProfileType profileType, const std::string &iccid, bool isActive, const std::string &nickName, const std::string &spn, const std::string &name, IconType iconType, std::vector< uint8_t > icon, ProfileClass profileClass, PolicyRuleMask policyRuleMask)
- int getSlotId ()
- int getProfileId ()
- ProfileType getType ()
- const std::string & getIccid ()
- bool isActive ()
- const std::string & getNickName ()
- const std::string & getSPN ()
- const std::string & getName ()
- IconType getIconType ()
- std::vector< uint8_t > getIcon ()
- ProfileClass getClass ()
- PolicyRuleMask getPolicyRule ()
- std::string toString ()

4.13.1.4.1 Constructors and Destructors

4.13.1.4.1.1 telux::tel::SimProfile::SimProfile (int *profileId,* ProfileType *profileType,* const std::string & *iccid,* bool *isActive,* const std::string & *nickName,* const std::string & *spn,* const std::string & *name,* IconType *iconType,* std::vector< uint8_t > *icon,* ProfileClass *profileClass,* PolicyRuleMask *policyRuleMask*)

4.13.1.4.2 Member Function Documentation

4.13.1.4.2.1 int telux::tel::SimProfile::getSlotId ( )

Get slot id associated for this profile

Returns

SlotId

4.13.1.4.2.2 int telux::tel::SimProfile::getProfileId ( )

Get profile identifier. The profile identifier is not persistently unique. It is unique for given snapshot of SIM profiles state. The profile identifier could change when any profile is deleted and added.

Returns

unique identifier for the profile

4.13.1.4.2.3 ProfileType telux::tel::SimProfile::getType ( )

Get profile Type.

Returns

profile type

4.13.1.4.2.4 const std::string& telux::tel::SimProfile::getIccid ( )

Get profile ICCID.

Returns

profile ICCID coded as in EF-ICCID

4.13.1.4.2.5 bool telux::tel::SimProfile::isActive ( )

Indicates the profile state whether active or not.

Returns

true if profile is Active

4.13.1.4.2.6 const std::string& telux::tel::SimProfile::getNickName ( )

Get profile nick name.

Returns

profile nick name

4.13.1.4.2.7 const std::string& telux::tel::SimProfile::getSPN ( )

Get profile service provider name.

Returns

profile service provider name.

4.13.1.4.2.8 const std::string& telux::tel::SimProfile::getName ( )

Get profile name.

Returns

profile name

4.13.1.4.2.9 IconType telux::tel::SimProfile::getIconType ( )

Get profile icon type.

Returns

profile icon type

4.13.1.4.2.10 std::vector<uint8_t> telux::tel::SimProfile::getIcon ( )

Get profile icon content.

Returns

profile icon content

4.13.1.4.2.11 ProfileClass telux::tel::SimProfile::getClass ( )

Get profile class.

Returns

profile class

4.13.1.4.2.12 PolicyRuleMask telux::tel::SimProfile::getPolicyRule ( )

Get profile policy rules.

Returns

mask of profile policy rules

4.13.1.4.2.13 std::string telux::tel::SimProfile::toString ( )

Get the text related informative representation of this object.

Returns

String containing informative string.

4.13.1.5 class telux::tel::ISimProfileListener

The interface listens for profile download indication and keep track of download and install progress of profile.

The methods in the listener can be invoked from multiple threads. It is client's responsibility to make sure the implementation is thread safe.

Public member functions

- virtual void onDownloadStatus (SlotId slotId, DownloadStatus status, DownloadErrorCause cause)
- virtual void onUserDisplayInfo (SlotId slotId, bool userConsentRequired, PolicyRuleMask mask)
- virtual void onConfirmationCodeRequired (SlotId slotId, std::string profileName)
- virtual ~ISimProfileListener ()

4.13.1.5.1 Constructors and Destructors

4.13.1.5.1.1 virtual telux::tel::ISimProfileListener::~ISimProfileListener ( ) [virtual]

Destructor of ISimProfileListener

4.13.1.5.2 Member Function Documentation

4.13.1.5.2.1 virtual void telux::tel::ISimProfileListener::onDownloadStatus (SlotId *slotId,* DownloadStatus *status,* DownloadErrorCause *cause*) [virtual]

This function is called when indication about status of profile download and installation comes.

On platforms with access control enabled, the client needs to have TELUX_TEL_SIM_PROFILE_OPS permission to invoke this API successfully.

Parameters

in	*slotId*	Slot on which profile get downloaded and installed.
in	*status*	telux::tel::DownloadStatus.
in	*cause*	telux::tel::DownloadErrorCause.

4.13.1.5.2.2 virtual void telux::tel::ISimProfileListener::onUserDisplayInfo (SlotId *slotId,* bool *userConsentRequired,* PolicyRuleMask *mask*) [virtual]

This function is invoked when information about user consent and profile policy rules is received. The client application is expected to provide user consent for download and install profile by calling telux::tel::ISimProfileManager::provideUserConsent if user consent is expected.

On platforms with access control enabled, the client needs to have TELUX_TEL_SIM_PROFILE_OPS permission to invoke this API successfully.

Parameters

in	*slotId*	Slot on which profile get downloaded and installed.
in	*userConsent↩ Required*	User consent required or not. If true it means user is expected to provide consent for download and install.
in	*mask*	telux::tel::PolicyRuleMask (Profile policy rules Mask)

4.13.1.5.2.3 virtual void telux::tel::ISimProfileListener::onConfirmationCodeRequired (SlotId *slotId,* std::string *profileName*) [virtual]

This function is invoked when confirmation code is required. The client application is expected to provide confirmation code for download and install profile by calling telux::tel::ISimProfileManager::provideConfirmationCode.

On platforms with access control enabled, the client needs to have TELUX_TEL_SIM_PROFILE_OPS permission to invoke this API successfully.

Parameters

in	*slotId*	Slot on which profile get downloaded and installed.
in	*profileName*	Profile name corresponding to which confirmation code is required.

4.13.1.6 class telux::tel::ISimProfileManager

ISimProfileManager is a primary interface for remote eUICCs (eSIMs or embedded SIMs) provisioning.This interface provides APIs to add, delete, set profile, update nickname, provide user consent, get Eid on the eUICC.

Public member functions

- virtual bool isSubsystemReady ()=0
- virtual std::future< bool > onSubsystemReady ()=0
- virtual telux::common::Status addProfile (SlotId slotId, const std::string &activationCode, const std::string &confirmationCode="", bool userConsentSupported=false, common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status deleteProfile (SlotId slotId, int profileId, common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status setProfile (SlotId slotId, int profileId, bool enable=false, common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status updateNickName (SlotId slotId, int profileId, const std::string &nickName, common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestProfileList (SlotId slotId, ProfileListResponseCb callback)=0
- virtual telux::common::Status requestEid (SlotId slotId, EidResponseCb callback)=0
- virtual telux::common::Status provideUserConsent (SlotId slotId, bool userConsent,

UserConsentReasonType reason, common::ResponseCallback callback=nullptr)=0

- virtual telux::common::Status provideConfirmationCode (SlotId slotId, std::string code, common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestServerAddress (SlotId slotId, ServerAddressResponseCb callback)=0
- virtual telux::common::Status setServerAddress (SlotId slotId, const std::string &smdpAddress, common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status memoryReset (SlotId slotId, ResetOptionMask mask, common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status registerListener (std::weak_ptr< ISimProfileListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< ISimProfileListener > listener)=0
- virtual ~ISimProfileManager ()

4.13.1.6.1 Constructors and Destructors

4.13.1.6.1.1 virtual telux::tel::ISimProfileManager::~ISimProfileManager ( ) [virtual]

Destructor for ISimProfileManager

4.13.1.6.2 Member Function Documentation

4.13.1.6.2.1 virtual bool telux::tel::ISimProfileManager::isSubsystemReady ( ) [pure virtual]

Checks if the eUICC subsystem is ready.

Returns

True if ISimProfileManager is ready for service, otherwise returns false.

4.13.1.6.2.2 virtual std::future<bool> telux::tel::ISimProfileManager::onSubsystemReady ( ) [pure virtual]

Wait for eUICC subsystem to be ready.

Returns

A future that caller can wait on to be notified when card manager is ready.

4.13.1.6.2.3 virtual telux::common::Status telux::tel::ISimProfileManager::addProfile (SlotId *slotId*, const std::string & *activationCode*, const std::string & *confirmationCode* = "", bool *userConsentSupported* = false, common::ResponseCallback *callback* = nullptr) [pure virtual]

Add new profile to eUICC card and download and install the profile on eUICC.

On platforms with access control enabled, caller needs to have TELUX_TEL_SIM_PROFILE_OPS permission to invoke this API successfully.

Parameters

in	*slotId*	Slot identifier corresponding to the card.
in	*activationCode*	Activation code.
in	*confirmationCode*	Optional confirmation code required for downloading the profile.
in	*userConsent↩ Supported*	Optional User consent supported or not.
in	*callback*	Optional callback function to get the result of add profile.

Returns

Status of add profile i.e. success or suitable error code.

4.13.1.6.2.4 virtual telux::common::Status telux::tel::ISimProfileManager::deleteProfile (SlotId *slotId,* int *profileId,* common::ResponseCallback *callback* = `nullptr`) `[pure virtual]`

Delete profile from eUICC card.

1. Deletion of enabled profile a) This API will disable the profile first and then delete it. b) The profile is associated with profile policy rules(PPRs) so before disabling the profile, this API checks if the PPRs telux::tel::PolicyRuleType allow the operation. c) If the policy rules are not set, then first disabling of profile happens followed by deletion of profile. d) If disable succeeds but deletion fails, then the API attempts to roll back the profile back to the original (enabled) state. e) If rollback fails due to any reason such as eUICC being in incompatabile state then the profile will be in disabled state and the API will return telux::common::ErrorCode::ROLLBACK_FAILED
2. Deletion of disabled profile a) This API checks the PPR telux::tel::PolicyRuleType::PROFILE_DELETE_NOT_ALLOWED before deletion of profile. b) If the PPR is not set, then deletion of profile is performed. If the PPR is set, then the API returns telux::common::ErrorCode::OPERATION_NOT_ALLOWED.

On platforms with access control enabled, caller needs to have TELUX_TEL_SIM_PROFILE_OPS permission to invoke this API successfully.

Parameters

in	*slotId*	Slot identifier corresponding to the card.
in	*profileId*	Profile identifier
in	*callback*	Optional callback function to get the result of delete profile.

Returns

Status of delete profile i.e. success or suitable error code.

4.13.1.6.2.5 virtual telux::common::Status telux::tel::ISimProfileManager::setProfile (SlotId *slotId,* int *profileId,* bool *enable = false,* common::ResponseCallback *callback = nullptr*) [pure virtual]

Enable or disable profile which allows to switch to other profile on eUICC card.

On platforms with access control enabled, caller needs to have TELUX_TEL_SIM_PROFILE_OPS permission to invoke this API successfully.

Parameters

in	*slotId*	Slot identifier corresponding to the card.
in	*profileId*	Profile identifier.
in	*enable*	Indicates whether a profile must be enabled or disabled. true - Enable and false - Disable.
in	*callback*	Optional callback function to get the result of set profile.

Returns

Status of set profile i.e. success or suitable error code.

4.13.1.6.2.6 virtual telux::common::Status telux::tel::ISimProfileManager::updateNickName (SlotId *slotId,* int *profileId,* const std::string & *nickName,* common::ResponseCallback *callback = nullptr*) [pure virtual]

Update nick name of the profile.

On platforms with access control enabled, caller needs to have TELUX_TEL_SIM_PROFILE_OPS permission to invoke this API successfully.

Parameters

in	*slotId*	Slot identifier corresponding to the card.
in	*profileId*	Profile identifier
in	*nickName*	New nick name for profile.
in	*callback*	Optional callback function to get the result of update nickname.

Returns

Status of update nick name i.e. success or suitable error code.

4.13.1.6.2.7 virtual telux::common::Status telux::tel::ISimProfileManager::requestProfileList (SlotId *slotId,* ProfileListResponseCb *callback*) [pure virtual]

Request list of profiles supported by the eUICC card.

On platforms with access control enabled, caller needs to have TELUX_TEL_SIM_PROFILE_READ permission to invoke this API successfully.

Parameters

in	*slotId*	Slot identifier corresponding to the card.
in	*callback*	Callback function to get the result of request profile list.

Returns

Status of request profile list i.e. success or suitable error code.

4.13.1.6.2.8 virtual telux::common::Status telux::tel::ISimProfileManager::requestEid (SlotId *slotId,* EidResponseCb *callback*) [pure virtual]

Request eUICC identifier(EID) for the slot.

On platforms with access control enabled, caller needs to have TELUX_TEL_SIM_PROFILE_READ permission to invoke this API successfully.

Parameters

in	*slotId*	Slot identifier corresponding to the card.
in	*callback*	Callback function to get the result of request EID.

Returns

Status of request EID i.e. success or suitable error code.

Deprecated

Use telux::tel::ICard::requestEid API instead

4.13.1.6.2.9 virtual telux::common::Status telux::tel::ISimProfileManager::provideUserConsent (SlotId *slotId,* bool *userConsent,* UserConsentReasonType *reason,* common::ResponseCallback *callback = nullptr*) [pure virtual]

Provide user consent required for downloading and installing profile. This API should be called in response to telux::tel::ISimProfileListener::onUserDisplayInfo.

On platforms with access control enabled, caller needs to have TELUX_TEL_SIM_PROFILE_USER_CONSENT permission to invoke this API successfully.

Parameters

in	*slotId*	Slot identifier corresponding to the card.
in	*userConsent*	Consent for profile download and install. True means user consent given to download and install.
in	*reason*	Reason for not providing user consent to download and install. telux::tel::UserConsentReasonType
in	*callback*	Optional callback function to get the result of user consent request.

Returns

Status of user consent request i.e. success or suitable error code.

4.13.1.6.2.10 virtual telux::common::Status telux::tel::ISimProfileManager::provideConfirmationCode (SlotId *slotId,* std::string *code,* common::ResponseCallback *callback = nullptr*) [pure virtual]

Provide confirmation code required for downloading and installing profile. This API should be called in response to telux::tel::ISimProfileListener::onConfirmationCodeRequired.

On platforms with access control enabled, caller needs to have TELUX_TEL_SIM_PROFILE_OPS permission to invoke this API successfully.

Parameters

in	*slotId*	Slot identifier corresponding to the card.
in	*code*	Confirmation code for profile download and install.
in	*callback*	Optional callback function to get the result of confirmation request.

Returns

Status of provide confirmation code i.e. success or suitable error code.

4.13.1.6.2.11 virtual telux::common::Status telux::tel::ISimProfileManager::requestServerAddress (SlotId *slotId,* ServerAddressResponseCb *callback*) [pure virtual]

Get Subscription Manager Data Preparation (SM-DP+) address and the Subscription Manager Discovery Server (SMDS) address configured on the eUICC.

On platforms with access control enabled, caller needs to have TELUX_TEL_SIM_PROFILE_READ permission to invoke this API successfully.

Parameters

in	*slotId*	Slot identifier corresponding to the card.
in	*callback*	Callback function to get the result of server address request.

Returns

Status of server address request i.e. success or suitable error code.

4.13.1.6.2.12 virtual telux::common::Status telux::tel::ISimProfileManager::setServerAddress (SlotId *slotId,* const std::string & *smdpAddress,* common::ResponseCallback *callback = nullptr*) [pure virtual]

Set Subscription Manager Data Preparation (SM-DP+) address on the eUICC. If SMDP+ address length is zero then the existing SM-DP+ address on the eUICC is removed.

On platforms with access control enabled, caller needs to have TELUX_TEL_SIM_PROFILE_CONFIG

permission to invoke this API successfully.

Parameters

in	*slotId*	Slot identifier corresponding to the card.
in	*smdpAddress*	SM-DP+ address to be configured on the eUICC.
in	*callback*	Optional Callback function to get the result of set SM-DP+ request.

Returns

Status of set server address request i.e. success or suitable error code.

4.13.1.6.2.13 virtual telux::common::Status telux::tel::ISimProfileManager::memoryReset (SlotId *slotId*, ResetOptionMask *mask*, common::ResponseCallback *callback* = `nullptr`) `[pure virtual]`

Resets the memory of the eUICC card based on telux::tel::ResetOptionMask.

On platforms with access control enabled, caller needs to have TELUX_TEL_SIM_PROFILE_OPS permission to invoke this API successfully.

Parameters

in	*slotId*	Slot identifier corresponding to the card.
in	*mask*	Memory reset options mask telux::tel::ResetOptionMask
in	*callback*	Optional Callback function to get the result of memory reset request.

Returns

Status of memory reset request i.e. success or suitable error code.

4.13.1.6.2.14 virtual telux::common::Status telux::tel::ISimProfileManager::registerListener (std::weak_ptr< ISimProfileListener > *listener*) `[pure virtual]`

Register a listener to listen for status of specific events like download and installation of profile on eUICC.

Parameters

in	*listener*	Pointer of ISimProfileListener object that processes the notification.

Returns

Status of registerListener success or suitable status code

4.13.1.6.2.15 virtual telux::common::Status telux::tel::ISimProfileManager::deregisterListener (std::weak_ptr< ISimProfileListener > *listener*) [pure virtual]

De-register the listener.

Parameters

in	*listener*	Pointer of ISimProfileListener object that needs to be removed

Returns

Status of deregisterListener success or suitable status code

4.13.2 Enumeration Type Documentation

4.13.2.1 enum telux::tel::HttpResult [strong]

Defines the HTTP request result.

Enumerator

TRANSACTION_SUCCESSFUL HTTP request successful

UNKNOWN_ERROR Unknown error

HTTP_SERVER_ERROR Server error

HTTP_TLS_ERROR TLS error

HTTP_NETWORK_ERROR Network error

4.13.2.2 enum telux::tel::ProfileType [strong]

Indicates profile type of card

Enumerator

UNKNOWN
REGULAR Regular profile
EMERGENCY Emergency profile

4.13.2.3 enum telux::tel::IconType [strong]

Indicates profile icon type.

Enumerator

NONE No icon information
JPEG JPEG icon
PNG PNG icon

4.13.2.4 enum telux::tel::ProfileClass [strong]

Indicates profile class.

Enumerator

UNKNOWN No info about profile class
TEST Test profile
PROVISIONING Provisioning profile
OPERATIONAL Operational profile

4.13.2.5 enum telux::tel::DownloadStatus [strong]

Indicates profile download status.

Enumerator

DOWNLOAD_ERROR Profile download error
DOWNLOAD_INSTALLATION_COMPLETE Profile download and installation is complete

4.13.2.6 enum telux::tel::DownloadErrorCause [strong]

Indicates profile download error cause.

Enumerator

GENERIC Generic error
SIM Error from the SIM card
NETWORK Error from the network
MEMORY Error due to no memory
UNSUPPORTED_PROFILE_CLASS Unsupported profile class
PPR_NOT_ALLOWED Profile policy rules not allowed
END_USER_REJECTION End user rejection
END_USER_POSTPONED End user postponed

4.13.2.7 enum telux::tel::UserConsentReasonType [strong]

Indicates the reason for user consent not provided.

Enumerator

END_USER_REJECTION End user rejection
END_USER_POSTPONED End user postponed

4.13.2.8 enum telux::tel::PolicyRuleType

Defines profile policy rules(PPR). Each value represents corresponding bit for PprMask bitset.

Enumerator

PROFILE_DISABLE_NOT_ALLOWED Disabling of the profile is not allowed
PROFILE_DELETE_NOT_ALLOWED Deletion of the profile is not allowed

PROFILE_DELETE_ON_DISABLE Deletion of the profile is required on successful disabling

4.13.2.9 enum telux::tel::ResetOption

Defines memory reset options. Each value represents corresponding bit for ResetOptionMask bitset.

Enumerator

TEST_PROFILES Delete all the test profiles
OPERATIONAL_PROFILE Delete all operational profiles
DEFAULT_SMDP_ADDRESS Reset the default SM-DP+ address

4.14 Remote SIM

This section contains APIs related to Remote SIM operations.

4.14.1 Data Structure Documentation

4.14.1.1 class telux::tel::IRemoteSimListener

A listener class for getting remote SIM notifications.

The methods in listener can be invoked from multiple different threads. The implementation should be thread safe.

Public member functions

- virtual void onApduTransfer (const unsigned int id, const std::vector< uint8_t > &apdu)
- virtual void onCardConnect ()
- virtual void onCardDisconnect ()
- virtual void onCardPowerUp ()
- virtual void onCardPowerDown ()
- virtual void onCardReset ()
- virtual ~IRemoteSimListener ()

4.14.1.1.1 Constructors and Destructors

4.14.1.1.1.1 virtual telux::tel::IRemoteSimListener::~IRemoteSimListener ( ) [virtual]

Destructor of IRemoteSimListener

4.14.1.1.2 Member Function Documentation

4.14.1.1.2.1 virtual void telux::tel::IRemoteSimListener::onApduTransfer (const unsigned int *id*, const std::vector< uint8_t > & *apdu*) [virtual]

This function is called when the modem wants to transmit a command APDU.

Parameters

in	*id*	Identifier for a command and response APDU pair
in	*apdu*	APDU request sent to the control point (max size = 261, per ETSI TS 102 221, section 10.1.4)

4.14.1.1.2.2 virtual void telux::tel::IRemoteSimListener::onCardConnect ( ) [virtual]

This function is called when the modem wants to establish a connection.

4.14.1.1.2.3 virtual void telux::tel::IRemoteSimListener::onCardDisconnect ( ) [virtual]

This function is called when the modem wants to tear down a connection.

4.14.1.1.2.4 virtual void telux::tel::IRemoteSimListener::onCardPowerUp ( ) [virtual]

This function is called when the modem wants to power up the card.

4.14.1.1.2.5 virtual void telux::tel::IRemoteSimListener::onCardPowerDown ( ) [virtual]

This function is called when the modem wants to power down the card.

4.14.1.1.2.6 virtual void telux::tel::IRemoteSimListener::onCardReset ( ) [virtual]

This function is called when the modem wants to warm reset the card.

4.14.1.2 class telux::tel::IRemoteSimManager

IRemoteSimManager provides APIs for remote SIM related operations. This allows a device to use a SIM card on another device for its WWAN modem functionality. The SIM provider service is the endpoint that interfaces with the SIM card (e.g. over bluetooth) and sends/receives data to the other endpoint, the modem. The modem sends requests to the SIM provider service to interact with the SIM card (e.g. power up, transmit APDU, etc.), and is notified of events (e.g. card errors, resets, etc.). This API is used by the SIM provider endpoint to provide a SIM card to the modem.

Public member functions

- virtual bool isSubsystemReady ()=0
- virtual std::future< bool > onSubsystemReady ()=0
- virtual telux::common::Status sendReset (telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status sendConnectionAvailable (telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status sendConnectionUnavailable (telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status sendCardReset (const std::vector< uint8_t > &atr, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status sendCardError (const CardErrorCause cause=CardErrorCause::INVALID, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status sendCardInserted (const std::vector< uint8_t > &atr, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status sendCardRemoved (telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status sendCardWakeup (telux::common::ResponseCallback callback=nullptr)=0

- virtual telux::common::Status sendApdu (const unsigned int id, const std::vector< uint8_t > &apdu, const bool isSuccess=true, const unsigned int totalSize=0, const unsigned int offset=0, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status registerListener (std::weak_ptr< IRemoteSimListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< IRemoteSimListener > listener)=0
- virtual int getSlotId ()=0
- virtual ~IRemoteSimManager ()

4.14.1.2.1 Constructors and Destructors

4.14.1.2.1.1 virtual telux::tel::IRemoteSimManager::~IRemoteSimManager ( ) [virtual]

Destructor of IRemoteSimManager

4.14.1.2.2 Member Function Documentation

4.14.1.2.2.1 virtual bool telux::tel::IRemoteSimManager::isSubsystemReady ( ) [pure virtual]

Checks the status of remote SIM subsystem and returns the result.

Returns

True if remote SIM subsystem is ready for service otherwise false.

4.14.1.2.2.2 virtual std::future<bool> telux::tel::IRemoteSimManager::onSubsystemReady ( ) [pure virtual]

Wait for remote SIM subsystem to be ready.

Returns

A future that caller can wait on to be notified when remote SIM subsystem is ready.

4.14.1.2.2.3 virtual telux::common::Status telux::tel::IRemoteSimManager::sendReset (telux::common::ResponseCallback *callback* = *nullptr*) [pure virtual]

Send reset command to the modem to reset state variables.

Parameters

out	*callback*	Callback function pointer to get the response of sendReset.

Returns

Status of sendReset i.e. success or suitable status code.

4.14.1.2.2.4 virtual telux::common::Status telux::tel::IRemoteSimManager::sendConnectionAvailable (telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

Send connection available event to the modem.

Parameters

out	*callback*	Callback function pointer to get the response.

Returns

Status of sendConnectionAvailable i.e. success or suitable status code.

4.14.1.2.2.5 virtual telux::common::Status telux::tel::IRemoteSimManager::sendConnectionUnavailable (telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

Send connection unavailable event to the modem.

Parameters

out	*callback*	Callback function pointer to get the response.

Returns

Status of sendConnectionUnavailable i.e. success or suitable status code.

4.14.1.2.2.6 virtual telux::common::Status telux::tel::IRemoteSimManager::sendCardReset (const std::vector< uint8_t > & *atr,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

Send card reset event to the modem.

Parameters

in	*atr*	Answer to Reset bytes (max size = 32, per ISO/IEC 7816-3:2006 section 8.1).
out	*callback*	Callback function pointer to get the response of sendCardReset.

Returns

Status of sendCardReset i.e. success or suitable status code.

4.14.1.2.2.7 virtual telux::common::Status telux::tel::IRemoteSimManager::sendCardError (const CardErrorCause *cause =* CardErrorCause::INVALID, telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

Send card error event to the modem.

Parameters

in	*cause*	Card Error cause.
out	*callback*	Callback function pointer to get the response of sendCardError.

Returns

Status of sendCardError i.e. success or suitable status code.

4.14.1.2.2.8 virtual telux::common::Status telux::tel::IRemoteSimManager::sendCardInserted (const std::vector< uint8_t > & *atr,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

Send card inserted event to the modem.

Parameters

in	*atr*	Answer to Reset bytes (max size = 32, per ISO/IEC 7816-3:2006 section 8.1).
out	*callback*	Callback function pointer to get the response of sendCardInserted.

Returns

Status of sendCardInserted i.e. success or suitable status code.

4.14.1.2.2.9 virtual telux::common::Status telux::tel::IRemoteSimManager::sendCardRemoved (telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

Send card removed event to the modem.

Parameters

out	*callback*	Callback function pointer to get the response of sendCardRemoved.

Returns

Status of sendCardRemoved i.e. success or suitable status code.

4.14.1.2.2.10 virtual telux::common::Status telux::tel::IRemoteSimManager::sendCardWakeup (telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

Send card wakeup event to the modem.

Parameters

out	*callback*	Callback function pointer to get the response of sendCardWakeup.

Returns

Status of sendCardWakeup i.e. success or suitable status code.

4.14.1.2.2.11 virtual telux::common::Status telux::tel::IRemoteSimManager::sendApdu (const unsigned int *id*, const std::vector< uint8_t > & *apdu*, const bool *isSuccess* = `true`, const unsigned int *totalSize* = `0`, const unsigned int *offset* = `0`, telux::common::↩ ResponseCallback *callback* = `nullptr`) `[pure virtual]`

Sends an APDU message to the modem, in response to a previous APDU sent by the modem.

Parameters

in	*id*	Identifier for command and response APDU pair.
in	*apdu*	Response APDU (max size = 1024).
in	*isSuccess*	Whether APDU transaction completed successfully.
in	*totalSize*	Total length of the APDU message (used when the response is larger than 1024 bytes and must be passed in multiple segments).
in	*offset*	Offset of this APDU segment in the original message.
out	*callback*	Callback function pointer to get the response of sendApdu.

Returns

Status of sendApdu i.e. success or suitable status code.

4.14.1.2.2.12 virtual telux::common::Status telux::tel::IRemoteSimManager::registerListener (std::weak_ptr< IRemoteSimListener > *listener*) `[pure virtual]`

Register a listener for specific updates from the modem.

Parameters

in	*listener*	Pointer of IRemoteSimListener object that processes the notification

Returns

Status of registerListener i.e success or suitable status code.

4.14.1.2.2.13 virtual telux::common::Status telux::tel::IRemoteSimManager::deregisterListener (std::weak_ptr< IRemoteSimListener > *listener*) `[pure virtual]`

Deregister the previously added listener.

Parameters

in	*listener*	Previously registered IRemoteSimListener that needs to be deregistered

Returns

Status of deregisterListener success or suitable status code

4.14.1.2.2.14 virtual int telux::tel::IRemoteSimManager::getSlotId ( ) [pure virtual]

Get associated slot ID for the RemoteSimManager

Returns

The slot ID associated with this IRemoteSimManager

4.14.2 Enumeration Type Documentation

4.14.2.1 enum telux::tel::CardErrorCause [strong]

Defines the card error cause, sent to the modem by the SIM provider

Enumerator

INVALID Card error cause value will not be passed to modem
UNKNOWN_ERROR Unknown error
NO_LINK_ESTABLISHED No link was established
COMMAND_TIMEOUT Command timeout
POWER_DOWN Error due to a card power down

4.15 Location

- Location Services

This section contains APIs related to Location.

4.16 Location Services

This section contains APIs related to Location Services.

4.16.1 Data Structure Documentation

4.16.1.1 class telux::loc::IDgnssStatusListener

Listener class for getting RTCM injection event notification information.

Public member functions

- virtual void onDgnssStatusUpdate (DgnssStatus status)
- virtual ~IDgnssStatusListener ()

4.16.1.1.1 Constructors and Destructors

4.16.1.1.1.1 virtual telux::loc::IDgnssStatusListener::~IDgnssStatusListener ( ) [virtual]

Destructor of IRTCMStatusListener

4.16.1.1.2 Member Function Documentation

4.16.1.1.2.1 virtual void telux::loc::IDgnssStatusListener::onDgnssStatusUpdate (DgnssStatus *status*) [virtual]

This function is called asynchronously to report RTCM injection status

Parameters

in	*status*	- the status enum.

4.16.1.2 class telux::loc::IDgnssManager

IRtcmManager provides interface to inject RTCM data into modem, register event listener reported by cdfw(correction data framework).

Public member functions

- virtual bool isSubsystemReady ()=0
- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual std::future< bool > onSubsystemReady ()=0
- virtual telux::common::Status registerListener (std::weak_ptr< IDgnssStatusListener > listener)=0
- virtual telux::common::Status deRegisterListener (void)=0
- virtual telux::common::Status createSource (DgnssDataFormat dataFormat)=0

- virtual telux::common::Status releaseSource (void)=0
- virtual telux::common::Status injectCorrectionData (const uint8_t *buffer, uint32_t bufferSize)=0
- virtual ~IDgnssManager ()

4.16.1.2.1 Constructors and Destructors

4.16.1.2.1.1 virtual telux::loc::IDgnssManager::~IDgnssManager ( ) [virtual]

Destructor of IRtcmManager

4.16.1.2.2 Member Function Documentation

4.16.1.2.2.1 virtual bool telux::loc::IDgnssManager::isSubsystemReady ( ) [pure virtual]

Checks the status of location subsystems and returns the result.

Returns

True if location subsystem is ready for service otherwise false.

Deprecated

use getServiceStatus()

4.16.1.2.2.2 virtual telux::common::ServiceStatus telux::loc::IDgnssManager::getServiceStatus ( ) [pure virtual]

This status indicates whether the object is in a usable state.

Returns

SERVICE_AVAILABLE - If Dgnss manager is ready for service. SERVICE_UNAVAILABLE - If Dgnss manager is temporarily unavailable. SERVICE_FAILED - If Dgnss manager encountered an irrecoverable failure.

4.16.1.2.2.3 virtual std::future<bool> telux::loc::IDgnssManager::onSubsystemReady ( ) [pure virtual]

Wait for location subsystem to be ready.

Returns

A future that caller can wait on to be notified when location subsystem is ready.

Deprecated

The callback mechanism introduced in the LocationFactory::getDgnssManager() API will provide the

similar notification mechanism as onSubsystemReady(). This API will soon be removed from further releases.

4.16.1.2.2.4 virtual telux::common::Status telux::loc::IDgnssManager::registerListener (std::weak_ptr< IDgnssStatusListener > *listener*) [pure virtual]

Register a listener for Dgnss injection status update.

Parameters

in	*listener*	- Pointer of IDgnssStatusListener object that processes the notification.

Returns

Status of registerListener i.e success or suitable status code.

4.16.1.2.2.5 virtual telux::common::Status telux::loc::IDgnssManager::deRegisterListener (void) [pure virtual]

deRegister a listener for Dgnss injection status update.

Returns

Status of registerListener i.e success or suitable status code.

4.16.1.2.2.6 virtual telux::common::Status telux::loc::IDgnssManager::createSource (DgnssDataFormat *dataFormat*) [pure virtual]

Create a Dgnss injection source. Only one source is permitted at any given time. If a new source is to be used, user must call releaseSource() to release previous source before calling this function.

Parameters

in	*dataFormat*	Dgnss injection data format.

Returns

Success of suitable status code

4.16.1.2.2.7 virtual telux::common::Status telux::loc::IDgnssManager::releaseSource (void) [pure virtual]

Release current Dgnss injection source (previously created by createSource() call) This function is to be called if it's determined that current injection data is not suitable anymore, and a new source will be created and used as injection source.

Returns

none

4.16.1.2.2.8 virtual telux::common::Status telux::loc::IDgnssManager::injectCorrectionData (const uint8_t * *buffer*, uint32_t *bufferSize*) [pure virtual]

Inject correction data This function is to be called when a source has been created, either through a explicit call to createSource(), or after DgnssManager object was instantiated through the factory method(The factory method create a default source for DgnssManager object).

Parameters

in	*buffer*	buffer contains the data to be injected.
in	*bufferSize*	size of the buffer.

Returns

success or suitable status code.

4.16.1.3 class telux::loc::ILocationConfigurator

ILocationConfigurator allows general engine configurations (example: TUNC, PACE etc), configuration of specific engines like SPE (example: minSVElevation, minGPSWeek etc) or DRE, deletion of warm and cold aiding data, NMEA configuration and support for XTRA feature. ILocationConfigurator APIs strictly adheres to the principle of single client per process.

Public Types

- using GetSecondaryBandCallback = std::function< void(const telux::loc::ConstellationSet set, telux::common::ErrorCode error)>
- using GetMinGpsWeekCallback = std::function< void(uint16_t minGpsWeek, telux::common::ErrorCode error)>
- using GetMinSVElevationCallback = std::function< void(uint8_t minSVElevation, telux::common::ErrorCode error)>
- using GetRobustLocationCallback = std::function< void(const telux::loc::RobustLocationConfiguration rLConfig, telux::common::ErrorCode error)>
- using GetXtraStatusCallback = std::function< void(const telux::loc::XtraStatus xtraStatus, telux::common::ErrorCode error)>

Public member functions

- virtual bool isSubsystemReady ()=0
- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual std::future< bool > onSubsystemReady ()=0
- virtual telux::common::Status configureCTunc (bool enable, telux::common::ResponseCallback callback=nullptr, float timeUncertainty=DEFAULT_TUNC_THRESHOLD, uint32_t energyBudget=DEFAULT_TUNC_ENERGY_THRESHOLD)=0
- virtual telux::common::Status configurePACE (bool enable, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status deleteAllAidingData (telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status configureLeverArm (const LeverArmConfigInfo &info, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status configureConstellations (const SvBlackList &list, telux::common::ResponseCallback callback=nullptr, bool resetToDefault=false)=0
- virtual telux::common::Status configureSecondaryBand (const ConstellationSet &set, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestSecondaryBandConfig (GetSecondaryBandCallback cb)=0
- virtual telux::common::Status configureRobustLocation (bool enable, bool enableForE911=false, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestRobustLocation (GetRobustLocationCallback cb)=0
- virtual telux::common::Status configureMinGpsWeek (uint16_t minGpsWeek, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestMinGpsWeek (GetMinGpsWeekCallback cb)=0
- virtual telux::common::Status configureMinSVElevation (uint8_t minSVElevation, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestMinSVElevation (GetMinSVElevationCallback cb)=0
- virtual telux::common::Status deleteAidingData (AidingData aidingDataMask, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status configureDR (const DREngineConfiguration &config, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status configureEngineState (const EngineType engineType, const LocationEngineRunState engineState, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status provideConsentForTerrestrialPositioning (bool userConsent, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status configureNmeaTypes (const NmeaSentenceConfig nmeaType, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status configureNmea (const NmeaConfig configParams,

telux::common::ResponseCallback callback=nullptr)=0

- virtual telux::common::Status configureEngineIntegrityRisk (const EngineType engineType, uint32_t integrityRisk, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status configureXtraParams (bool enable, const XtraConfig configParams, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestXtraStatus (GetXtraStatusCallback callback)=0
- virtual telux::common::Status registerListener (LocConfigIndications indicationList, std::weak_ptr< ILocationConfigListener > listener)=0
- virtual telux::common::Status deRegisterListener (LocConfigIndications indicationList, std::weak_ptr< ILocationConfigListener > listener)=0
- virtual ~ILocationConfigurator ()

4.16.1.3.1 Member Typedef Documentation

4.16.1.3.1.1 using telux::loc::ILocationConfigurator::GetSecondaryBandCallback = std::function<void(const telux::loc::ConstellationSet set, telux::common::ErrorCode error)>

This function is called with the response to requestSecondaryBandConfig API.

Parameters

in	*set*	- disabled secondary band constellation configuration used by the GNSS standard position engine (SPE).
in	*error*	- Return code which indicates whether the operation succeeded or not.

4.16.1.3.1.2 using telux::loc::ILocationConfigurator::GetMinGpsWeekCallback = std::function<void(uint16_t minGpsWeek, telux::common::ErrorCode error)>

This function is called with the response to requestMinGpsWeek API.

Parameters

in	*minGpsWeek*	- minimum gps week.
in	*error*	- Return code which indicates whether the operation succeeded or not.

4.16.1.3.1.3 using telux::loc::ILocationConfigurator::GetMinSVElevationCallback = std::function<void(uint8_t minSVElevation, telux::common::ErrorCode error)>

This function is called with the response to requestMinSVElevation API.

Parameters

in	*minSVElevation*	- minimum SV Elevation angle in units of degree.
in	*error*	- Return code which indicates whether the operation succeeded or not.

4.16.1.3.1.4 using telux::loc::ILocationConfigurator::GetRobustLocationCallback = std::function<void(const telux::loc:: RobustLocationConfiguration rLConfig, telux::common::ErrorCode error)>

This function is called with the response to requestRobustLocation API.

Parameters

in	*rLConfig*	- robust location settings information.
in	*error*	- Return code which indicates whether the operation succeeded or not.

4.16.1.3.1.5 using telux::loc::ILocationConfigurator::GetXtraStatusCallback = std::function<void(const telux::loc::XtraStatus xtraStatus, telux::common::ErrorCode error)>

This function is called with the response to requestXtraStatus API.

Parameters

in	*xtraStatus*	- Information pertaining to Xtra assistance data.
in	*error*	- Return code which indicates whether the operation succeeded or not.

4.16.1.3.2 Constructors and Destructors

4.16.1.3.2.1 virtual telux::loc::ILocationConfigurator::~ILocationConfigurator ( ) [virtual]

Destructor of ILocationConfigurator

4.16.1.3.3 Member Function Documentation

4.16.1.3.3.1 virtual bool telux::loc::ILocationConfigurator::isSubsystemReady ( ) [pure virtual]

Checks the status of location configuration subsystems and returns the result.

Returns

True if location configuration subsystem is ready for service otherwise false.

Deprecated

use getServiceStatus()

4.16.1.3.3.2 virtual telux::common::ServiceStatus telux::loc::ILocationConfigurator::getServiceStatus ( ) [pure virtual]

This status indicates whether the object is in a usable state.

Returns

SERVICE_AVAILABLE - If location configurator is ready for service. SERVICE_UNAVAILABLE - If location configurator is temporarily unavailable. SERVICE_FAILED - If location configurator encountered an irrecoverable failure.

4.16.1.3.3.3 virtual std::future<bool> telux::loc::ILocationConfigurator::onSubsystemReady ( ) [pure virtual]

Wait for location configuration subsystem to be ready.

Returns

A future that caller can wait on to be notified when location configuration subsystem is ready.

Deprecated

The callback mechanism introduced in the LocationFactory::getLocationConfigurator() API will provide the similar notification mechanism as onSubsystemReady(). This API will soon be removed from further releases.

4.16.1.3.3.4 virtual telux::common::Status telux::loc::ILocationConfigurator::configureCTunc (bool *enable*, telux::common::ResponseCallback *callback* = `nullptr`, float *timeUncertainty* = DEFAULT_TUNC_THRESHOLD, uint32_t *energyBudget* = DEFAULT_TUNC_ENERGY_THRESHOLD) [pure virtual]

This API enables or disables the constrained time uncertainty(C-TUNC) feature. When the vehicle is turned off this API helps to put constraint on the time uncertainty. For multiple invocations of this API, client should wait for the command to finish, e.g.: via ResponseCallback received before issuing a second configureCTunc command. Behavior is not defined if client issues a second request of configureCTunc without waiting for the finish of the previous configureCTunc request.

On platforms with Access control enabled, caller needs to have TELUX_LOC_CONFIG permission to invoke this API successfully.

Parameters

in	*enable*	- true for enable C-TUNC feature and false for disable C-TUNC feature.
in	*callback*	- Optional callback to get the response of enablement/disablement of C-TUNC.
in	*timeUncertainty*	- specifies the time uncertainty threshold that gps engine needs to maintain, in unit of milli-seconds. This parameter is ignored when the request is to disable this feature.

in	*energyBudget*	- specifies the power budget that the GPS engine is allowed to spend to maintain the time uncertainty, in the unit of 100 micro watt second. If the power exceeds the energyBudget then this API is disabled. This is a cumulative energy budget. This parameter is ignored when the request is to disable this feature.

Returns

Status of configureCTunc i.e. success or suitable status code.

4.16.1.3.3.5 virtual telux::common::Status telux::loc::ILocationConfigurator::configurePACE (bool *enable,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

This API enables or disables position assisted clock estimator feature. For multiple invocations of this API, client should wait for the command to finish, e.g.: via ResponseCallback received before issuing a second configurePACE command. Behavior is not defined if client issues a second request of configurePACE without waiting for the finish of the previous configurePACE request.

On platforms with Access control enabled, caller needs to have TELUX_LOC_CONFIG permission to invoke this API successfully.

Parameters

in	*enable*	- to enable/disable position assisted clock estimator feature.
in	*callback*	- Optional callback to get the response of enablement/disablement of PACE.

4.16.1.3.3.6 virtual telux::common::Status telux::loc::ILocationConfigurator::deleteAllAidingData (telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

This API deletes all forms of aiding data from all position engines. This API deletes all assistance data used by GPS engine and force engine to do a cold start for next session. Invoking this API will trigger cold start of all position engines on the device and will cause significant delay for the position engines to produce next fix and may have other performance impact. So, this API should only be exercised with caution and only for very limited usage scenario, e.g.: for performance test and certification process.

On platforms with Access control enabled, caller needs to have TELUX_LOC_CONFIG permission to invoke this API successfully.

Parameters

in	*callback*	- Optional callback to get the response of delete aiding data.

4.16.1.3.3.7 virtual telux::common::Status telux::loc::ILocationConfigurator::configureLeverArm (const LeverArmConfigInfo & *info,* telux::common::ResponseCallback *callback* = `nullptr`) [pure virtual]

This API sets the lever arm parameters for the vehicle. LeverArm is sytem level parameters and it is not expected to change. So, it is needed to issue configureLeverArm once for every application processor boot-up. For multiple invocations of this API client should wait for the command to finish, e.g.: via ResponseCallback received before issuing a second configureLeverArm command. Behavior is not defined if client issues a second request of configureLeverArm without waiting for the finish of the previous configureLeverArm request.

On platforms with Access control enabled, caller needs to have TELUX_LOC_CONFIG permission to invoke this API successfully.

Parameters

in	*info*	- lever arm configuration info regarding below three types of lever arm info: a: GNSS Antenna with respect to the origin at the IMU (inertial measurement unit) for DR engine b: GNSS Antenna with respect to the origin at the IMU (inertial measurement unit) for VEPP engine c: VRP (Vehicle Reference Point) with respect to the origin (at the GNSS Antenna). Vehicle manufacturers prefer the position output to be tied to a specific point in the vehicle rather than where the antenna is placed (midpoint of the rear axle is typical).
in	*callback*	- Optional callback to get the response of configure lever arm.

4.16.1.3.3.8 virtual telux::common::Status telux::loc::ILocationConfigurator::configureConstellations (const SvBlackList & *list,* telux::common::ResponseCallback *callback* = `nullptr`, bool *resetToDefault* = `false`) [pure virtual]

This API blacklists some constellations or subset of SVs from the constellation from being used by the GNSS standard position engine (SPE). Supported constellations for this API are GLONASS, QZSS, BEIDOU, GALILEO and SBAS. For other constellations NOTSUPPORTED status will be returned. For SBAS, SVs are not used in positioning by the GNSS standard position engine (SPE) by default. Blacklisting SBAS SV only blocks SBAS data demodulation and will not disable SBAS cross-correlation detection algorithms as they are necessary for optimal GNSS standard position engine (SPE) performance. When resetToDefault is false then the list is expected to contain the constellations or SVs that should be blacklisted. An empty list could be specified to allow all constellations/SVs (i.e. none will be blacklisted) in determining the fix. When resetToDefault is set to true, the device will revert to the default list of SV/ constellations to be blacklisted. For multiple invocations of this API, client should wait for the command to finish, e.g.: via ResponseCallback received before issuing a second configureConstellations command. Behavior is not defined if client issues a second request of configureConstellations without waiting for the finish of the previous configureConstellations request. This API call is not incremental and the new settings will completely overwrite the previous call.

On platforms with Access control enabled, caller needs to have TELUX_LOC_CONFIG permission to invoke this API successfully.

Parameters

in	*list*	- specify the set of constellations and SVs that should not be used by the GNSS engine on modem. Constellations and SVs not specified in blacklistedSvList could get used by the GNSS engine on modem.
in	*callback*	- Optional callback to get the response of configure constellations.
in	*resetToDefault*	- when set to true, the device will revert to the default list of SV/constellation to be blacklisted. When set to false, list will be inspected to determine what should be blacklisted.

4.16.1.3.3.9 virtual telux::common::Status telux::loc::ILocationConfigurator::configureSecondaryBand (const ConstellationSet & *set,* telux::common::ResponseCallback *callback* = `nullptr`) `[pure virtual]`

This API configures the secondary band constellations used by the GNSS standard position engine. This API call is not incremental and the new settings will completely overwrite the previous call. The set specifies the supported constellations whose secondary band information should be disabled. The absence of a constellation in the set will result in the secondary band being enabled for that constellation. The modem has its own configuration in NV (persistent memory) about which constellation's secondary bands are allowed to be enabled. When a constellation is omitted when this API is invoked the secondary band for that constellation will only be enabled if the modem configuration allows it. If not allowed then this API would be a no-op for that constellation. Passing an empty set to this API will result in all constellations as allowed by the modem configuration to be enabled. For multiple invocations of this API, client should wait for the command to finish, e.g.: via ResponseCallback recieved, before issuing a second configureSecondaryBand command. Behavior is not defined if client issues a second request of configureSecondaryBand without waiting for the finish of the previous configureSecondaryBand request.

On platforms with Access control enabled, caller needs to have TELUX_LOC_CONFIG permission to invoke this API successfully.

Parameters

in	*set*	- specifies the set of constellations whose secondary bands need to be disabled.
in	*callback*	- Optional callback to get the response of configureSecondaryBand.

4.16.1.3.3.10 virtual telux::common::Status telux::loc::ILocationConfigurator::requestSecondary↩ BandConfig (GetSecondaryBandCallback *cb*) `[pure virtual]`

This API retrieves the secondary band configurations for constellation used by the standard GNSS engine (SPE).

Parameters

in	*cb*	- callback to retrieve secondary band information about constellations.

Returns

Status of requestSecondaryBandConfig i.e. success or suitable status code.

4.16.1.3.3.11 virtual telux::common::Status telux::loc::ILocationConfigurator::configureRobustLocation (bool *enable*, bool *enableForE911* = `false`, telux::common::ResponseCallback *callback* = `nullptr`) [pure virtual]

This API enables/disables robust location 2.0 feature and enables/disables robust location while device is on E911. When this API is enabled it reports confidence of the GNSS spoofing by the getConformityIndex() API defined in the ILocationInfoEx class, which is a measure of robustness of the underlying navigation solution. It indicates how well the various input data considered for navigation solution conform to expectations. In the presence of detected spoofed inputs, the navigation solution may take corrective actions to mitigate the spoofed inputs and improve robustness of the solution.

On platforms with Access control enabled, caller needs to have TELUX_LOC_CONFIG permission to invoke this API successfully.

Parameters

in	*enable*	- true to enable robust location and false to disable robust location.
in	*enableForE911*	- true to enable robust location when the device is on E911 session and false to disable on E911 session. This parameter is only valid if robust location is enabled.
in	*callback*	- Optional callback to get the response of configure robust location.

4.16.1.3.3.12 virtual telux::common::Status telux::loc::ILocationConfigurator::requestRobustLocation (GetRobustLocationCallback *cb*) [pure virtual]

This API retrieves the robust location 2.0 settings and version info used by the GNSS standard position engine (SPE).

Parameters

in	*cb*	- callback to retrieve robust location information.

Returns

Status of requestRobustLocation i.e. success or suitable status code.

4.16.1.3.3.13 virtual telux::common::Status telux::loc::ILocationConfigurator::configureMinGpsWeek (uint16_t *minGpsWeek*, telux::common::ResponseCallback *callback* = `nullptr`) [pure virtual]

This API configures the minimum GPS week used by the modem GNSS standard position engine (SPE) and shall not be called while GNSS SPE is in the middle of a session. Client needs to assure that there is no active GNSS SPE session prior to issuing this command. Client should wait for the command to finish, e.g.:

via ResponseCallback received before issuing a second configureMinGpsWeek command. Behavior is not defined if client issues a second request of configureMinGpsWeek without waiting for the previous configureMinGpsWeek to finish. Additionally minimum GPS week number shall NEVER be in the future of the current GPS Week.

On platforms with Access control enabled, caller needs to have TELUX_LOC_CONFIG permission to invoke this API successfully.

Parameters

in	*minGpsWeek*	- minimum GPS week to be used by modem GNSS engine.
in	*callback*	- Optional callback to get the response of configure minimum GPS week.

Returns

Status of configureMinGpsWeek i.e. success or suitable status code.

4.16.1.3.3.14 virtual telux::common::Status telux::loc::ILocationConfigurator::requestMinGpsWeek (GetMinGpsWeekCallback *cb*) [pure virtual]

This API retrieves the minimum GPS week configuration used by the modem GNSS standard position engine (SPE). If this API is called right after configureMinGpsWeek, the returned setting may not match the one specified in configureMinGpsWeek, as the setting configured via configureMinGpsWeek can not be applied to the GNSS standard position engine(SPE) when the engine is in middle of a session. In poor GPS signal condition, the session may take up to 255 seconds to finish. If after 255 seconds of invoking configureMinGpsWeek, the returned value still does not match, then the caller need to reapply the setting by invoking configureMinGpsWeek again.

Parameters

in	*cb*	- callback to retrieve the minimum gps week.

Returns

Status of requestMinGpsWeek i.e. success or suitable status code.

4.16.1.3.3.15 virtual telux::common::Status telux::loc::ILocationConfigurator::configureMinSVElevation (uint8_t *minSVElevation,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

This API configures the minimum SV elevation angle setting used by the GNSS standard position engine. Configuring minimum SV elevation setting will not cause SPE to stop tracking low elevation SVs. It only controls the list of SVs that are used in the filtered position solution, so SVs with elevation below the setting will be excluded from use in the filtered position solution. Configuring this setting to large angle will cause more SVs to get filtered out in the filtered position solution and will have negative performance impact.

This setting does not impact the SV information and SV measurement reports retrieved from APIs such as IGnssSvINfo::getSVInfoList, ILocationListener::onGnssMeasurementsInfo.

To apply the setting, the GNSS standard position engine(SPE) will require GNSS measurement engine and

position engine to be turned off briefly. This may cause glitch for on-going tracking session and may have other performance impact. So, it is advised to use this API with caution and only for very limited usage scenario, e.g.: for performance test and certification process and for one-time device configuration.

Client should wait for the command to finish, e.g.: via ResponseCallback received, before issuing a second configureMinElevation command. If this API is called while the GNSS Position Engine is in the middle of a session, ResponseCallback will still be invoked shortly to indicate the setting has been received by the SPE engine. However the actual setting can not be applied until the current session ends, and this may take up to 255 seconds in poor GPS signal condition.

On platforms with Access control enabled, caller needs to have TELUX_LOC_CONFIG permission to invoke this API successfully.

Parameters

in	*minSVElevation*	- minimum SV elevation to be used by GNSS standard position engine (SPE). Valid range is [0, 90] in unit of degree.
in	*callback*	- Optional callback to get the response of configure minimum SV Elevation angle.

Returns

Status of configureMinSVElevation i.e. success or suitable status code.

4.16.1.3.3.16 virtual telux::common::Status telux::loc::ILocationConfigurator::requestMinSVElevation (GetMinSVElevationCallback *cb*) [pure virtual]

This API retrieves the minimum SV Elevation configuration used by the modem GNSS SPE engine. If this API is invoked right after the configureMinSVElevation, the returned setting may not match the one specified in configureMinSVElevation, as the setting received via configureMinSVElevation might not have been applied yet as it takes time to apply the setting if the GNSS SPE engine has an on-going session. In poor GPS signal condition, the session may take up to 255 seconds to finish. If after 255 seconds of invoking configureMinSVElevation, the returned value still does not match, then the caller need to reapply the setting by invoking configureMinSVElevation again.

Parameters

in	*cb*	- callback to retrieve the minimum SV elevation.

Returns

Status of requestMinSVElevation i.e. success or suitable status code.

4.16.1.3.3.17 virtual telux::common::Status telux::loc::ILocationConfigurator::deleteAidingData (AidingData *aidingDataMask,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

This API deletes specified aiding data from all position engines on the device. For example, removing ephemeris data may trigger GNSS engine to do a warm start. Invoking this API may cause noticeable delay for the position engine to produce first fix and may have other performance impact. So, this API should

only be exercised with caution and only for very limited usage scenario, e.g.: for performance test and certification process.

On platforms with Access control enabled, caller needs to have TELUX_LOC_CONFIG permission to invoke this API successfully.

Parameters

in	*aidingDataMask*	- specify the set of aiding data to be deleted from all position engines. Currently, only ephemeris deletion is supported.
in	*callback*	- Optional callback to get the response of delete aiding data.

Returns

Status of deleteAidingData i.e. success or suitable status code.

4.16.1.3.3.18 virtual telux::common::Status telux::loc::ILocationConfigurator::configureDR (const DREngineConfiguration & *config,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

This API configures various parameters for dead reckoning position engine. Clients should wait for the command to finish e.g.: via ResponseCallback to be received before issuing a second configureDR command. Behavior is not defined if client issues a second request of configureDR without waiting for the completion of the previous configureDR request.

On platforms with Access control enabled, caller needs to have TELUX_LOC_CONFIG permission to invoke this API successfully.

Parameters

in	*config*	- specify dead reckoning engine configuration.
in	*callback*	- Optional callback to get the response of configureDR.

Returns

Status of configureDR i.e. success or suitable status code.

4.16.1.3.3.19 virtual telux::common::Status telux::loc::ILocationConfigurator::configureEngineState (const EngineType *engineType,* const LocationEngineRunState *engineState,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

This API is used to instruct the specified engine to be in the suspended/running state. When the engine is placed in suspended state, the engine will stop. If there is an on-going session, engine will no longer produce fixes. In the suspended state, calling API to delete aiding data from the paused engine may not have effect. Request to delete Aiding data shall be issued after engine resume.

Currently, only DR engine will support this request. The request to suspend/running DR engine can be made with or without an on-going session. With DR engine, on resume, GNSS position & heading re-acquisition may be needed for DR to engage.

On platforms with Access control enabled, caller needs to have TELUX_LOC_CONFIG permission to

invoke this API successfully.

Parameters

in	*engineType*	- the engine that is instructed to change its run state.
in	*engineState*	- the new engine run state that the engine is instructed to be in.
in	*callback*	- Optional callback to get the response of configureEngineState.

Returns

Status of configureEngineState i.e. success or suitable status code.

4.16.1.3.3.20 virtual telux::common::Status telux::loc::ILocationConfigurator::provideConsentForTerrestrialPositioning (bool *userConsent,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

Clients can request Terrestrial Positioning using ILocationManager::getTerrestrialPosition. Terrestrial Positioning requires sending device data to the cloud to get the position. This functionality requires user consent. This API needs to be invoked to provide the user consent.

The consent will remain effective across power cycles, until this API is called with a different value.

On platforms with Access control enabled, caller needs to have TELUX_LOC_CONSENT permission to invoke this API successfully.

Parameters

in	*userConsent*	- true indicates user consents to sending device data to cloud, false indicates user does not consent.
in	*callback*	- Optional callback to get the response of provideConsentForTerrestrialPositioning.

Returns

Status of provideConsentForTerrestrialPositioning i.e. success or suitable status code.

4.16.1.3.3.21 virtual telux::common::Status telux::loc::ILocationConfigurator::configureNmeaTypes (const NmeaSentenceConfig *nmeaType,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

This API is used to configure the NMEA sentence types that clients will receive via ILocationManager::startDetailedReports or ILocationManager::startDetailedEngineReports. Without prior invocation to this API, all NMEA sentences supported in the system will get generated and delivered to all the clients that register to receive NMEA sentences. The NMEA sentence type configuration is common across all clients and updating it will affect all clients. This API call is not incremental and the new NMEA sentence types will completely overwrite the previous call to this API.

On platforms with Access control enabled, caller needs to have TELUX_LOC_CONFIG permission to invoke this API successfully.

Parameters

in	*nmeaType*	- specify the set of NMEA sentences
in	*callback*	- Optional callback to get the response of configureNmeaTypes.

Returns

Status of configureNmeaTypes i.e. success or suitable status code.

4.16.1.3.3.22 virtual telux::common::Status telux::loc::ILocationConfigurator::configureNmea (const NmeaConfig *configParams*, telux::common::ResponseCallback *callback* = `nullptr`) [pure virtual]

This API is used to configure the NMEA sentences that the clients will receive via ILocationManager::startDetailedReports or ILocationManager::startDetailedEngineReports. Without prior invocation to this API, all NMEA sentences supported in the system will get generated and delivered to all the clients that register to receive NMEA sentences. The NMEA sentence type configuration is common across all clients and updating it will affect all clients.

Please note that for the NMEA datum type request to be successful, the nmea provider configuration in the GPS configuration file should be set to application processor.

This API call is not incremental and the new NMEA configuration will completely overwrite the previous call to this API.

Parameters

in	*configParams*	- Configuration Parameters for Nmea on the device.
in	*callback*	- Optional callback to get the response of configureNmea.

Returns

Status of configureNmea i.e. success or suitable status code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.16.1.3.3.23 virtual telux::common::Status telux::loc::ILocationConfigurator::configure↩ EngineIntegrityRisk (const EngineType *engineType*, uint32_t *integrityRisk*, telux::common::ResponseCallback *callback* = `nullptr`) [pure virtual]

This API is used to instruct the specified engine to use the provided integrity risk level for protection level calculation in position report. This API can be called when a position session is in progress. Prior to calling this API for a particular engine, the engine shall not calculate the protection levels and shall not include the protection levels in its position report. The implementation might not support protection levels across all engines. For engines that don't support it, telux::common::ResponseCallback will get invoked with telux::common::ErrorCode::NOT_SUPPORTED.

On platforms with Access control enabled, caller needs to have TELUX_LOC_CONFIG permission to

invoke this API successfully.

Parameters

in	*engineType*	- the engine that is instructed to use the specified integrity risk level for protection level calculation.
in	*integrityRisk*	- the integrity risk level used for calculating protection level. The integrity risk is defined as a probability per epoch, in unit of 2.5e-10. The valid range for actual integrity is [2.5e-10, 1-2.5e-10]), this corresponds to range of [1,4e9-1] of this parameter.
in	*callback*	- Optional callback to get the response of configureEngineIntegrityRisk.

Returns

Status of configureEngineIntegrityRisk i.e. success or suitable status code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.16.1.3.3.24 virtual telux::common::Status telux::loc::ILocationConfigurator::configureXtraParams (bool *enable,* const XtraConfig *configParams,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

This API is used to enable/disable the XTRA (Predicted GNSS Satellite Orbit Data) feature on device. If XTRA feature is to be enabled, this API is also used to configure the various XTRA settings in device.

Clients need to note the below-

1. Wait for the ongoing request to finish prior to the next invocation else the behavior is undefined.
2. The API is non-incremental i.e, the second call will overwrite the first call. Also the configured XTRA params will be persistent.

On platforms with Access control enabled, caller needs to have TELUX_LOC_CONFIG permission to invoke this API successfully.

Parameters

in	*enable*	- Enable XTRA Feature on the device. False would disable both the XTRA Assistance Data and NTP Time Download.
in	*configParams*	- Configuration Parameters for XTRA on the device.
in	*callback*	- Optional callback stating the response errorcode.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.16.1.3.3.25 virtual telux::common::Status telux::loc::ILocationConfigurator::requestXtraStatus (GetXtraStatusCallback *callback*) [pure virtual]

This API is used to query xtra feature setting and xtra assistance data status used by the GNSS standard position engine (SPE). If XTRA_DATA_STATUS_UNKNOWN is returned but XTRA feature is enabled, the client shall wait a few seconds before calling this API again.

Parameters

in	*callback*	- Callback to get the Xtra data status information.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.16.1.3.3.26 virtual telux::common::Status telux::loc::ILocationConfigurator::registerListener (LocConfigIndications *indicationList,* std::weak_ptr< ILocationConfigListener > *listener*) [pure virtual]

This API is used to register a configuration listener for getting specific indications/updates.

Parameters

in	*indicationList*	- List of indications client wants to register under telux::loc::LocConfigIndicationsType.
in	*listener*	- Pointer of ILocationConfigListener object.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.16.1.3.3.27 virtual telux::common::Status telux::loc::ILocationConfigurator::deRegisterListener (LocConfigIndications *indicationList,* std::weak_ptr< ILocationConfigListener > *listener*) [pure virtual]

This API is used to deregister a configuration listener from specific indications/updates.

Parameters

in	*indicationList*	- List of indications client wants to deregister from under telux::loc::LocConfigIndicationsType.
in	*listener*	- Pointer of ILocationConfigListener object.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.16.1.4 struct telux::loc::GnssKinematicsData

Specifies kinematics related information related to device body frame parameters.

Data fields

Type	Field	Description
Kinematic↩ DataValidity	bodyFrame↩ DataMask	Contains Body frame data valid bits.
float	longAccel	Forward Acceleration in body frame (meters/second^2)
float	latAccel	Sideward Acceleration in body frame (meters/second^2)
float	vertAccel	Vertical Acceleration in body frame (meters/second^2)
float	yawRate	Heading Rate (Radians/second)
float	pitch	Body pitch (Radians)
float	longAccelUnc	Uncertainty of Forward Acceleration in body frame (meters/second^2) Uncertainty is defined with 68% confidence level.
float	latAccelUnc	Uncertainty of Side-ward Acceleration in body frame meters/second^2) Uncertainty is defined with 68% confidence level.
float	vertAccelUnc	Uncertainty of Vertical Acceleration in body frame (meters/second^2) Uncertainty is defined with 68% confidence level.
float	yawRateUnc	Uncertainty of Heading Rate (Radians/second) Uncertainty is defined with 68% confidence level.
float	pitchUnc	Uncertainty of Body pitch (Radians) Uncertainty is defined with 68% confidence level.
float	pitchRate	Body pitch rate, in unit of radians/second.
float	pitchRateUnc	Uncertainty of pitch rate, in unit of radians/second. Uncertainty is defined with 68% confidence level.
float	roll	Roll of body frame, clockwise is positive, in unit of radian.
float	rollUnc	Uncertainty of roll, in unit of radian. Uncertainty is defined with 68% confidence level.
float	rollRate	Roll rate of body frame, clockwise is positive, in unit of radian/second.
float	rollRateUnc	Uncertainty of roll rate, in unit of radian/second. Uncertainty is defined with 68% confidence level.
float	yaw	Yaw of body frame, clockwise is positive, in unit of radian.
float	yawUnc	Uncertainty of yaw, in unit of radian. Uncertainty is defined with 68% confidence level.

4.16.1.5 struct telux::loc::LLAInfo

The location info is calculated according to the vehicle's GNSS antenna where as Vehicle Reference Point(VRP) refers to a point on the vehicle where the display of the car sits. The VRP based info is calculated by adding that extra difference between GNSS antenna and the VRP on the top where the location info is recieved. The VRP parameters can be configured through ILocationConfigurator::configureLeverArm. LLAInfo specifies latitude, longitude and altitude info of location for VRP-based.

Data fields

Type	Field	Description
double	latitude	Latitude, in unit of degrees, range [-90.0, 90.0].
double	longitude	Longitude, in unit of degrees, range [-180.0, 180.0].
float	altitude	Altitude above the WGS 84 reference ellipsoid, in unit of meters.

4.16.1.6 struct telux::loc::TimeInfo

Specify non-Glonass Gnss system time info.

Data fields

Type	Field	Description
GnssTime↩ Validity	validityMask	Validity mask for below fields
uint16_t	systemWeek	Extended week number at reference tick. Unit: Week. Set to 65535 if week number is unknown. For GPS: Calculated from midnight, Jan. 6, 1980. OTA decoded 10 bit GPS week is extended to map between: [NV6264 to (NV6264 + 1023)]. For BDS: Calculated from 00:00:00 on January 1, 2006 of Coordinated Universal Time (UTC). For GAL: Calculated from 00:00 UT on Sunday August 22, 1999 (midnight between August 21 and August 22).
uint32_t	systemMsec	Time in to the current week at reference tick. Unit: Millisecond. Range: 0 to 604799999.
float	systemClk↩ TimeBias	System clock time bias Units: Millisecond Note: System time (TOW Millisecond) = systemMsec - systemClkTimeBias.
float	systemClk↩ TimeUncMs	Single sided maximum time bias uncertainty Units: Millisecond
uint32_t	refFCount	FCount (free running HW timer) value. Don't use for relative time purpose due to possible discontinuities. Unit: Millisecond
uint32_t	numClock↩ Resets	Number of clock resets/discontinuities detected, affecting the local hardware counter value.

4.16.1.7 struct telux::loc::GlonassTimeInfo

Specifies Glonass system time info.

Data fields

Type	Field	Description
uint16_t	gloDays	GLONASS day number in four years. Refer to GLONASS ICD. Applicable only for GLONASS and shall be ignored for other constellations. If unknown shall be set to 65535
TimeValidity	validityMask	Validity mask for GlonassTimeInfo fields
uint32_t	gloMsec	GLONASS time of day in Millisecond. Refer to GLONASS ICD. Units: Millisecond.

Type	Field	Description
float	gloClkTime↩ Bias	GLONASS clock time bias. Units: Millisecond Note: GLO time (TOD Millisecond) = gloMsec - gloClkTimeBias. Check for gloClkTimeUncMs before use.
float	gloClkTime↩ UncMs	Single sided maximum time bias uncertainty Units: Millisecond
uint32_t	refFCount	FCount (free running HW timer) value. Don't use for relative time purpose due to possible discontinuities. Unit: Millisecond
uint32_t	numClock↩ Resets	Number of clock resets/discontinuities detected, affecting the local hardware counter value.
uint8_t	gloFourYear	GLONASS four year number from 1996. Refer to GLONASS ICD. Applicable only for GLONASS and shall be ignored for other constellations.

4.16.1.8 union telux::loc::SystemTimeInfo

Union to hold GNSS system time from different constellations in SystemTime.

Data fields

Type	Field	Description
TimeInfo	gps	System time info from GPS constellation.
TimeInfo	gal	System time info from GALILEO constellation.
TimeInfo	bds	System time info from BEIDOU constellation.
TimeInfo	qzss	System time info from QZSS constellation.
GlonassTime↩ Info	glo	System time info from GLONASS constellation.
TimeInfo	navic	System time info from NAVIC constellation.

4.16.1.9 struct telux::loc::SystemTime

GNSS system time in ILocationInfoEx.

Data fields

Type	Field	Description
GnssSystem	gnssSystem↩ TimeSrc	Specify the source constellation for GNSS system time.
SystemTime↩ Info	time	Specify the GNSS system time corresponding to the source.

Type	Field	Description

4.16.1.10 struct telux::loc::GnssMeasurementInfo

Specify the satellite vehicle measurements that are used to calculate location in ILocationInfoEx.

Data fields

Type	Field	Description
GnssSignal	gnssSignalType	GnssSignalType mask
GnssSystem	gnss↩ Constellation	Specifies GNSS Constellation Type
uint16_t	gnssSvId	GNSS SV ID. For GPS: 1 to 32. For GLONASS: [65, 96] or [97, 110]. [65, 96] if orbital slot number(OSN) is known. [97, 110] as frequency channel number(FCN) [-7, 6] plus 104. i.e. encode FCN (-7) as 97, FCN (0) as 104, FCN (6) as 110. For SBAS: 120 to 158 and 183 to 191. For QZSS: 193 to 197. For BDS: 201 to 263. For GAL: 301 to 336. For NAVIC: 401 to 414.

4.16.1.11 struct telux::loc::SvUsedInPosition

Specify the set of SVs that are used to calculate location in ILocationInfoEx.

Data fields

Type	Field	Description
uint64_t	gps	Specify the set of SVs from GPS constellation that are used to compute the position. Bit 0 to Bit 31 corresponds to GPS SV id 1 to 32.
uint64_t	glo	Specify the set of SVs from GLONASS constellation that are used to compute the position. Bit 0 to Bit 31 corresponds to GLO SV id 65 to 96.
uint64_t	gal	Specify the set of SVs from GALILEO constellation that are used to compute the position. Bit 0 to Bit 35 corresponds to GAL SV id 301 to 336.
uint64_t	bds	Specify the set of SVs from BEIDOU constellation that are used to compute the position. Bit 0 to Bit 62 corresponds to BDS SV id 201 to 263.
uint64_t	qzss	Specify the set of SVs from QZSS constellation that are used to compute the position. Bit 0 to Bit 4 corresponds to QZSS SV id 193 to 197.
uint64_t	navic	Specify the set of SVs from NAVIC constellation that are used to compute the position. Bit 0 to Bit 13 corresponds to NAVIC SV id 401 to 414.

4.16.1.12 struct telux::loc::GnssData

Specify the additional GNSS data that can be provided during a tracking session, currently jammer and automatic gain control data are available.

Data fields

Type	Field	Description
GnssData↩ Validity	gnssData↩ Mask[Gnss↩ DataSignal↩ Types::GNSS↩ _DATA_MA↩ X_NUMBER↩ _OF_SIGNA↩ L_TYPES]	bitwise OR of GnssDataValidityType
double	jammer↩ Ind[GnssData↩ SignalTypes::↩ GNSS_DAT↩ A_MAX_NU↩ MBER_OF_↩ SIGNAL_TY↩ PES]	Jammer Indication for each signal type. Each index represents the signal type in GnssDataSignalTypes.
double	agc[Gnss↩ DataSignal↩ Types::GNSS↩ _DATA_MA↩ X_NUMBER↩ _OF_SIGNA↩ L_TYPES]	Automatic gain control for each signal type. Each index corresponds to the signal type in GnssDataSignalTypes.

4.16.1.13 struct telux::loc::SvBlackListInfo

Specify parameters related to enable/disable SVs

Data fields

Type	Field	Description
Gnss↩ Constellation↩ Type	constellation	constellation for the sv
uint32_t	svId	sv id for the constellation: 0 means blacklist for all SVIds of a given constellation type GLONASS SV id range: 65 to 96 QZSS SV id range: 193 to 197 BDS SV id range: 201 to 237 GAL SV id range: 301 to 336 SBAS SV id range: 120 to 158 and 183 to 191

4.16.1.14 struct telux::loc::LeverArmParams

Specify parameters related to lever arm

Data fields

Type	Field	Description
float	forwardOffset	Offset along the vehicle forward axis, in unit of meters

Type	Field	Description
float	sidewaysOffset	Offset along the vehicle starboard axis, in unit of meters
float	upOffset	Offset along the vehicle up axis, in unit of meters

4.16.1.15 struct telux::loc::GnssMeasurementsData

Specify the signal measurement information such as satellite vehicle pseudo range, satellite vehicle time, carrier phase measurement etc. from GNSS positioning engine.

Data fields

Type	Field	Description
Gnss↩ Measurements↩ DataValidity	valid	Bitwise OR of GnssMeasurementsDataValidityType to specify the valid fields in GnssMeasurementsData.
int16_t	svId	Specify satellite vehicle ID number.
Gnss↩ Constellation↩ Type	svType	SV constellation type.
double	timeOffsetNs	Time offset when the measurement was taken, in unit of nanoseconds.
Gnss↩ Measurements↩ StateValidity	stateMask	Bitwise OR of GnssMeasurementsStateValidityType to specify the GNSS measurement state.
int64_t	receivedSv↩ TimeNs	Received GNSS time of the week in nanoseconds when the measurement was taken. Total time is: receivedSvTimeNs+receivedSvTimeSubNs.
float	receivedSv↩ TimeSubNs	Sub nanoseconds portion of the received GNSS time of the week when the measurement was taken. Total time is: receivedSvTimeNs+receivedSvTimeSubNs.
int64_t	receivedSv↩ Time↩ UncertaintyNs	Satellite time. All SV times in the current measurement block are already propagated to a common reference time epoch, in unit of nano seconds.
double	carrierTo↩ NoiseDbHz	Signal strength, carrier to noise ratio, in unit of dB-Hz.
double	pseudorange↩ RateMps	Uncorrected pseudorange rate, in unit of metres/second.
double	pseudorange↩ Rate↩ Uncertainty↩ Mps	Uncorrected pseudorange rate uncertainty, in unit of meters/second.
Gnss↩ Measurements↩ AdrState↩ Validity	adrStateMask	Bitwise OR of GnssMeasurementsAdrStateValidityType.
double	adrMeters	Accumulated delta range, in unit of meters.
double	adr↩ Uncertainty↩ Meters	Accumulated delta range uncertainty, in unit of meters.

Type	Field	Description
float	carrier↩ FrequencyHz	Carrier frequency of the tracked signal, in unit of Hertz.
int64_t	carrierCycles	The number of full carrier cycles between the receiver and the satellite.
double	carrierPhase	The RF carrier phase that the receiver has detected.
double	carrierPhase↩ Uncertainty	The RF carrier phase uncertainty.
Gnss↩ Measurements↩ Multipath↩ Indicator	multipath↩ Indicator	Multipath indicator, could be unknown, present or not present.
double	signalTo↩ NoiseRatioDb	Signal to noise ratio, in unit of dB.
double	agcLevelDb	Automatic gain control level, in unit of dB.
GnssSignal	gnssSignalType	GnssSignalType mask
double	baseband↩ CarrierToNoise	Carrier-to-noise ratio of the signal measured at baseband, in unit of dB-Hz.
double	fullInter↩ SignalBias	The full inter-signal bias (ISB) in nanoseconds. This value is the sum of the estimated receiver-side and the space-segment-side inter-system bias, inter-frequency bias and inter-code bias.
double	fullInter↩ SignalBias↩ Uncertainty	Uncertainty associated with the full inter-signal bias in nanoseconds.

4.16.1.16 struct telux::loc::GnssMeasurementsClock

Specify GNSS measurements clock. The main equation describing the relationship between various components is: utcTimeNs = timeNs - (fullBiasNs + biasNs) - leapSecond * 1,000,000,000

Data fields

Type	Field	Description
Gnss↩ Measurements↩ ClockValidity	valid	Bitwise OR of GnssMeasurementsClockValidityType.
int16_t	leapSecond	Leap second, in unit of seconds.
int64_t	timeNs	Time, monotonically increasing as long as the power is on, in unit of nanoseconds.
double	time↩ UncertaintyNs	Time uncertainty (one sigma), in unit of nanoseconds.
int64_t	fullBiasNs	Full bias, in uint of nanoseconds.
double	biasNs	Sub-nanoseconds bias, in unit of nonoseconds.
double	bias↩ UncertaintyNs	Bias uncertainty (one sigma), in unit of nanoseconds.
double	driftNsps	Clock drift, in unit of nanoseconds/second.
double	drift↩ Uncertainty↩ Nsps	Clock drift uncertainty (one sigma), in unit of nanoseconds/second.

Type	Field	Description
uint32_t	hwClock↩ Discontinuity↩ Count	HW clock discontinuity count - incremented for each discontinuity in HW clock.

4.16.1.17 struct telux::loc::GnssMeasurements

Specify GNSS measurements clock and data. GnssMeasurementInfo is used to convey the satellite vehicle info whose measurements are actually used to generate the current position report. While GnssMeasurements contains the satellite measurements that device observed during tracking session, regardless the measurement is used or not used to compute the fix. Furthermore GnssMeasurements contains much richer set of information which can enable other third party engines to utilize the measurements and compute the position by itself.

Data fields

Type	Field	Description
Gnss↩ Measurements↩ Clock	clock	GNSS measurements clock info.
vector< Gnss↩ Measurements↩ Data >	measurements	GNSS measurements data.
bool	isNHz	Indicates the frequency for GNSS measurements generated at NHz or not.

4.16.1.18 struct telux::loc::GnssDisasterCrisisReport

Specify the Disaster-crisis type and data payload received from the GNSS engine.

Data fields

Type	Field	Description
GnssReportD↩ CType	dcReportType	Disaster and crisis report types supported by the GNSS Engine.
vector< uint8↩ _t >	dcReportData	The disaster crisis report data, packed into uint8_t. The bits in the payload are packed w.r.t the MSB First ordering.
uint16_t	numValidBits	Number of valid bits that client should use in the payload as per the dcReportData.

4.16.1.19 struct telux::loc::LeapSecondChangeInfo

Specify leap second change event info.

Data fields

Type	Field	Description
TimeInfo	timeInfo	GPS timestamp that corrresponds to the last known leap second change event. The info can be available on two scenario: 1: This leap second change event has been scheduled and yet to happen 2: This leap second change event has already happened and next leap second change event has not yet been scheduled.
uint8_t	leapSeconds↩ BeforeChange	Number of leap seconds prior to the leap second change event that corresponds to the timestamp at timeInfo.
uint8_t	leapSeconds↩ AfterChange	Number of leap seconds after the leap second change event that corresponds to the timestamp at timeInfo.

4.16.1.20 struct telux::loc::LeapSecondInfo

Specify leap second info, including current leap second and leap second change event info if available.

Data fields

Type	Field	Description
LeapSecond↩ InfoValidity	valid	Validity of LeapSecondInfo fields.
uint8_t	current	Current leap seconds, in unit of seconds. This info will only be available only if the leap second change info is not available.
LeapSecond↩ ChangeInfo	info	Leap second change event info. The info can be available on two scenario: 1: this leap second change event has been scheduled and yet to happen 2: this leap second change event has already happened and next leap second change event has not yet been scheduled. If leap second change info is available, to figure out the current leap second info, compare current gps time with LeapSecondChangeInfo::timeInfo to know whether to choose leapSecondBefore or leapSecondAfter as current leap second.

4.16.1.21 struct telux::loc::LocationSystemInfo

Specify location system information.

Data fields

Type	Field	Description
Location↩ SystemInfo↩ Validity	valid	validity of LocationSystemInfo::info
LeapSecond↩ Info	info	Current leap second and leap second info.

4.16.1.22 struct telux::loc::GnssEnergyConsumedInfo

Specify the info regarding energy consumed by GNSS engine.

Data fields

Type	Field	Description
GnssEnergy↩ Consumed↩ InfoValidity	valid	Bitwise OR of GnssEnergyConsumedInfoValidityType to specify the valid fields in GnssEnergyConsumedInfo.
uint64_t	energySince↩ FirstBoot	Energy consumed by the modem GNSS engine since device first ever bootup, in unit of 0.1 milli watt seconds. For an invalid reading, INVALID_ENERGY_CONSUMED is returned.

4.16.1.23 struct telux::loc::NmeaConfig

Specify the Nmea Config Parameters

Data fields

Type	Field	Description
Nmea↩ SentenceConfig	sentenceConfig	Specify the sentences to be configured.
Geodetic↩ DatumType	datumType	Specify the datum type to be configured.

4.16.1.24 struct telux::loc::RobustLocationVersion

Specify the versioning info of robust location module for the GNSS standard position engine (SPE).

Data fields

Type	Field	Description
uint8_t	major	Major version number.
uint16_t	minor	Minor version number.

4.16.1.25 struct telux::loc::RobustLocationConfiguration

Specify the robust location configuration used by the GNSS standard position engine (SPE)

Data fields

Type	Field	Description
Robust↩ LocationConfig	validMask	Validity mask
bool	enabled	Specify whether robust location feature is enabled or not.
bool	enabledFor↩ E911	Specify whether robust location feature is enabled or not when device is on E911 call.

Type	Field	Description
Robust↩ Location↩ Version	version	Specify the version info of robust location module used by the GNSS standard position engine (SPE).

4.16.1.26 struct telux::loc::BodyToSensorMountParams

Specify vehicle body-to-Sensor mount parameters for use by dead reckoning positioning engine.

Data fields

Type	Field	Description
float	rollOffset	The misalignment of the sensor board along the horizontal plane of the vehicle chassis measured looking from the vehicle to forward direction. In unit of degrees. Range: [-180.0, 180.0].
float	yawOffset	The misalignment along the horizontal plane of the vehicle chassis measured looking from the vehicle to the right side. Positive pitch indicates vehicle is inclined such that forward wheels are at higher elevation than rear wheels. In unit of degrees. Range: [-180.0, 180.0].
float	pitchOffset	The angle between the vehicle forward direction and the sensor axis as seen from the top of the vehicle, and measured in counterclockwise direction. In unit of degrees. Range: [-180.0, 180.0].
float	offsetUnc	Single uncertainty number that may be the largest of the uncertainties for roll offset, pitch offset and yaw offset. In unit of degrees. Range: [-180.0, 180.0].

4.16.1.27 struct telux::loc::DREngineConfiguration

Specify the dead reckoning engine configuration parameters.

Data fields

Type	Field	Description
DRConfig↩ Validity	validMask	Specify the valid fields.
BodyTo↩ SensorMount↩ Params	mountParam	Body to sensor mount parameters used by dead reckoning positioning engine.
float	speedFactor	Vehicle Speed Scale Factor configuration input for the dead reckoning positioning engine. The multiplicative scale factor is applied to the received Vehicle Speed value (in meter/second) to obtain the true Vehicle Speed. Range is [0.9 to 1.1]. Note: The scale factor is specific to a given vehicle make & model.
float	speedFactor↩ Unc	Vehicle Speed Scale Factor Uncertainty (68% confidence) configuration input for the dead reckoning positioning engine. Range is [0.0 to 0.1]. Note: The scale factor uncertainty is specific to a given vehicle make & model.

Type	Field	Description
float	gyroFactor	Gyroscope Scale Factor configuration input for the dead reckoning positioning engine. The multiplicative scale factor is applied to received gyroscope value to obtain the true value. Range is [0.9 to 1.1]. Note: The scale factor is specific to the Gyroscope sensor and typically derived from either sensor data-sheet or from actual calibration.
float	gyroFactorUnc	Gyroscope Scale Factor uncertainty (68% confidence) configuration input for the dead reckoning positioning engine. Range is [0.0 to 0.1]. Note: The scale factor uncertainty is specific to the Gyroscope sensor and typically derived from either sensor data-sheet or from actual calibration.

4.16.1.28 struct telux::loc::XtraConfig

Xtra feature configuration parameters

Data fields

Type	Field	Description
uint32_t	download↩ IntervalMinute	Number of minutes between periodic, consecutive successful XTRA assistance data downloads. If 0 is specified, modem default download for XTRA assistance data will be performed.
uint32_t	download↩ TimeoutSec	Connection timeout when connecting backend for both xtra assistance data download and NTP time download. If 0 is specified, the download timeout value will use device default values.
uint32_t	download↩ RetryInterval↩ Minute	Interval to wait before retrying for xtra assistance data's download in case of failure. If 0 is specified, XTRA download retry will follow device default behavior and downloadRetryAttempts will also use device default value.
uint32_t	download↩ RetryAttempts	Total number of allowed retry attempts for assistance data's download in case of failure. If 0 is specified, XTRA download retry will follow device default behavior and downloadRetryIntervalMinute will also use device default value.
string	caPath	Path to the certificate authority (CA) repository that needs to be used for XTRA assistance data download. If empty string is specified, device default CA repositaory will be used.

Type	Field	Description
vector< string >	serverURLs	URLs from which XTRA assistance data will be fetched. At least one and up to three URLs need to be configured when this API is used. The URLs, if provided, shall include the port number to be used for download. Valid xtra server URLs should start with "https://". Example of a valid URL : https://path.exampleserver.net:443
vector< string >	ntpServerURLs	URLs for NTP server to fetch current time. If no NTP server URL is provided, then device will use the default NTP server. The URLs, if provided, shall include the port number to be used for download. Example of a valid ntp server URL is: ntp.exampleserver.com:123.
DebugLog↩ Level	daemon↩ DebugLog↩ Level	Level of debug log messages that will be logged.

4.16.1.29 struct telux::loc::XtraStatus

Specify Xtra assistant data's current status, validity and whether it is enabled.

Data fields

Type	Field	Description
bool	featureEnabled	XTRA assistance data and NTP time download is enabled or disabled.
XtraDataStatus	xtraDataStatus	XTRA assistance data status. If XTRA assistance data download is not enabled, this field will be set to XTRA_DATA_STATUS_UNKNOWN.
uint32_t	xtraValidFor↩ Hours	Number of hours that xtra assistance data will remain valid. This field will be valid when xtraDataStatus is set to XTRA_DATA_STATUS_VALID. For all other XtraDataStatus, this field will be set to 0.

4.16.1.30 class telux::loc::ILocationInfoBase

ILocationInfoBase provides interface to get basic position related information like latitude, longitude, altitude, timestamp.

Public member functions

- virtual LocationInfoValidity getLocationInfoValidity ()=0
- virtual LocationTechnology getTechMask ()=0
- virtual float getSpeed ()=0

- virtual double getLatitude ()=0
- virtual double getLongitude ()=0
- virtual double getAltitude ()=0
- virtual float getHeading ()=0
- virtual float getHorizontalUncertainty ()=0
- virtual float getVerticalUncertainty ()=0
- virtual uint64_t getTimeStamp ()=0
- virtual float getSpeedUncertainty ()=0
- virtual float getHeadingUncertainty ()=0
- virtual uint64_t getElapsedRealTime ()=0
- virtual uint64_t getElapsedRealTimeUncertainty ()=0

4.16.1.30.1 Member Function Documentation

4.16.1.30.1.1 virtual LocationInfoValidity telux::loc::ILocationInfoBase::getLocationInfoValidity ( ) [pure virtual]

Retrieves the validity of the Location basic Info.

Returns

Location basic validity mask.

4.16.1.30.1.2 virtual LocationTechnology telux::loc::ILocationInfoBase::getTechMask ( ) [pure virtual]

Retrieves technology used in computing this fix.

Returns

Location technology mask.

4.16.1.30.1.3 virtual float telux::loc::ILocationInfoBase::getSpeed ( ) [pure virtual]

Retrieves Speed.

Returns

speed in meters per second.

4.16.1.30.1.4 virtual double telux::loc::ILocationInfoBase::getLatitude ( ) [pure virtual]

Retrieves latitude. Positive and negative values indicate northern and southern latitude respectively

- Units: Degrees
- Range: -90.0 to 90.0

Returns

Latitude if available else returns NaN.

4.16.1.30.1.5 virtual double telux::loc::ILocationInfoBase::getLongitude ( ) [pure virtual]

Retrieves longitude. Positive and negative values indicate eastern and western longitude respectively

- Units: Degrees
- Range: -180.0 to 180.0

Returns

Longitude if available else returns NaN.

4.16.1.30.1.6 virtual double telux::loc::ILocationInfoBase::getAltitude ( ) [pure virtual]

Retrieves altitude above the WGS 84 reference ellipsoid.

- Units: Meters

Returns

Altitude if available else returns NaN.

4.16.1.30.1.7 virtual float telux::loc::ILocationInfoBase::getHeading ( ) [pure virtual]

Retrieves heading/bearing.

- Units: Degrees
- Range: 0 to 359.999

Returns

Heading if available else returns NaN.

4.16.1.30.1.8 virtual float telux::loc::ILocationInfoBase::getHorizontalUncertainty ( ) [pure virtual]

Retrieves the horizontal uncertainty.

- Units: Meters Uncertainty is defined with 68% confidence level.

Returns

Horizontal uncertainty.

4.16.1.30.1.9 virtual float telux::loc::ILocationInfoBase::getVerticalUncertainty ( ) [pure virtual]

Retrieves the vertical uncertainty.

- Units: Meters Uncertainty is defined with 68% confidence level.

Returns

Vertical uncertainty if available else returns NaN.

4.16.1.30.1.10 virtual uint64_t telux::loc::ILocationInfoBase::getTimeStamp ( ) [pure virtual]

Retrieves UTC timeInfo for the location fix.

- Units: Milliseconds since Jan 1, 1970

Returns

TimeStamp in milliseconds if available else returns UNKNOWN_TIMESTAMP which is zero(as UTC timeStamp has elapsed since January 1, 1970, it cannot be 0)

4.16.1.30.1.11 virtual float telux::loc::ILocationInfoBase::getSpeedUncertainty ( ) [pure virtual]

Retrieves 3-D speed uncertainty/accuracy.

- Units: Meters per Second Uncertainty is defined with 68% confidence level.

Returns

Speed uncertainty if available else returns NaN.

4.16.1.30.1.12 virtual float telux::loc::ILocationInfoBase::getHeadingUncertainty ( ) [pure virtual]

Retrieves heading uncertainty.

- Units: Degrees
- Range: 0 to 359.999 Uncertainty is defined with 68% confidence level.

Returns

Heading uncertainty if available else returns NaN.

4.16.1.30.1.13 virtual uint64_t telux::loc::ILocationInfoBase::getElapsedRealTime ( ) [pure virtual]

Boot timestamp corresponding to the UTC timestamp for Location fix.

- Units: Nano-second

Returns

elapsed real time.

4.16.1.30.1.14 virtual uint64_t telux::loc::ILocationInfoBase::getElapsedRealTimeUncertainty ( ) [pure virtual]

Retrieves elapsed real time uncertainty.

- Units: Nano-second

Returns

elapsed real time uncertainty.

4.16.1.31 class telux::loc::ILocationInfoEx

ILocationInfoEx provides interface to get richer position related information like latitude, longitude, altitude and other information like time stamp, session status, dop, reliabilities, uncertainities etc.

Public member functions

- virtual LocationInfoExValidity getLocationInfoExValidity ()=0
- virtual float getAltitudeMeanSeaLevel ()=0
- virtual float getPositionDop ()=0
- virtual float getHorizontalDop ()=0
- virtual float getVerticalDop ()=0
- virtual float getGeometricDop ()=0
- virtual float getTimeDop ()=0
- virtual float getMagneticDeviation ()=0
- virtual LocationReliability getHorizontalReliability ()=0
- virtual LocationReliability getVerticalReliability ()=0
- virtual float getHorizontalUncertaintySemiMajor ()=0
- virtual float getHorizontalUncertaintySemiMinor ()=0
- virtual float getHorizontalUncertaintyAzimuth ()=0
- virtual float getEastStandardDeviation ()=0

- virtual float getNorthStandardDeviation ()=0
- virtual uint16_t getNumSvUsed ()=0
- virtual SvUsedInPosition getSvUsedInPosition ()=0
- virtual void getSVIds (std::vector< uint16_t > &idsOfUsedSVs)=0
- virtual SbasCorrection getSbasCorrection ()=0
- virtual GnssPositionTech getPositionTechnology ()=0
- virtual GnssKinematicsData getBodyFrameData ()=0
- virtual std::vector< GnssMeasurementInfo > getmeasUsageInfo ()=0
- virtual SystemTime getGnssSystemTime ()=0
- virtual float getTimeUncMs ()=0
- virtual telux::common::Status getLeapSeconds (uint8_t &leapSeconds)=0
- virtual telux::common::Status getVelocityEastNorthUp (std::vector< float > &velocityEastNorthUp)=0
- virtual telux::common::Status getVelocityUncertaintyEastNorthUp (std::vector< float > &velocityUncertaintyEastNorthUp)=0
- virtual uint8_t getCalibrationConfidencePercent ()=0
- virtual DrCalibrationStatus getCalibrationStatus ()=0
- virtual LocationAggregationType getLocOutputEngType ()=0
- virtual PositioningEngine getLocOutputEngMask ()=0
- virtual float getConformityIndex ()=0
- virtual LLAInfo getVRPBasedLLA ()=0
- virtual std::vector< float > getVRPBasedENUVelocity ()=0
- virtual AltitudeType getAltitudeType ()=0
- virtual ReportStatus getReportStatus ()=0
- virtual uint32_t getIntegrityRiskUsed ()=0
- virtual float getProtectionLevelAlongTrack ()=0
- virtual float getProtectionLevelCrossTrack ()=0
- virtual float getProtectionLevelVertical ()=0

4.16.1.31.1 Member Function Documentation

4.16.1.31.1.1 virtual LocationInfoExValidity telux::loc::ILocationInfoEx::getLocationInfoExValidity ( ) [pure virtual]

Retrives the validity of the location info ex. It provides the validity of various information like dop, reliabilities, uncertainities etc.

Returns

Location ex validity mask

4.16.1.31.1.2 virtual float telux::loc::ILocationInfoEx::getAltitudeMeanSeaLevel ( ) [pure virtual]

Retrieves the altitude with respect to mean sea level.

- Units: Meters

Returns

Altitude with respect to mean sea level if available else returns NaN.

4.16.1.31.1.3 virtual float telux::loc::ILocationInfoEx::getPositionDop ( ) [pure virtual]

Retrieves position dilution of precision.

Returns

Position dilution of precision if available else returns NaN. Range: 1 (highest accuracy) to 50 (lowest accuracy)

4.16.1.31.1.4 virtual float telux::loc::ILocationInfoEx::getHorizontalDop ( ) [pure virtual]

Retrieves horizontal dilution of precision.

Returns

Horizontal dilution of precision if available else returns NaN. Range: 1 (highest accuracy) to 50 (lowest accuracy)

4.16.1.31.1.5 virtual float telux::loc::ILocationInfoEx::getVerticalDop ( ) [pure virtual]

Retrieves vertical dilution of precision.

Returns

Vertical dilution of precision if available else returns NaN Range: 1 (highest accuracy) to 50 (lowest accuracy)

4.16.1.31.1.6 virtual float telux::loc::ILocationInfoEx::getGeometricDop ( ) [pure virtual]

Retrieves geometric dilution of precision.

Returns

geometric dilution of precision.

4.16.1.31.1.7 virtual float telux::loc::ILocationInfoEx::getTimeDop ( ) [pure virtual]

Retrieves time dilution of precision.

Returns

Time dilution of precision.

4.16.1.31.1.8 virtual float telux::loc::ILocationInfoEx::getMagneticDeviation ( ) [pure virtual]

Retrieves the difference between the bearing to true north and the bearing shown on magnetic compass. The deviation is positive when the magnetic north is east of true north.

- Units: Degrees

Returns

Magnetic Deviation if available else returns NaN

4.16.1.31.1.9 virtual LocationReliability telux::loc::ILocationInfoEx::getHorizontalReliability ( ) [pure virtual]

Specifies the reliability of the horizontal position.

Returns

LocationReliability of the horizontal position if available else returns UNKNOWN.

4.16.1.31.1.10 virtual LocationReliability telux::loc::ILocationInfoEx::getVerticalReliability ( ) [pure virtual]

Specifies the reliability of the vertical position.

Returns

LocationReliability of the vertical position if available else returns UNKNOWN.

4.16.1.31.1.11 virtual float telux::loc::ILocationInfoEx::getHorizontalUncertaintySemiMajor ( ) [pure virtual]

Retrieves semi-major axis of horizontal elliptical uncertainty.

- Units: Meters Uncertainty is defined with 39% confidence level.

Returns

Semi-major horizontal elliptical uncertainty if available else returns NaN.

4.16.1.31.1.12 virtual float telux::loc::ILocationInfoEx::getHorizontalUncertaintySemiMinor ( ) [pure virtual]

Retrieves semi-minor axis of horizontal elliptical uncertainty.

- Units: Meters Uncertainty is defined with 39% confidence level.

Returns

Semi-minor horizontal elliptical uncertainty if available else returns NaN.

4.16.1.31.1.13 virtual float telux::loc::ILocationInfoEx::getHorizontalUncertaintyAzimuth ( ) [pure virtual]

Retrieves elliptical horizontal uncertainty azimuth of orientation.

- Units: Decimal degrees
- Range: 0 to 180 Confidence for uncertainty is not specified.

Returns

Elliptical horizontal uncertainty azimuth of orientation if available else returns NaN.

4.16.1.31.1.14 virtual float telux::loc::ILocationInfoEx::getEastStandardDeviation ( ) [pure virtual]

Retrieves east standard deviation.

- Units: Meters Uncertainty is defined with 68% confidence level.

Returns

East Standard Deviation.

4.16.1.31.1.15 virtual float telux::loc::ILocationInfoEx::getNorthStandardDeviation ( ) [pure virtual]

Retrieves north standard deviation.

- Units: Meters Uncertainty is defined with 68% confidence level.

Returns

North Standard Deviation.

4.16.1.31.1.16 virtual uint16_t telux::loc::ILocationInfoEx::getNumSvUsed ( ) [pure virtual]

Retrieves number of satellite vehicles used in position report.

Returns

number of Sv used.

4.16.1.31.1.17 virtual SvUsedInPosition telux::loc::ILocationInfoEx::getSvUsedInPosition ( ) [pure virtual]

Retrives the set of satellite vehicles that are used to calculate position.

Returns

set of satellite vehicles for different constellations.

4.16.1.31.1.18 virtual void telux::loc::ILocationInfoEx::getSVIds (std::vector< uint16_t > & *idsOfUsedSVs*) [pure virtual]

Retrieves GNSS Satellite Vehicles used in position data.

Parameters

out	*idsOfUsedSVs*	Vector of Satellite Vehicle identifiers.

4.16.1.31.1.19 virtual SbasCorrection telux::loc::ILocationInfoEx::getSbasCorrection ( ) [pure virtual]

Retrieves navigation solution mask used to indicate SBAS corrections.

Returns

- SBAS (Satellite Based Augmentation System) Correction mask used.

4.16.1.31.1.20 virtual GnssPositionTech telux::loc::ILocationInfoEx::getPositionTechnology ( ) [pure virtual]

Retrieves position technology mask used to indicate which technology is used.

Returns

- Position technology used in computing this fix.

4.16.1.31.1.21 virtual GnssKinematicsData telux::loc::ILocationInfoEx::getBodyFrameData ( ) [pure virtual]

Retrieves position related information.

4.16.1.31.1.22 virtual std::vector<GnssMeasurementInfo> telux::loc::ILocationInfoEx::getmeas↩ UsageInfo ( ) [pure virtual]

Retrieves gnss measurement usage info.

4.16.1.31.1.23 virtual SystemTime telux::loc::ILocationInfoEx::getGnssSystemTime ( ) [pure virtual]

Retrieves type of gnss system.

Returns

- Type of Gnss System.

4.16.1.31.1.24 virtual float telux::loc::ILocationInfoEx::getTimeUncMs ( ) [pure virtual]

Retrieves time uncertainity. For PVT report from SPE engine, confidence level is at 99%. For PVT reports from other engines, confidence level is undefined.

Returns

- Time uncertainty in milliseconds.

4.16.1.31.1.25 virtual telux::common::Status telux::loc::ILocationInfoEx::getLeapSeconds (uint8_t & *leapSeconds*) [pure virtual]

Retrieves leap seconds if available.

Parameters

out	*leapSeconds*	- leap seconds • Units: Seconds

Returns

Status of leap seconds.

4.16.1.31.1.26 virtual telux::common::Status telux::loc::ILocationInfoEx::getVelocityEastNorthUp (std::vector< float > & *velocityEastNorthUp*) [pure virtual]

Retrieves east, North, Up velocity if available.

Parameters

out	*velocityEastNorthUp*	- east, North, Up velocity • Units: Meters/second

Returns

Status of availability of east, North, Up velocity.

4.16.1.31.1.27 virtual telux::common::Status telux::loc::ILocationInfoEx::getVelocityUncertainty↩ EastNorthUp (std::vector< float > & *velocityUncertaintyEastNorthUp*) [pure virtual]

Retrieves east, North, Up velocity uncertainty if available. Uncertainty is defined with 68% confidence level.

Parameters

out	*velocity*↩ *UncertaintyEast*↩ *NorthUp*	- east, North, Up velocity uncertainty • Units: Meters/second

Returns

Status of availability of east, North, Up velocity uncertainty.

4.16.1.31.1.28 virtual uint8_t telux::loc::ILocationInfoEx::getCalibrationConfidencePercent ( ) [pure virtual]

Sensor calibration confidence percent, range [0, 100].

Returns

the percentage of calibration taking all the parameters into account.

4.16.1.31.1.29 virtual DrCalibrationStatus telux::loc::ILocationInfoEx::getCalibrationStatus ( ) [pure virtual]

Sensor calibration status.

Returns

mask indicating the calibration status with respect to different parameters.

4.16.1.31.1.30 virtual LocationAggregationType telux::loc::ILocationInfoEx::getLocOutputEngType ( ) [pure virtual]

Location engine type. When the type is set to LOC_ENGINE_SRC_FUSED, the fix is the propagated/aggregated reports from all engines running on the system (e.g.: DR/SPE/PPE) based QTI algorithm. To check which location engine contributes to the fused output, check for locOutputEngMask.

Returns

the type of engine that was used for calculating the position fix.

4.16.1.31.1.31 virtual PositioningEngine telux::loc::ILocationInfoEx::getLocOutputEngMask ( ) [pure virtual]

When loc output eng type is set to fused, this field indicates the set of engines contribute to the fix.

Returns

the combination of position engines used in calculating the position report when the loc output end type is set to fused.

4.16.1.31.1.32 virtual float telux::loc::ILocationInfoEx::getConformityIndex ( ) [pure virtual]

When robust location is enabled, this field will indicate how well the various input data considered for navigation solution conforms to expectations.

Returns

values in the range [0.0, 1.0], with 0.0 for least conforming and 1.0 for most conforming.

4.16.1.31.1.33 virtual LLAInfo telux::loc::ILocationInfoEx::getVRPBasedLLA ( ) [pure virtual]

Vehicle Reference Point(VRP) based latitude, longitude and altitude information.

4.16.1.31.1.34 virtual std::vector<float> telux::loc::ILocationInfoEx::getVRPBasedENUVelocity ( ) [pure virtual]

VRP-based east, north and up velocity information.

Returns

- vector of directional velocities in this order {east velocity, north velocity, up velocity}

4.16.1.31.1.35 virtual AltitudeType telux::loc::ILocationInfoEx::getAltitudeType ( ) [pure virtual]

Determination of altitude is assumed or calculated. ASSUMED means there may not be enough satellites to determine the precise altitude.

Returns

altitude type ASSUMED/CALCULATED or if not avalilable then UNKNOWN.

4.16.1.31.1.36 virtual ReportStatus telux::loc::ILocationInfoEx::getReportStatus ( ) [pure virtual]

Indicates the status of this report in terms of how optimally the report was calculated by the engine.

Returns

Status of the report. Returns ReportStatus::UNKNOWN if status is unavailable.

4.16.1.31.1.37 virtual uint32_t telux::loc::ILocationInfoEx::getIntegrityRiskUsed ( ) [pure virtual]

Integrity risk used for protection level parameters. Unit of 2.5e-10. Valid range is [1 to (4e9-1)]. Values other than valid range means integrity risk is disabled and ILocationInfoEx::getProtectionLevelAlongTrack, ILocationInfoEx::getProtectionLevelCrossTrack and ILocationInfoEx::getProtectionLevelVertical will not be available.

4.16.1.31.1.38 virtual float telux::loc::ILocationInfoEx::getProtectionLevelAlongTrack ( ) [pure virtual]

Along-track protection level at specified integrity risk, in unit of meter.

4.16.1.31.1.39 virtual float telux::loc::ILocationInfoEx::getProtectionLevelCrossTrack ( ) [pure virtual]

Cross-track protection level at specified integrity risk, in unit of meter.

4.16.1.31.1.40 virtual float telux::loc::ILocationInfoEx::getProtectionLevelVertical ( ) [pure virtual]

Vertical component protection level at specified integrity risk, in unit of meter.

4.16.1.32 class telux::loc::ISVInfo

ISVInfo provides interface to retrieve information about Satellite Vehicles, their position and health status.

Public member functions

- virtual GnssConstellationType getConstellation ()=0
- virtual uint16_t getId ()=0
- virtual SVHealthStatus getSVHealthStatus ()=0
- virtual SVStatus getStatus ()=0
- virtual SVInfoAvailability getHasEphemeris ()=0
- virtual SVInfoAvailability getHasAlmanac ()=0
- virtual SVInfoAvailability getHasFix ()=0
- virtual float getElevation ()=0
- virtual float getAzimuth ()=0
- virtual float getSnr ()=0
- virtual float getCarrierFrequency ()=0
- virtual GnssSignal getSignalType ()=0
- virtual uint16_t getGlonassFcn ()=0
- virtual double getBasebandCnr ()=0

4.16.1.32.1 Member Function Documentation

4.16.1.32.1.1 virtual GnssConstellationType telux::loc::ISVInfo::getConstellation ( ) [pure virtual]

Indicates to which constellation this satellite vehicle belongs.

Returns

GnssConstellationType if available else returns UNKNOWN.

4.16.1.32.1.2 virtual uint16_t telux::loc::ISVInfo::getId ( ) [pure virtual]

GNSS satellite vehicle ID. SV id range of each supported constellations mentioned in GnssMeasurementInfo.

Returns

Identifier of the satellite vehicle otherwise 0(as 0 is not an ID for any of the SVs)

4.16.1.32.1.3 virtual SVHealthStatus telux::loc::ISVInfo::getSVHealthStatus ( ) [pure virtual]

Health status of satellite vehicle.

Returns

HealthStatus of Satellite Vehicle if available else returns UNKNOWN.

- SVHealthStatus

Deprecated

This API is not supported.

4.16.1.32.1.4 virtual SVStatus telux::loc::ISVInfo::getStatus ( ) [pure virtual]

Status of satellite vehicle.

Note

This API is work-in-progress and is subject to change.

Returns

Satellite Vehicle Status if available else returns UNKNOWN.

- SVStatus

Deprecated

This API is not supported.

4.16.1.32.1.5 virtual SVInfoAvailability telux::loc::ISVInfo::getHasEphemeris ( ) [pure virtual]

Indicates whether ephemeris information(which allows the receiver to calculate the satellite's position) is available.

Returns

SVInfoAvailability if Ephemeris exists or not else returns UNKNOWN.

4.16.1.32.1.6 virtual SVInfoAvailability telux::loc::ISVInfo::getHasAlmanac ( ) [pure virtual]

Indicates whether almanac information(which allows receivers to know which satellites are available for tracking) is available.

Returns

SVInfoAvailability if almanac exists or not else returns UNKNOWN.

4.16.1.32.1.7 virtual SVInfoAvailability telux::loc::ISVInfo::getHasFix ( ) [pure virtual]

Indicates whether the satellite is used in computing the fix.

Returns

SVInfoAvailability, if satellite used or not else returns UNKNOWN.

4.16.1.32.1.8 virtual float telux::loc::ISVInfo::getElevation ( ) [pure virtual]

Retrieves satellite vehicle elevation angle.

- Units: Degrees
- Range: 0 to 90

Returns

Elevation if available else returns NaN.

4.16.1.32.1.9 virtual float telux::loc::ISVInfo::getAzimuth ( ) [pure virtual]

Retrieves satellite vehicle azimuth angle.

- Units: Degrees
- Range: 0 to 360

Returns

Azimuth if available else returns NaN.

4.16.1.32.1.10 virtual float telux::loc::ISVInfo::getSnr ( ) [pure virtual]

Retrieves signal-to-noise ratio of the signal measured at antenna of the satellite vehicle.

- Units: dB-Hz

Returns

SNR if available else returns 0.0 value.

4.16.1.32.1.11 virtual float telux::loc::ISVInfo::getCarrierFrequency ( ) [pure virtual]

Indicates the carrier frequency of the signal tracked.

Returns

carrier frequency in Hz else returns UNKNOWN_CARRIER_FREQ frequency when not supported.

4.16.1.32.1.12 virtual GnssSignal telux::loc::ISVInfo::getSignalType ( ) [pure virtual]

Indicates the validity for different types of signal for gps, galileo, beidou etc.

Returns

signalType mask else return UNKNOWN_SIGNAL_MASK when not supported.

4.16.1.32.1.13 virtual uint16_t telux::loc::ISVInfo::getGlonassFcn ( ) [pure virtual]

Retrieves GLONASS frequency channel number in the range [1, 14] which is calculated as FCN [-7, 6] + 8.

Returns

GLONASS frequency channel number.

4.16.1.32.1.14 virtual double telux::loc::ISVInfo::getBasebandCnr ( ) [pure virtual]

Carrier-to-noise ratio of the signal measured at baseband.

- Units: dB-Hz

Returns

carrier-to-noise ratio at baseband else returns UNKNOWN_BASEBAND_CARRIER_NOISE ratio when not supported.

4.16.1.33 class telux::loc::IGnssSVInfo

IGnssSVInfo provides interface to retrieve the list of SV info available and whether altitude is assumed or calculated.

Public member functions

- virtual AltitudeType getAltitudeType ()=0
- virtual std::vector< std::shared_ptr< ISVInfo > > getSVInfoList ()=0

4.16.1.33.1 Member Function Documentation

4.16.1.33.1.1 virtual AltitudeType telux::loc::IGnssSVInfo::getAltitudeType ( ) [pure virtual]

Indicates whether altitude is assumed or calculated.

Returns

AltitudeType if available else returns UNKNOWN.

Deprecated

This API is not supported.

4.16.1.33.1.2 virtual std::vector<std::shared_ptr<ISVInfo> > telux::loc::IGnssSVInfo::getSVInfoList ( ) [pure virtual]

Pointer to satellite vehicles information for all GNSS constellations except GPS.

Returns

Vector of pointer of ISVInfo object if available else returns empty vector.

4.16.1.34 class telux::loc::IGnssSignalInfo

IGnssSignalInfo provides interface to retrieve GNSS data information like jammer metrics and automatic gain control for satellite signal type.

Public member functions

- virtual GnssData getGnssData ()=0

4.16.1.34.1 Member Function Documentation

4.16.1.34.1.1 virtual GnssData telux::loc::IGnssSignalInfo::getGnssData ( ) [pure virtual]

Retrieves jammer metric and Automatic Gain Control(AGC) corresponding to signal types.Jammer metric is linearly proportional to the sum of jammer and noise power at the GNSS antenna port.

Returns

List of jammer metric and a list of automatic gain control for signal type.

4.16.1.35 class telux::loc::LocationFactory

LocationFactory allows creation of location manager.

Public member functions

- virtual std::shared_ptr< ILocationManager > getLocationManager (telux::common::InitResponseCb callback=nullptr)=0
- virtual std::shared_ptr< ILocationConfigurator > getLocationConfigurator (telux::common::InitResponseCb callback=nullptr)=0
- virtual std::shared_ptr< IDgnssManager > getDgnssManager (DgnssDataFormat dataFormat=DgnssDataFormat::DATA_FORMAT_RTCM_3, telux::common::InitResponseCb callback=nullptr)=0

Static Public Member Functions

- static LocationFactory & getInstance ()

4.16.1.35.1 Member Function Documentation

4.16.1.35.1.1 static LocationFactory& telux::loc::LocationFactory::getInstance ( ) [static]

Get Location Factory instance.

4.16.1.35.1.2 virtual std::shared_ptr<ILocationManager> telux::loc::LocationFactory::getLocationManager (telux::common::InitResponseCb *callback* = `nullptr`) [pure virtual]

Get instance of Location Manager

Parameters

in	*callback*	Optional callback to get the response of the manager initialization.

Returns

Pointer of ILocationManager object.

4.16.1.35.1.3 virtual std::shared_ptr<ILocationConfigurator> telux::loc::LocationFactory::getLocationConfigurator (telux::common::InitResponseCb *callback* = `nullptr`) [pure virtual]

Get instance of Location Configurator.

Parameters

in	*callback*	Optional callback pointer to get the response of the manager initialisation.

Returns

Pointer of ILocationConfigurator object.

4.16.1.35.1.4 virtual std::shared_ptr<IDgnssManager> telux::loc::LocationFactory::getDgnssManager (DgnssDataFormat *dataFormat* = DgnssDataFormat::DATA_FORMAT_RTCM_3, telux::common::InitResponseCb *callback* = `nullptr`) [pure virtual]

Get instance of Dgnss manager.

Parameters

in	*dataFormat*	DgnssDataFormat RTCM injection data format
in	*callback*	Optional callback pointer to get the response of the manager initialisation.

Returns

Pointer of IDgnssManager object.

4.16.1.36 class telux::loc::ILocationListener

Listener class for getting location updates and satellite vehicle information.

The methods in listener can be invoked from multiple different threads. Client needs to make sure that implementation is thread-safe.

Public member functions

- virtual void onBasicLocationUpdate (const std::shared_ptr< ILocationInfoBase > &locationInfo)
- virtual void onDetailedLocationUpdate (const std::shared_ptr< ILocationInfoEx > &locationInfo)
- virtual void onDetailedEngineLocationUpdate (const std::vector< std::shared_ptr< ILocationInfoEx > > &locationEngineInfo)
- virtual void onGnssSVInfo (const std::shared_ptr< IGnssSVInfo > &gnssSVInfo)
- virtual void onGnssSignalInfo (const std::shared_ptr< IGnssSignalInfo > &info)
- virtual void onGnssNmeaInfo (uint64_t timestamp, const std::string &nmea)
- virtual void onGnssMeasurementsInfo (const telux::loc::GnssMeasurements &measurementInfo)
- virtual void onGnssDisasterCrisisInfo (const telux::loc::GnssDisasterCrisisReport &dcReportInfo)
- virtual void onCapabilitiesInfo (const telux::loc::LocCapability capabilityInfo)
- virtual ~ILocationListener ()

4.16.1.36.1 Constructors and Destructors

4.16.1.36.1.1 virtual telux::loc::ILocationListener::~ILocationListener ( ) [virtual]

Destructor of ILocationListener

4.16.1.36.2 Member Function Documentation

4.16.1.36.2.1 virtual void telux::loc::ILocationListener::onBasicLocationUpdate (const std::shared_ptr< ILocationInfoBase > & *locationInfo*) [virtual]

This function is called when device receives location update. When there are multiple engines running on the system, the received location information is fused report from all engines.

On platforms with Access control enabled, the client needs to have TELUX_LOC_DATA permission for this listener API to be invoked.

Parameters

in	*locationInfo*	- Location information like latitude, longitude, timeInfo other information such as heading, altitude and velocity etc.

4.16.1.36.2.2 virtual void telux::loc::ILocationListener::onDetailedLocationUpdate (const std::shared_ptr< ILocationInfoEx > & *locationInfo*) [virtual]

This function is called when device receives Gnss location update. When there are multiple engines running on the system, the received location information is fused report from all engines.

On platforms with Access control enabled, the client needs to have TELUX_LOC_DATA permission for this listener API to be invoked.

Parameters

in	*locationInfo*	- Contains richer set of location information like latitude, longitude, timeInfo, heading, altitude, velocity and other information such as deviations, elliptical accuracies etc.

4.16.1.36.2.3 virtual void telux::loc::ILocationListener::onDetailedEngineLocationUpdate (const std::vector< std::shared_ptr< ILocationInfoEx > > & *locationEngineInfo*) [virtual]

This function is called when device receives multiple Gnss location update from the different engine types requested, which are SPE/PPE/FUSED. This API will be called ONLY if we use startDetailedEngineReports.

On platforms with Access control enabled, the client needs to have TELUX_LOC_DATA permission for this listener API to be invoked.

Parameters

in	*locationEngineInfo*	- Contains a list of location infos. Each element in the list corresponds to one of SPE/PPE/FUSED.

4.16.1.36.2.4 virtual void telux::loc::ILocationListener::onGnssSVInfo (const std::shared_ptr< IGnssSVInfo > & *gnssSVInfo*) [virtual]

This function is called when device receives GNSS satellite information.

On platforms with Access control enabled, the client needs to have TELUX_LOC_DATA permission for this listener API to be invoked.

Parameters

in	*gnssSVInfo*	- GNSS satellite information

4.16.1.36.2.5 virtual void telux::loc::ILocationListener::onGnssSignalInfo (const std::shared_ptr< IGnssSignalInfo > & *info*) [virtual]

This function is called when device receives GNSS data information like jammer metrics and automatic gain control for satellite signal type.

On platforms with Access control enabled, the client needs to have TELUX_LOC_DATA permission for this listener API to be invoked.

Parameters

in	*info*	- GNSS signal info

4.16.1.36.2.6 virtual void telux::loc::ILocationListener::onGnssNmeaInfo (uint64_t *timestamp*, const std::string & *nmea*) [virtual]

This function is called when device receives GNSS NMEA sentences.

On platforms with Access control enabled, the client needs to have TELUX_LOC_DATA permission for this listener API to be invoked.

Parameters

in	*timestamp*	- Timestamp
in	*nmea*	- Nmea sentence

4.16.1.36.2.7 virtual void telux::loc::ILocationListener::onGnssMeasurementsInfo (const telux::loc::GnssMeasurements & *measurementInfo*) [virtual]

This function is called when device receives signal measurement information such as satellite vehicle pseudo range, satellite vehicle clock time, carrier phase measurement etc. The frequency at which this API is called is determined by what was requested telux::loc::GnssReportType::MEASUREMENT or telux::loc::GnssReportType::HIGH_RATE_MEASUREMENT in ILocationManager::startDetailedReports and ILocationManager::startDetailedEngineReports.

On platforms with Access control enabled, the client needs to have TELUX_LOC_DATA permission for this listener API to be invoked.

Parameters

in	*measurementInfo*	- GNSS measurement information

4.16.1.36.2.8 virtual void telux::loc::ILocationListener::onGnssDisasterCrisisInfo (const telux::loc::GnssDisasterCrisisReport & *dcReportInfo*) [virtual]

This function is called during a disaster/crisis to update the disaster/crisis reports.

Parameters

in	*dcReportInfo*	- GNSS disaster/crisis report information. This includes the report type and data payload received from the GNSS engine.

4.16.1.36.2.9 virtual void telux::loc::ILocationListener::onCapabilitiesInfo (const telux::loc::LocCapability *capabilityInfo*) [virtual]

This function is called when the capabilities of the location stack gets updated.

On platforms with Access control enabled, the client needs to have TELUX_LOC_DATA permission for this listener API to be invoked.

Parameters

in	*capabilityInfo*	- telux::loc::LocCapability, capability information

4.16.1.37 class telux::loc::ILocationSystemInfoListener

Public member functions

- virtual void onLocationSystemInfo (const LocationSystemInfo &locationSystemInfo)
- virtual ~ILocationSystemInfoListener ()

4.16.1.37.1 Constructors and Destructors

4.16.1.37.1.1 virtual telux::loc::ILocationSystemInfoListener::~ILocationSystemInfoListener ( ) [virtual]

Destructor of ILocationSystemInfoListener

4.16.1.37.2 Member Function Documentation

4.16.1.37.2.1 virtual void telux::loc::ILocationSystemInfoListener::onLocationSystemInfo (const LocationSystemInfo & *locationSystemInfo*) [virtual]

This function is called when device receives location related system information such as leap second change.

On platforms with Access control enabled, the client needs to have TELUX_LOC_DATA permission for this listener API to be invoked.

Parameters

in	*locationSystemInfo*	- contains location system information such as current leap seconds change

4.16.1.38 class telux::loc::ILocationConfigListener

ILocationConfigListener interface is used to receive notifications related to configuration events.

The listener method can be invoked from multiple different threads. Client needs to make sure that implementation is thread-safe.

Public member functions

- virtual void onXtraStatusUpdate (const XtraStatus xtraStatus)
- virtual ~ILocationConfigListener ()

4.16.1.38.1 Constructors and Destructors

4.16.1.38.1.1 virtual telux::loc::ILocationConfigListener::~ILocationConfigListener ( ) [virtual]

4.16.1.38.2 Member Function Documentation

4.16.1.38.2.1 virtual void telux::loc::ILocationConfigListener::onXtraStatusUpdate (const XtraStatus *xtraStatus*) [virtual]

The API is invoked when there is any update in the Xtra assistance data.

Parameters

in	*xtraStatus*	- Xtra assistant data's current status, validity and whether it is enabled.

4.16.1.39 class telux::loc::ILocationManager

ILocationManager provides interface to register and remove listeners. It also allows to set and get configuration/ criteria for position reports. The new APIs(registerListenerEx, deRegisterListenerEx, startDetailedReports, startBasicReports) and old/deprecated APIs(registerListener, removeListener, setPositionReportTimeout, setHorizontalAccuracyLevel, setMinIntervalForReports) should not be used interchangebly, either the new APIs should be used or the old APIs should be used.

Public Types

- using GetEnergyConsumedCallback = std::function< void(telux::loc::GnssEnergyConsumedInfo energyConsumed, telux::common::ErrorCode error)>
- using GetYearOfHwCallback = std::function< void(uint16_t yearOfHw, telux::common::ErrorCode error)>
- using GetTerrestrialInfoCallback = std::function< void(const std::shared_ptr< ILocationInfoBase > terrestrialInfo)>

Public member functions

- virtual bool isSubsystemReady ()=0
- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual std::future< bool > onSubsystemReady ()=0
- virtual telux::common::Status registerListenerEx (std::weak_ptr< ILocationListener > listener)=0
- virtual telux::common::Status deRegisterListenerEx (std::weak_ptr< ILocationListener > listener)=0
- virtual telux::common::Status startDetailedReports (uint32_t interval, telux::common::ResponseCallback callback=nullptr, GnssReportTypeMask reportMask=DEFAULT_GNSS_REPORT)=0
- virtual telux::common::Status startDetailedEngineReports (uint32_t interval, LocReqEngine engineType, telux::common::ResponseCallback callback=nullptr, GnssReportTypeMask reportMask=DEFAULT_GNSS_REPORT)=0
- virtual telux::common::Status startBasicReports (uint32_t distanceInMeters, uint32_t intervalInMs, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status registerForSystemInfoUpdates (std::weak_ptr< ILocationSystemInfoListener > listener, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status deRegisterForSystemInfoUpdates (std::weak_ptr< ILocationSystemInfoListener > listener, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestEnergyConsumedInfo (GetEnergyConsumedCallback cb)=0
- virtual telux::common::Status stopReports (telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status getYearOfHw (GetYearOfHwCallback cb)=0
- virtual telux::common::Status getTerrestrialPosition (uint32_t timeoutMsec, TerrestrialTechnology techMask, GetTerrestrialInfoCallback cb, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status cancelTerrestrialPositionRequest (telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::loc::LocCapability getCapabilities ()=0
- virtual ~ILocationManager ()

4.16.1.39.1 Member Typedef Documentation

4.16.1.39.1.1 using telux::loc::ILocationManager::GetEnergyConsumedCallback = std::function<void(telux::loc::GnssEnergyConsumedInfo energyConsumed, telux::common::ErrorCode error)>

This function is called with the response to getEnergyConsumedInfoUpdate API.

Parameters

in	*energyConsumed*	- Information regarding energy consumed by Gnss engine.
in	*error*	- Return code which indicates whether the operation succeeded or not.

4.16.1.39.1.2 using telux::loc::ILocationManager::GetYearOfHwCallback = std::function<void(uint16_t yearOfHw, telux::common::ErrorCode error)>

This function is called with the response to getYearOfHw API.

Parameters

in	*yearOfHw*	- Year of hardware information.
in	*error*	- Return code which indicates whether the operation succeeded or not.

4.16.1.39.1.3 using telux::loc::ILocationManager::GetTerrestrialInfoCallback = std::function<void(const std::shared_ptr<ILocationInfoBase> terrestrialInfo)>

This function is called with the response to getTerrestrialPosition API.

Parameters

in	*terrestrialInfo*	- basic position related information.

4.16.1.39.2 Constructors and Destructors

4.16.1.39.2.1 virtual telux::loc::ILocationManager::~ILocationManager ( ) [virtual]

Destructor of ILocationManager

4.16.1.39.3 Member Function Documentation

4.16.1.39.3.1 virtual bool telux::loc::ILocationManager::isSubsystemReady ( ) [pure virtual]

Checks the status of location subsystems and returns the result.

Returns

True if location subsystem is ready for service otherwise false.

Deprecated

use getServiceStatus()

4.16.1.39.3.2 virtual telux::common::ServiceStatus telux::loc::ILocationManager::getServiceStatus ( ) [pure virtual]

This status indicates whether the object is in a usable state.

Returns

SERVICE_AVAILABLE - If location manager is ready for service. SERVICE_UNAVAILABLE - If location manager is temporarily unavailable. SERVICE_FAILED - If location manager encountered an irrecoverable failure.

4.16.1.39.3.3 virtual std::future<bool> telux::loc::ILocationManager::onSubsystemReady ( ) [pure virtual]

Wait for location subsystem to be ready.

Returns

A future that caller can wait on to be notified when location subsystem is ready.

Deprecated

The callback mechanism introduced in the LocationFactory::getLocationManager() API will provide the similar notification mechanism as onSubsystemReady(). This API will soon be removed from further releases.

4.16.1.39.3.4 virtual telux::common::Status telux::loc::ILocationManager::registerListenerEx (std::weak_ptr< ILocationListener > *listener*) [pure virtual]

Register a listener for specific updates from location manager like location, jamming info and satellite vehicle info. If enhanced position, using Dead Reckoning etc., is enabled, enhanced fixes will be provided. Otherwise raw GNSS fixes will be provided. The position reports will start only when startDetailedReports or startBasicReports is invoked.

Parameters

in	*listener*	- Pointer of ILocationListener object that processes the notification.

Returns

Status of registerListener i.e success or suitable status code.

4.16.1.39.3.5 virtual telux::common::Status telux::loc::ILocationManager::deRegisterListenerEx (std::weak_ptr< ILocationListener > *listener*) [pure virtual]

Remove a previously registered listener.

Parameters

in	*listener*	- Previously registered ILocationListener that needs to be removed.

Returns

Status of removeListener success or suitable status code

4.16.1.39.3.6 virtual telux::common::Status telux::loc::ILocationManager::startDetailedReports (uint32_t *interval,* telux::common::ResponseCallback *callback = nullptr,* GnssReportTypeMask *reportMask =* DEFAULT_GNSS_REPORT) [pure virtual]

Starts the richer location reports by configuring the time between them as the interval. Any of the 3 APIs that is startDetailedReports or startDetailedEngineReports or startBasicReports can be called one after the other irrespective of order, without calling stopReports in between any of them and the API which is called last will be honored for providing the callbacks. In case of multiple clients invoking this API with different intervals, if the platforms is configured, then the clients will receive the reports at their requested intervals. If not configured then all the clients will be serviced at the smallest interval among all clients' intervals. The supported periodicities are 100ms, 200ms, 500ms, 1sec, 2sec, nsec and a periodicity that a caller send which is not one of these will result in the implementation picking one of these periodicities. Calling this Api will result in ILocationListener::onDetailedLocationUpdate, ILocationListener::onGnssSVInfo, ILocationListener::onGnssSignalInfo, ILocationListener::onGnssNmeaInfo and ILocationListener::onGnssMeasurementsInfo APIs on the listener being invoked, assuming they have not been disabled using the GnssReportTypeMask. If a client issues second request to this API then new request for GnssReportTypeMask will over write the previous call to this API.

On platforms with Access control enabled, caller needs to have TELUX_LOC_DATA permission to invoke this API successfully.

Parameters

in	*interval*	- Minimum time interval between two consecutive reports in milliseconds.

E.g. If minInterval is 1000 milliseconds, reports will be provided with a periodicity of 1 second or more

depending on the number of applications listening to location updates.

Parameters

in	*callback*	- Optional callback to get the response of set minimum interval for reports.
in	*reportMask*	- Optional field to specify which reports a client is interested in. By default all the reports will be enabled.

Returns

Status of startDetailedReports i.e. success or suitable status code.

4.16.1.39.3.7 virtual telux::common::Status telux::loc::ILocationManager::startDetailedEngineReports (uint32_t *interval,* LocReqEngine *engineType,* telux::common::ResponseCallback *callback* = `nullptr,` GnssReportTypeMask *reportMask* = DEFAULT_GNSS_REPORT) `[pure virtual]`

Starts a session which may provide richer default combined position reports and position reports from other engines. The fused position report type will always be supported if at least one engine in the system is producing valid report. Any of the 3 APIs that is startDetailedReports or startDetailedEngineReports or startBasicReports can be called one after the other irrespective of order, without calling stopReports in between any of them and the API which is called last will be honored for providing the callbacks. In case of multiple clients invoking this API with different intervals, if the platforms is configured, then the clients will receive the reports at their requested intervals. If not configured then all the clients will be serviced at the smallest interval among all clients' intervals. The supported periodicities are 100ms, 200ms, 500ms, 1sec, 2sec, nsec and a periodicity that a caller send which is not one of these will result in the implementation picking one of these periodicities. Calling this Api will result in ILocationListener::onDetailedEngineLocationUpdate, ILocationListener::onGnssSVInfo, ILocationListener::onGnssSignalInfo, ILocationListener::onGnssNmeaInfo and ILocationListener::onGnssMeasurementsInfo APIs on the listener being invoked, assuming they have not been disabled using the GnssReportTypeMask. If a client issues second request to this API then new request for GnssReportTypeMask will over write the previous call to this API.

On platforms with Access control enabled, caller needs to have TELUX_LOC_DATA permission to invoke this API successfully.

Parameters

in	*interval*	- Minimum time interval between two consecutive reports in milliseconds.

E.g. If minInterval is 1000 milliseconds, reports will be provided with a periodicity of 1 second or more depending on the number of applications listening to location updates.

Parameters

in	*engineType*	- The type of engine requested for fixes such as SPE or PPE or FUSED. The FUSED includes all the engines that are running to generate the fixes such as reports from SPE, PPE and DRE.

in	*callback*	- Optional callback to get the response of set minimum interval for reports.
in	*reportMask*	- Optional field to specify which reports a client is interested in. By default all the reports will be enabled.

Returns

Status of startDetailedEngineReports i.e. success or suitable status code.

4.16.1.39.3.8 virtual telux::common::Status telux::loc::ILocationManager::startBasicReports (uint32_t *distanceInMeters,* uint32_t *intervalInMs,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

Starts the Location report by configuring the time and distance between the consecutive reports. Any of the 3 APIs that is startDetailedReports or startDetailedEngineReports or startBasicReports can be called one after the other irrespective of order, without calling stopReports in between any of them and the API which is called last will be honored for providing the callbacks. In case of multiple clients invoking this API with different intervals, if the platforms is configured, then the clients will receive the reports at their requested intervals. If not configured then all the clients will be serviced at the smallest interval among all clients' intervals. The supported periodicities are 100ms, 200ms, 500ms, 1sec, 2sec, nsec and a periodicity that a caller send which is not one of these will result in the implementation picking one of these periodicities. This Api enables the onBasicLocationUpdate Api on the listener.

On platforms with Access control enabled, caller needs to have TELUX_LOC_DATA permission to invoke this API successfully.

E.g. If intervalInMs is 1000 milliseconds and distanceInMeters is 100m, reports will be provided according to the condition that happens first. So we need to provide both the parameters for evaluating the report.

The underlying system may have a minimum distance threshold(e.g. 1 meter). Effective distance will not be smaller than this lower bound.

The effective distance may have a granularity level higher than 1 m, e.g. 5 m. So distanceInMeters being 59 may be honored at 60 m, depending on the system.

Where there is another application in the system having a session with shorter distance, this client may benefit and receive reports at that distance.

Parameters

in	*distanceInMeters*	- DistanceInMeters between two consecutive reports in meters.
in	*intervalInMs*	- Minimum time interval between two consecutive reports in milliseconds.
in	*callback*	- Optional callback to get the response of set minimum distance for reports.

Returns

Status of startBasicReports i.e. success or suitable status code.

4.16.1.39.3.9 virtual telux::common::Status telux::loc::ILocationManager::registerForSystemInfo↩ Updates (std::weak_ptr< ILocationSystemInfoListener > *listener,* telux::common::↩ ResponseCallback *callback = nullptr*) [pure virtual]

This API registers a ILocationSystemInfoListener listener and will receive information related to location system that are not tied with location fix session, e.g.: next leap second event. The startBasicReports, startDetailedReports, startDetailedEngineReports does not need to be called before calling this API, in order to receive updates.

Parameters

in	*listener*	- Pointer of ILocationSystemInfoListener object.
in	*callback*	- Optional callback to get the response of location system info.

Returns

Status of getLocationSystemInfo i.e success or suitable status code.

4.16.1.39.3.10 virtual telux::common::Status telux::loc::ILocationManager::deRegisterFor↩ SystemInfoUpdates (std::weak_ptr< ILocationSystemInfoListener > *listener,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

This API removes a previously registered listener and will also stop receiving informations related to location system for that particular listener.

Parameters

in	*listener*	- Previously registered ILocationSystemInfoListener that needs to be removed.
in	*callback*	- Optional callback to get the response of location system info.

Returns

Status of deRegisterForSystemInfoUpdates success or suitable status code.

4.16.1.39.3.11 virtual telux::common::Status telux::loc::ILocationManager::requestEnergyConsumed↩ Info (GetEnergyConsumedCallback *cb*) [pure virtual]

This API receives information on energy consumed by modem GNSS engine. If this API is called on this object while this is already a pending request, then it will overwrite the callback to be invoked and the callback from the previous invocation will not be called.

Parameters

in	*cb*	- callback to get the information of Gnss energy consumed.

Returns

Status of requestEnergyConsumedInfo i.e success or suitable status code.

4.16.1.39.3.12 virtual telux::common::Status telux::loc::ILocationManager::stopReports (telux::common::ResponseCallback *callback* = `nullptr`) [pure virtual]

This API will stop reports started using startDetailedReports or startBasicReports or registerListener or setMinIntervalForReports.

On platforms with Access control enabled, caller needs to have TELUX_LOC_DATA permission to invoke this API successfully.

Parameters

in	*callback*	- Optional callback to get the response of stop reports.

Returns

Status of stopReports i.e. success or suitable status code.

4.16.1.39.3.13 virtual telux::common::Status telux::loc::ILocationManager::getYearOfHw (GetYearOfHwCallback *cb*) [pure virtual]

This API retrieves the year of hardware information.

Parameters

in	*cb*	- callback to get information of year of hardware.

Returns

Status of getYearOfHw i.e success or suitable status code.

4.16.1.39.3.14 virtual telux::common::Status telux::loc::ILocationManager::getTerrestrialPosition (uint32_t *timeoutMsec*, TerrestrialTechnology *techMask*, GetTerrestrialInfoCallback *cb*, telux::common::ResponseCallback *callback* = `nullptr`) [pure virtual]

This API retrieves single-shot terrestrial position using the set of specified terrestrial technologies. This API can be invoked even while there is an on-going tracking session that was started using startBasicReports/startDetailedReports/startDetailedEngineReports. If this API is invoked while there is already a pending request for terrestrial position, the request will fail and telux::common::ResponseCallback will get invoked with telux::common::ErrorCode::OP_IN_PROGRESS. To cancel a pending request, use ILocationManager::cancelTerrestrialPositionRequest. Before using this API, user consent needs to be set true via ILocationConfigurator::provideConsentForTerrestrialPositioning.

On platforms with Access control enabled, caller needs to have TELUX_LOC_DATA permission to invoke this API successfully.

Parameters

in	*timeoutMsec*	- the time in milliseconds within which the client is expecting a response. If the system is unable to provide a report within this time, the telux::common::ResponseCallback will be invoked with telux::common::ErrorCode::OPERATION_TIMEOUT.
in	*techMask*	- the set of terrestrial technologies that are allowed to be used for producing the position.
in	*cb*	- callback to receive terrestrial position. This callback will only be invoked when ResponseCallback is invoked with SUCCESS.
in	*callback*	- Optional callback to get the response of getTerrestrialPosition.

Returns

Status of getTerrestrialPosition i.e success or suitable status code.

4.16.1.39.3.15 virtual telux::common::Status telux::loc::ILocationManager::cancelTerrestrial↩ PositionRequest (telux::common::ResponseCallback *callback* = `nullptr`) [pure virtual]

This API cancels the pending request invoked by ILocationManager::getTerrestrialPosition. If this API is invoked while there is no pending request for terrestrial position from ILocationManager::getTerrestrialPosition, then telux::common::ResponseCallback will be invoked with telux::common::ErrorCode::INVALID_ARGUMENTS.

On platforms with Access control enabled, caller needs to have TELUX_LOC_DATA permission to invoke this API successfully.

Parameters

in	*callback*	- Optional callback to get the response of cancelTerrestrialPositionRequest.

Returns

Status of cancelTerrestrialPositionRequest i.e success or suitable status code.

4.16.1.39.3.16 virtual telux::loc::LocCapability telux::loc::ILocationManager::getCapabilities ( ) [pure virtual]

This API retrieves capability information.

Returns

Status of getCapabilities i.e success or suitable status code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.16.2 Enumeration Type Documentation

4.16.2.1 enum telux::loc::DgnssDataFormat [strong]

Defines RTCM injection data format

Enumerator

DATA_FORMAT_UNKNOWN Source data format is unknown
DATA_FORMAT_RTCM_3 Source data format is RTCM_3
DATA_FORMAT_3GPP_RTK_R15 Source data format is 3GPP RTK Rel-15

4.16.2.2 enum telux::loc::DgnssStatus [strong]

Defines status reported by cdfw for RTCM injection.

Enumerator

DATA_SOURCE_NOT_SUPPORTED Dgnss subsystem doesn't support the data source
DATA_FORMAT_NOT_SUPPORTED Dgnss subsystem doesn't support the data format
OTHER_SOURCE_IN_USE After the source injects the data, dgnss subsystem discovers there is another higher priority source injecting the data at the same time, and the current injected data is dropped
MESSAGE_PARSE_ERROR There is a parsing error such as unrecognized format, CRC check failure, value range check failure, etc.; the injected data is dropped
DATA_SOURCE_NOT_USABLE Data source is not usable anymore

4.16.2.3 enum telux::loc::HorizontalAccuracyLevel [strong]

Defines the horizontal accuracy level of the fix.

Enumerator

LOW Client requires low horizontal accuracy
MEDIUM Client requires medium horizontal accuracy
HIGH Client requires high horizontal accuracy

4.16.2.4 enum telux::loc::LocationReliability [strong]

Specifies the reliability of the position.

Enumerator

UNKNOWN Unknown location reliability
NOT_SET Location reliability is not set. The reliability of this position report could not be determined. It could be unreliable/reliable
VERY_LOW Location reliability is very low
LOW Location reliability is low, little or no cross-checking is possible
MEDIUM Location reliability is medium, limited cross-check passed
HIGH Location reliability is high, strong cross-check passed

4.16.2.5 enum telux::loc::SbasCorrectionType

Specify set of navigation solutions that contribute to Gnss Location. Defines Satellite Based Augmentation System(SBAS) corrections. SBAS contributes to improve the performance of GNSS system.

Enumerator

SBAS_CORRECTION_IONO Bit mask to specify whether SBAS ionospheric correction is used
SBAS_CORRECTION_FAST Bit mask to specify whether SBAS fast correction is used
SBAS_CORRECTION_LONG Bit mask to specify whether SBAS long correction is used
SBAS_INTEGRITY Bit mask to specify whether SBAS integrity information is used
SBAS_CORRECTION_DGNSS Bit mask to specify whether SBAS DGNSS correction is used
SBAS_CORRECTION_RTK Bit mask to specify whether SBAS RTK correction is used
SBAS_CORRECTION_PPP Bit mask to specify whether SBAS PPP correction is used
SBAS_CORRECTION_RTK_FIXED Bit mask to specify whether SBAS RTK fixed correction is used
SBAS_CORRECTION_ONLY_SBAS_CORRECTED_SV_USED_ Bit mask to specify only SBAS corrected SV is used
SBAS_COUNT Bitset

4.16.2.6 enum telux::loc::AltitudeType `[strong]`

Indicates whether altitude is assumed or calculated.

Enumerator

UNKNOWN Unknown altitude type
CALCULATED Altitude is calculated
ASSUMED Altitude is assumed, there may not be enough satellites to determine the precise altitude

4.16.2.7 enum telux::loc::GnssConstellationType `[strong]`

Defines constellation type of GNSS.

Enumerator

UNKNOWN Unknown constellation type
GPS GPS satellite
GALILEO GALILEO satellite
SBAS SBAS satellite
COMPASS COMPASS satellite.

Deprecated

constellation type is not supported.

GLONASS GLONASS satellite
BDS BDS satellite
QZSS QZSS satellite
NAVIC NAVIC satellite

4.16.2.8 enum telux::loc::SVHealthStatus [strong]

Health status indicates whether satellite is operational or not. This information comes from the most recent data transmitted in satellite almanacs.

Enumerator

UNKNOWN Unknown sv health status
UNHEALTHY satellite is not operational and cannot be used in position calculations
HEALTHY satellite is fully operational

4.16.2.9 enum telux::loc::SVStatus [strong]

Satellite vehicle processing status.

Enumerator

UNKNOWN Unknown sv status
IDLE SV is not being actively processed
SEARCH The system is searching for this SV
TRACK SV is being tracked

4.16.2.10 enum telux::loc::SVInfoAvailability [strong]

Indicates whether Satellite Vehicle info like ephemeris and almanac are present or not

Enumerator

UNKNOWN Unknown sv info availability
YES Ephemeris or Almanac exits
NO Ephemeris or Almanac doesn't exist

4.16.2.11 enum telux::loc::GnssPositionTechType

Specifies which position technology was used to generate location information in the ILocationInfoEx.

Enumerator

GNSS_DEFAULT Technology used to generate location info is unknown.
GNSS_SATELLITE Satellites-based technology was used to generate location info.
GNSS_CELLID Cell towers were used to generate location info.
GNSS_WIFI Wi-Fi access points were used to generate location info.
GNSS_SENSORS Sensors were used to generate location info.
GNSS_REFERENCE_LOCATION Reference location was used to generate location info.
GNSS_INJECTED_COARSE_POSITION Coarse position injected into the location engine was used to generate location info.
GNSS_AFLT AFLT was used to generate location info.
GNSS_HYBRID GNSS and network-provided measurements were used to generate location info.
GNSS_PPE Precise position engine was used to generate location info.
GNSS_VEHICLE Location was calculated using Vehicular data.
GNSS_VISUAL Location was calculated using Visual data.
GNSS_PROPAGATED Location was calculated using Propagation logic, which uses cached

measurements.

4.16.2.12 enum telux::loc::KinematicDataValidityType

Specifies related kinematics mask

Enumerator

HAS_LONG_ACCEL Navigation data has Forward Acceleration
HAS_LAT_ACCEL Navigation data has Sideward Acceleration
HAS_VERT_ACCEL Navigation data has Vertical Acceleration
HAS_YAW_RATE Navigation data has Heading Rate
HAS_PITCH Navigation data has Body pitch
HAS_LONG_ACCEL_UNC Navigation data has Forward Acceleration
HAS_LAT_ACCEL_UNC Navigation data has Sideward Acceleration
HAS_VERT_ACCEL_UNC Navigation data has Vertical Acceleration
HAS_YAW_RATE_UNC Navigation data has Heading Rate
HAS_PITCH_UNC Navigation data has Body pitch
HAS_PITCH_RATE_BIT Navigation data has Body pitch rate
HAS_PITCH_RATE_UNC_BIT Navigation data has Body pitch rate uncertainty
HAS_ROLL_BIT Navigation data has roll
HAS_ROLL_UNC_BIT Navigation data has roll uncertainty
HAS_ROLL_RATE_BIT Navigation data has roll rate
HAS_ROLL_RATE_UNC_BIT Navigation data has roll rate uncertainty
HAS_YAW_BIT Navigation data has yaw
HAS_YAW_UNC_BIT Navigation data has yaw uncertainty

4.16.2.13 enum telux::loc::GnssSystem `[strong]`

Specify the different types of constellation supported.

Enumerator

GNSS_LOC_SV_SYSTEM_UNKNOWN UNKNOWN satellite.
GNSS_LOC_SV_SYSTEM_GPS GPS satellite.
GNSS_LOC_SV_SYSTEM_GALILEO GALILEO satellite.
GNSS_LOC_SV_SYSTEM_SBAS SBAS satellite.
GNSS_LOC_SV_SYSTEM_COMPASS COMPASS satellite.

Deprecated

constellation type is not supported.

GNSS_LOC_SV_SYSTEM_GLONASS GLONASS satellite.
GNSS_LOC_SV_SYSTEM_BDS BDS satellite.
GNSS_LOC_SV_SYSTEM_QZSS QZSS satellite.
GNSS_LOC_SV_SYSTEM_NAVIC NAVIC satellite.

4.16.2.14 enum telux::loc::GnssTimeValidityType

Validity field for different system time in struct TimeInfo.

Enumerator

GNSS_SYSTEM_TIME_WEEK_VALID valid systemWeek.
GNSS_SYSTEM_TIME_WEEK_MS_VALID valid systemMsec
GNSS_SYSTEM_CLK_TIME_BIAS_VALID valid systemClkTimeBias
GNSS_SYSTEM_CLK_TIME_BIAS_UNC_VALID valid systemClkTimeUncMs
GNSS_SYSTEM_REF_FCOUNT_VALID valid refFCount
GNSS_SYSTEM_NUM_CLOCK_RESETS_VALID valid numClockResets

4.16.2.15 enum telux::loc::GlonassTimeValidity

Validity field for GLONASS time in struct GlonassTimeInfo.

Enumerator

GNSS_CLO_DAYS_VALID valid gloDays
GNSS_GLOS_MSEC_VALID valid gloMsec
GNSS_GLO_CLK_TIME_BIAS_VALID valid gloClkTimeBias
GNSS_GLO_CLK_TIME_BIAS_UNC_VALID valid gloClkTimeUncMs
GNSS_GLO_REF_FCOUNT_VALID valid refFCount
GNSS_GLO_NUM_CLOCK_RESETS_VALID valid numClockResets
GNSS_GLO_FOUR_YEAR_VALID valid gloFourYear

4.16.2.16 enum telux::loc::GnssSignalType

Specify GNSS Signal Type and RF Band used in struct GnssMeasurementInfo and ISVInfo class.

Enumerator

GPS_L1CA Gnss signal is of GPS L1CA RF Band.
GPS_L1C Gnss signal is of GPS L1C RF Band.
GPS_L2 Gnss signal is of GPS L2 RF Band.
GPS_L5 Gnss signal is of GPS L5 RF Band.
GLONASS_G1 Gnss signal is of GLONASS G1 (L1OF) RF Band.
GLONASS_G2 Gnss signal is of GLONASS G2 (L2OF) RF Band.
GALILEO_E1 Gnss signal is of GALILEO E1 RF Band.
GALILEO_E5A Gnss signal is of GALILEO E5A RF Band.
GALILIEO_E5B Gnss signal is of GALILEO E5B RF Band.
BEIDOU_B1 Gnss signal is of BEIDOU B1 RF Band.
BEIDOU_B2 Gnss signal is of BEIDOU B2 RF Band.
QZSS_L1CA Gnss signal is of QZSS L1CA RF Band.
QZSS_L1S Gnss signal is of QZSS L1S RF Band.
QZSS_L2 Gnss signal is of QZSS L2 RF Band.
QZSS_L5 Gnss signal is of QZSS L5 RF Band.
SBAS_L1 Gnss signal is of SBAS L1 RF Band.
BEIDOU_B1I Gnss signal is of BEIDOU B1I RF Band.
BEIDOU_B1C Gnss signal is of BEIDOU B1C RF Band.

BEIDOU_B2I Gnss signal is of BEIDOU B2I RF Band.
BEIDOU_B2AI Gnss signal is of BEIDOU B2AI RF Band.
NAVIC_L5 Gnss signal is of NAVIC L5 RF Band.
BEIDOU_B2AQ Gnss signal is of BEIDOU B2A_Q RF Band.

4.16.2.17 enum telux::loc::LocCapabilityType

Specify Location Capabilities Type.

Enumerator

TIME_BASED_TRACKING Support time based tracking session via ILocationManager::startDetailedReports, ILocationManager::startDetailedEngineReports and ILocationManager::startBasicReports with distanceInMeters set to 0.
DISTANCE_BASED_TRACKING Support distance based tracking session via ILocationManager::startBasicReports with distanceInMeters specified.
GNSS_MEASUREMENTS Support Gnss Measurement data via ILocationListener::onGnssMeasurementsInfo when a tracking session is enabled.
CONSTELLATION_ENABLEMENT Support configure constellations via ILocationConfigurator::configureConstellations.
CARRIER_PHASE Support carrier phase for Precise Positioning Measurement Engine (PPME).
QWES_GNSS_SINGLE_FREQUENCY Support GNSS Single Frequency feature.
QWES_GNSS_MULTI_FREQUENCY Supports GNSS Multi Frequency feature.
QWES_VPE Support VEPP license bundle is enabled. VEPP bundle include Carrier Phase features.
QWES_CV2X_LOCATION_BASIC Support for CV2X Location basic features. This includes features for GTS Time & Freq, ILocationConfigurator::configureCTunc.
QWES_CV2X_LOCATION_PREMIUM Support for CV2X Location premium features. This includes features for CV2X Location Basic features, QDR3 feature and ILocationConfigurator::configurePACE.
QWES_PPE Support PPE (Precise Positioning Engine) library is enabled or Precise Positioning Framework (PPF) is available. This includes features for Carrier Phase and SV Ephermeris.
QWES_QDR2 Support QDR2_C license bundle is enabled.
QWES_QDR3 Support QDR3_C license bundle is enabled.

4.16.2.18 enum telux::loc::LocationTechnologyType

Specify the set of technologies that contribute to ILocationInfoBase.

Enumerator

LOC_GNSS Location was calculated using GNSS-based technology.
LOC_CELL Location was calculated using Cell-based technology.
LOC_WIFI Location was calculated using WiFi-based technology.
LOC_SENSORS Location was calculated using Sensors-based technology.
LOC_REFERENCE_LOCATION Location was calculated using Reference location.
LOC_INJECTED_COARSE_POSITION Location was calculated using Coarse position injected into the location engine.
LOC_AFLT Location was calculated using AFLT.
LOC_HYBRID Location was calculated using GNSS and network-provided measurements.
LOC_PPE Location was calculated using Precise position engine.

LOC_VEH Location was calculated using Vehicular data.
LOC_VIS Location was calculated using Visual data.
LOC_PROPAGATED Location was calculated using Propagation logic, which uses cached measurements.

4.16.2.19 enum telux::loc::LocationValidityType

Specify the valid fields in LocationInfoValidity User should determine whether a field in LocationInfoValidity is valid or not by checking the corresponding bit is set or not.

Enumerator

HAS_LAT_LONG_BIT Location has valid latitude and longitude.
HAS_ALTITUDE_BIT Location has valid altitude.
HAS_SPEED_BIT Location has valid speed.
HAS_HEADING_BIT Location has valid heading.
HAS_HORIZONTAL_ACCURACY_BIT
HAS_VERTICAL_ACCURACY_BIT Location has valid vertical accuracy.
HAS_SPEED_ACCURACY_BIT Location has valid speed accuracy.
HAS_HEADING_ACCURACY_BIT Location has valid heading accuracy.
HAS_TIMESTAMP_BIT Location has valid timestamp.
HAS_ELAPSED_REAL_TIME_BIT Location has valid elapsed real time.
HAS_ELAPSED_REAL_TIME_UNC_BIT Location has valid elapsed real time uncertainty.

4.16.2.20 enum telux::loc::LocationInfoExValidityType

Specify the valid fields in LocationInfoExValidityType. User should determine whether a field in LocationInfoExValidityType is valid or not by checking the corresponding bit is set or not.

Enumerator

HAS_ALTITUDE_MEAN_SEA_LEVEL valid altitude mean sea level
HAS_DOP valid pdop, hdop, and vdop
HAS_MAGNETIC_DEVIATION valid magnetic deviation
HAS_HOR_RELIABILITY valid horizontal reliability
HAS_VER_RELIABILITY valid vertical reliability
HAS_HOR_ACCURACY_ELIP_SEMI_MAJOR valid elipsode semi major
HAS_HOR_ACCURACY_ELIP_SEMI_MINOR valid elipsode semi minor
HAS_HOR_ACCURACY_ELIP_AZIMUTH valid accuracy elipsode azimuth
HAS_GNSS_SV_USED_DATA valid gnss sv used in pos data
HAS_NAV_SOLUTION_MASK valid navSolutionMask
HAS_POS_TECH_MASK valid LocPosTechMask
HAS_SV_SOURCE_INFO valid LocSvInfoSource
HAS_POS_DYNAMICS_DATA valid position dynamics data
HAS_EXT_DOP valid gdop, tdop
HAS_NORTH_STD_DEV valid North standard deviation
HAS_EAST_STD_DEV valid East standard deviation
HAS_NORTH_VEL valid North Velocity
HAS_EAST_VEL valid East Velocity
HAS_UP_VEL valid Up Velocity

HAS_NORTH_VEL_UNC valid North Velocity Uncertainty
HAS_EAST_VEL_UNC valid East Velocity Uncertainty
HAS_UP_VEL_UNC valid Up Velocity Uncertainty
HAS_LEAP_SECONDS valid leap_seconds
HAS_TIME_UNC valid timeUncMs
HAS_NUM_SV_USED_IN_POSITION valid number of sv used
HAS_CALIBRATION_CONFIDENCE_PERCENT valid sensor calibrationConfidencePercent
HAS_CALIBRATION_STATUS valid sensor calibrationConfidence
HAS_OUTPUT_ENG_TYPE valid output engine type
HAS_OUTPUT_ENG_MASK valid output engine mask
HAS_CONFORMITY_INDEX_FIX valid conformity index
HAS_LLA_VRP_BASED valid lla vrp based
HAS_ENU_VELOCITY_VRP_BASED valid enu velocity vrp based
HAS_ALTITUDE_TYPE valid altitude type
HAS_REPORT_STATUS valid report status
HAS_INTEGRITY_RISK_USED valid integrity risk
HAS_PROTECT_LEVEL_ALONG_TRACK valid protect level along track
HAS_PROTECT_LEVEL_CROSS_TRACK valid protect level cross track
HAS_PROTECT_LEVEL_VERTICAL valid protect level vertical

4.16.2.21 enum telux::loc::GnssDataSignalTypes

Specify the GNSS signal type and RF band for jammer info and automatic gain control metric in GnssData.

Enumerator

GNSS_DATA_SIGNAL_TYPE_GPS_L1CA GPS L1CA RF Band.
GNSS_DATA_SIGNAL_TYPE_GPS_L1C GPS L1C RF Band.
GNSS_DATA_SIGNAL_TYPE_GPS_L2C_L GPS L2C_L RF Band.
GNSS_DATA_SIGNAL_TYPE_GPS_L5_Q GPS L5_Q RF Band.
GNSS_DATA_SIGNAL_TYPE_GLONASS_G1 GLONASS G1 (L1OF) RF Band.
GNSS_DATA_SIGNAL_TYPE_GLONASS_G2 GLONASS G2 (L2OF) RF Band.
GNSS_DATA_SIGNAL_TYPE_GALILEO_E1_C GALILEO E1_C RF Band.
GNSS_DATA_SIGNAL_TYPE_GALILEO_E5A_Q GALILEO E5A_Q RF Band.
GNSS_DATA_SIGNAL_TYPE_GALILEO_E5B_Q GALILEO E5B_Q RF Band.
GNSS_DATA_SIGNAL_TYPE_BEIDOU_B1_I BEIDOU B1_I RF Band.
GNSS_DATA_SIGNAL_TYPE_BEIDOU_B1C BEIDOU B1C RF Band.
GNSS_DATA_SIGNAL_TYPE_BEIDOU_B2_I BEIDOU B2_I RF Band.
GNSS_DATA_SIGNAL_TYPE_BEIDOU_B2A_I BEIDOU B2A_I RF Band.
GNSS_DATA_SIGNAL_TYPE_QZSS_L1CA QZSS L1CA RF Band.
GNSS_DATA_SIGNAL_TYPE_QZSS_L1S QZSS L1S RF Band.
GNSS_DATA_SIGNAL_TYPE_QZSS_L2C_L QZSS L2C_L RF Band.
GNSS_DATA_SIGNAL_TYPE_QZSS_L5_Q QZSS L5_Q RF Band.
GNSS_DATA_SIGNAL_TYPE_SBAS_L1_CA SBAS L1_CA RF Band.
GNSS_DATA_SIGNAL_TYPE_NAVIC_L5 NAVIC L5 RF Band.
GNSS_DATA_SIGNAL_TYPE_BEIDOU_B2A_Q BEIDOU B2A_Q RF Band. Maximum number of signal types.
GNSS_DATA_MAX_NUMBER_OF_SIGNAL_TYPES

4.16.2.22 enum telux::loc::GnssDataValidityType

Specify valid mask of data fields in GnssData.

Enumerator

HAS_JAMMER Jammer Indicator is available
HAS_AGC AGC is available

4.16.2.23 enum telux::loc::DrCalibrationStatusType

Specify the sensor calibration status in ILocationInfoEx.

Enumerator

DR_ROLL_CALIBRATION_NEEDED Indicate that roll calibration is needed. Need to take more turns on level ground.
DR_PITCH_CALIBRATION_NEEDED Indicate that pitch calibration is needed. Need to take more turns on level ground.
DR_YAW_CALIBRATION_NEEDED Indicate that yaw calibration is needed. Need to accelerate in a straight line.
DR_ODO_CALIBRATION_NEEDED Indicate that odo calibration is needed. Need to accelerate in a straight line.
DR_GYRO_CALIBRATION_NEEDED Indicate that gyro calibration is needed. Need to take more turns on level ground.

4.16.2.24 enum telux::loc::LocReqEngineType

Specifies the set of engines whose position reports are requested via startDetailedEngineReports.

Enumerator

LOC_REQ_ENGINE_FUSED_BIT Indicate that the fused/default position is needed to be reported back for the tracking sessions. The default position is the propagated/aggregated reports from all engines running on the system (e.g.: DR/SPE/PPE) according to QTI algorithm.
LOC_REQ_ENGINE_SPE_BIT Indicate that the unmodified SPE position is needed to be reported back for the tracking sessions.
LOC_REQ_ENGINE_PPE_BIT Indicate that the unmodified PPE position is needed to be reported back for the tracking sessions.
LOC_REQ_ENGINE_VPE_BIT Indicate that the unmodified VPE position is needed to be reported back for the tracking sessions.

4.16.2.25 enum telux::loc::LocationAggregationType

Specifies the type of engine for the reported fixes

Enumerator

LOC_OUTPUT_ENGINE_FUSED This is the propagated/aggregated report from the fixes of all engines running on the system (e.g.: DR/SPE/PPE).
LOC_OUTPUT_ENGINE_SPE This fix is the unmodified fix from modem GNSS engine

LOC_OUTPUT_ENGINE_PPE This is the unmodified fix from PPP engine
LOC_OUTPUT_ENGINE_VPE This is the unmodified fix from VPE engine.

4.16.2.26 enum telux::loc::PositioningEngineType

Specifies the type of engine responsible for fixes when the engine type is fused

Enumerator

STANDARD_POSITIONING_ENGINE For standard GNSS position engines.
DEAD_RECKONING_ENGINE For dead reckoning position engines.
PRECISE_POSITIONING_ENGINE For precise position engines.
VP_POSITIONING_ENGINE For VP position engine.

4.16.2.27 enum telux::loc::LeverArmType

Lever ARM type

Enumerator

LEVER_ARM_TYPE_GNSS_TO_VRP Lever arm parameters regarding the VRP (Vehicle Reference Point) w.r.t the origin (at the GNSS Antenna)
LEVER_ARM_TYPE_DR_IMU_TO_GNSS Lever arm regarding GNSS Antenna w.r.t the origin at the IMU (inertial measurement unit) for DR (dead reckoning engine)
LEVER_ARM_TYPE_VEPP_IMU_TO_GNSS Lever arm regarding GNSS Antenna w.r.t the origin at the IMU (inertial measurement unit) for VEPP (vision enhanced precise positioning engine)

Deprecated

enum type is not supported.

LEVER_ARM_TYPE_VPE_IMU_TO_GNSS Lever arm regarding GNSS Antenna w.r.t the origin at the IMU (inertial measurement unit) for VPE (vision positioning engine)

4.16.2.28 enum telux::loc::GnssMeasurementsDataValidityType

Specify valid fields in GnssMeasurementsData.

Enumerator

SV_ID_BIT Validity of svId.
SV_TYPE_BIT Validity of svType.
STATE_BIT Validity of stateMask.
RECEIVED_SV_TIME_BIT Validity of receivedSvTimeNs and receivedSvTimeSubNs.
RECEIVED_SV_TIME_UNCERTAINTY_BIT Validity of receivedSvTimeUncertaintyNs.
CARRIER_TO_NOISE_BIT Validity of carrierToNoiseDbHz.
PSEUDORANGE_RATE_BIT Validity of pseudorangeRateMps.
PSEUDORANGE_RATE_UNCERTAINTY_BIT Validity of pseudorangeRateUncertaintyMps.
ADR_STATE_BIT Validity of adrStateMask.
ADR_BIT Validity of adrMeters.
ADR_UNCERTAINTY_BIT Validity of adrUncertaintyMeters.

CARRIER_FREQUENCY_BIT Validity of carrierFrequencyHz.
CARRIER_CYCLES_BIT Validity of carrierCycles.
CARRIER_PHASE_BIT Validity of carrierPhase.
CARRIER_PHASE_UNCERTAINTY_BIT Validity of carrierPhaseUncertainty.
MULTIPATH_INDICATOR_BIT Validity of multipathIndicator.
SIGNAL_TO_NOISE_RATIO_BIT Validity of signalToNoiseRatioDb.
AUTOMATIC_GAIN_CONTROL_BIT Validity of agcLevelDb.
GNSS_SIGNAL_TYPE Validity of signal type.
BASEBAND_CARRIER_TO_NOISE Validity of basebandCarrierToNoise.
FULL_ISB Validity of fullInterSignalBias.
FULL_ISB_UNCERTAINTY Validity of fullInterSignalBiasUncertainty.

4.16.2.29 enum telux::loc::GnssMeasurementsStateValidityType

Specify GNSS measurement state in GnssMeasurementsData::stateMask.

Enumerator

UNKNOWN_BIT State is unknown.
CODE_LOCK_BIT State is "code lock".
BIT_SYNC_BIT State is "bit sync".
SUBFRAME_SYNC_BIT State is "subframe sync".
TOW_DECODED_BIT State is "tow decoded".
MSEC_AMBIGUOUS_BIT State is "msec ambiguous".
SYMBOL_SYNC_BIT State is "symbol sync".
GLO_STRING_SYNC_BIT State is "GLONASS string sync".
GLO_TOD_DECODED_BIT State is "GLONASS TOD decoded".
BDS_D2_BIT_SYNC_BIT State is "BDS D2 bit sync".
BDS_D2_SUBFRAME_SYNC_BIT State is "BDS D2 subframe sync".
GAL_E1BC_CODE_LOCK_BIT State is "Galileo E1BC code lock".
GAL_E1C_2ND_CODE_LOCK_BIT State is "Galileo E1C second code lock".
GAL_E1B_PAGE_SYNC_BIT State is "Galileo E1B page sync".
SBAS_SYNC_BIT State is "SBAS sync".

4.16.2.30 enum telux::loc::GnssMeasurementsAdrStateValidityType

Specify accumulated delta range state in GnssMeasurementsData::adrStateMask.

Enumerator

UNKNOWN_STATE State is unknown.
VALID_BIT State is valid.
RESET_BIT State is "reset".
CYCLE_SLIP_BIT State is "cycle slip".

4.16.2.31 enum telux::loc::GnssMeasurementsMultipathIndicator

Specify the GNSS multipath indicator state in GnssMeasurementsData::multipathIndicator.

Enumerator

UNKNOWN_INDICATOR Multipath indicator is unknown.
PRESENT Multipath indicator is present.
NOT_PRESENT Multipath indicator is not present.

4.16.2.32 enum telux::loc::GnssMeasurementsClockValidityType

Specify the valid fields in GnssMeasurementsClock.

Enumerator

LEAP_SECOND_BIT Validity of leapSecond.
TIME_BIT Validity of timeNs.
TIME_UNCERTAINTY_BIT Validity of timeUncertaintyNs.
FULL_BIAS_BIT Validity of fullBiasNs.
BIAS_BIT Validity of biasNs.
BIAS_UNCERTAINTY_BIT Validity of biasUncertaintyNs.
DRIFT_BIT Validity of driftNsps.
DRIFT_UNCERTAINTY_BIT Validity of driftUncertaintyNsps.
HW_CLOCK_DISCONTINUITY_COUNT_BIT Validity of hwClockDiscontinuityCount.

4.16.2.33 enum telux::loc::GnssReportDCType

Disaster and crisis report types that are currently supported by the GNSS Engine.

Enumerator

QZSS_JMA_DISASTER_PREVENTION_INFO Disaster Prevention information provided by Japan Meteorological Agency.
QZSS_NON_JMA_DISASTER_PREVENTION_INFO Disaster Prevention information provided by other organizations.

4.16.2.34 enum telux::loc::LeapSecondInfoValidityType

Specify the valid fields in LeapSecondInfo.

Enumerator

LEAP_SECOND_SYS_INFO_CURRENT_LEAP_SECONDS_BIT Validity of LeapSecondInfo::current.
LEAP_SECOND_SYS_INFO_LEAP_SECOND_CHANGE_BIT Validity of LeapSecondInfo::info.

4.16.2.35 enum telux::loc::LocationSystemInfoValidityType

Specify the set of valid fields in LocationSystemInfo

Enumerator

LOCATION_SYS_INFO_LEAP_SECOND contains current leap second or leap second change info

4.16.2.36 enum telux::loc::GnssEnergyConsumedInfoValidityType

Specify the valid fields in GnssEnergyConsumedInfo.

Enumerator

ENERGY_CONSUMED_SINCE_FIRST_BOOT_BIT validity of GnssEnergyConsumedInfo

4.16.2.37 enum telux::loc::AidingDataType

Specifies the set of aiding data. This is referenced in the deleteAidingData for deleting any aiding data.

Enumerator

AIDING_DATA_EPHEMERIS Mask to delete ephemeris aiding data
AIDING_DATA_DR_SENSOR_CALIBRATION Mask to delete calibration data from dead reckoning position engine

4.16.2.38 enum telux::loc::TerrestrialTechnologyType

Specifies the set of terrestrial technologies.

Enumerator

GTP_WWAN Cell-based technology

4.16.2.39 enum telux::loc::NmeaSentenceType

Specifies the HLOS generated NMEA sentence types.

Enumerator

GGA GGA NMEA sentence
RMC RMC NMEA sentence
GSA GSA NMEA sentence
VTG VTG NMEA sentence
GNS GNS NMEA sentence
DTM DTM NMEA sentence
GPGSV GPGSV NMEA sentence for SVs from GPS constellation
GLGSV GLGSV NMEA sentence for SVs from GLONASS constellation
GAGSV GAGSV NMEA sentence for SVs from GALILEO constellation
GQGSV GQGSV NMEA sentence for SVs from QZSS constellation
GBGSV GBGSV NMEA sentence for SVs from BEIDOU constellation
GIGSV GIGSV NMEA sentence for SVs from NAVIC constellation
ALL All NMEA sentences

4.16.2.40 enum telux::loc::GeodeticDatumType [strong]

Specify the Geodetic datum for NMEA sentence types that are generated.

Enumerator

GEODETIC_TYPE_NONE No type
GEODETIC_TYPE_WGS_84 Geodetic datum type to indicate the use of World Geodetic System 1984 (WGS84) system
GEODETIC_TYPE_PZ_90 Geodetic datum type to indicate the use of PZ90/GLONASS system

4.16.2.41 enum telux::loc::RobustLocationConfigType

Specify the valid mask for robust location configuration used by the GNSS standard position engine (SPE).

Enumerator

VALID_ENABLED Validity of enabled
VALID_ENABLED_FOR_E911 Validity of enabledForE911.
VALID_VERSION Validity of version.

4.16.2.42 enum telux::loc::DRConfigValidityType

Specify the valid mask for the configuration parameters of dead reckoning position engine

Enumerator

BODY_TO_SENSOR_MOUNT_PARAMS_VALID Validity of body to sensor mount parameters.
VEHICLE_SPEED_SCALE_FACTOR_VALID Validity of vehicle speed scale factor.
VEHICLE_SPEED_SCALE_FACTOR_UNC_VALID Validity of vehicle speed scale factor uncertainty.
GYRO_SCALE_FACTOR_VALID Validity of gyro scale factor.
GYRO_SCALE_FACTOR_UNC_VALID Validity of gyro scale factor uncertainty.

4.16.2.43 enum telux::loc::GnssReportType

Specifies the set of gnss reports.

Enumerator

LOCATION Location reports
SATELLITE_VEHICLE Satellite reports
NMEA Nmea reports
DATA Data reports
MEASUREMENT Low rate measurement reports. Currently the rate is defined to be 1 Hz.
HIGH_RATE_MEASUREMENT High rate measurement reports. Currently the rate is defined to be 10 Hz. Client cannot specify rates. The data in high rate would be different that from low rate. Also there might be difference in accuracy of fields for the both the rates.
DISASTER_CRISIS

4.16.2.44 enum telux::loc::EngineType `[strong]`

Specify the position engine types

Enumerator

UNKNOWN Unknown engine type.

SPE Standard GNSS position engine.
PPE Precise position engine.
DRE Dead reckoning position engine.
VPE Vision positioning engine.

4.16.2.45 enum telux::loc::LocationEngineRunState [strong]

Specify the position engine run state

Enumerator

UNKNOWN Unknown engine run state.
SUSPENDED Request the position engine to be put into suspended state.
RUNNING Request the position engine to be put into running state.

4.16.2.46 enum telux::loc::ReportStatus [strong]

Specify the status of the report

Enumerator

UNKNOWN Report status is unknown.
SUCCESS Report status is successful. The engine is able to calculate the desired fix. Most of the fields in ILocationInfoEx will be valid.
INTERMEDIATE Report is still in progress. The engine has not completed its calculations when this report was generated. Accuracy of various fields is non-optimal. Only some of the fields in ILocationInfoEx will be valid.
FAILURE Report status has failed. The engine is not able to calculate the fix. Most of the fields in ILocationInfoEx will be invalid.

4.16.2.47 enum telux::loc::DebugLogLevel [strong]

Specify the logcat debug level during XTRA's param configuration. Currently, only XTRA daemon will support the runtime configuration of the debug log level.

Enumerator

DEBUG_LOG_LEVEL_NONE No message is logged.
DEBUG_LOG_LEVEL_ERROR Only error level debug messages will get logged.
DEBUG_LOG_LEVEL_WARNING Only warning and error level debug messages will get logged.
DEBUG_LOG_LEVEL_INFO Only info, warning and error level debug messages will get logged.
DEBUG_LOG_LEVEL_DEBUG Only debug, info, warning and error level debug messages will get logged.
DEBUG_LOG_LEVEL_VERBOSE Verbose, debug, info, warning and error level debug messages will get logged.

4.16.2.48 enum telux::loc::XtraDataStatus [strong]

Provides the status of the previously downloaded Xtra data.

Enumerator

STATUS_UNKNOWN If XTRA feature is disabled or if XTRA feature is enabled, but XTRA daemon has not yet retrieved the assistance data status from modem on early stage of device bootup, xtra data status will be unknown.

STATUS_NOT_AVAIL If XTRA feature is enabled, but XTRA data is not present on the device.

STATUS_NOT_VALID If XTRA feature is enabled, XTRA data has been downloaded ever but no longer valid.

STATUS_VALID If XTRA feature is enabled, XTRA data has been downloaded and is currently valid.

4.16.2.49 enum telux::loc::LocConfigIndicationsType

Enum of all the possible indications invoked by a Location Configurator listener.

Enumerator

LOC_CONF_IND_XTRA_STATUS

4.16.3 Variable Documentation

4.16.3.1 const float telux::loc::UNKNOWN_CARRIER_FREQ = -1

4.16.3.2 const int telux::loc::UNKNOWN_SIGNAL_MASK = 0

4.16.3.3 const double telux::loc::UNKNOWN_BASEBAND_CARRIER_NOISE = 0.0

4.16.3.4 const uint64_t telux::loc::UNKNOWN_TIMESTAMP = 0

4.16.3.5 const float telux::loc::DEFAULT_TUNC_THRESHOLD = 0.0

Default value for threshold of time uncertainty. Units: milli-seconds.

4.16.3.6 const int telux::loc::DEFAULT_TUNC_ENERGY_THRESHOLD = 0

Default value for energy consumed of time uncertainty. The default here means that the engine is allowed to use infinite power. Units: 100 micro watt second.

4.16.3.7 const uint64_t telux::loc::INVALID_ENERGY_CONSUMED = 0xffffffffffffffff

0xffffffffffffffff indicates an invalid reading for energy consumed info.

4.16.3.8 const uint32_t telux::loc::DEFAULT_GNSS_REPORT = 0xffffffff

0xffffffff indicates all the reports are enabled.

4.16.3.9 const float telux::loc::UNKNOWN_SV_TIME_SUB_NS = -1

Unknown Sub nanoseconds portion of the received GNSS time.

4.17 Common

This section contains APIs related to Logger, Command Callbacks, Error Codes and Version information. Also contains Macros to print message at different log level.

4.17.1 Data Structure Documentation

4.17.1.1 class telux::common::ICommandCallback

Base command callback class is responsible for single shot asynchronous callback. This callback will be invoked only once when the operation succeeds or fails.

Public member functions

- virtual ~ICommandCallback ()

4.17.1.1.1 Constructors and Destructors

4.17.1.1.1.1 virtual telux::common::ICommandCallback::~ICommandCallback ( ) [virtual]

4.17.1.2 class telux::common::ICommandResponseCallback

General command response callback for most of the requests, client needs to implement this interface to get single shot response.

The methods in callback can be invoked from multiple different threads. The implementation should be thread safe.

Public member functions

- virtual void commandResponse (ErrorCode error)=0
- virtual ~ICommandResponseCallback ()

4.17.1.2.1 Constructors and Destructors

4.17.1.2.1.1 virtual telux::common::ICommandResponseCallback::~ICommandResponseCallback ( ) [virtual]

4.17.1.2.2 Member Function Documentation

4.17.1.2.2.1 virtual void telux::common::ICommandResponseCallback::commandResponse (ErrorCode *error*) [pure virtual]

This function is called with the response to the command operation.

Parameters

in	*error*	- ErrorCode

4.17.1.3 class telux::common::DeviceConfig

Static Public Member Functions

- static bool isMultiSimSupported ()

4.17.1.3.1 Member Function Documentation

4.17.1.3.1.1 static bool telux::common::DeviceConfig::isMultiSimSupported ( ) [static]

Check whether multi SIM support available.

Returns

bool to determine multi SIM support

4.17.1.4 class telux::common::Log

Static Public Member Functions

- template<typename... MessageArgs>
  static void logMessage (LogLevel logLevel, const std::string &fileName, const std::string &lineNo, const int &component, MessageArgs...params)

4.17.1.4.1 Member Function Documentation

4.17.1.4.1.1 template<typename... MessageArgs> static void telux::common::Log::logMessage (LogLevel *logLevel,* const std::string & *fileName,* const std::string & *lineNo,* const int & *component,* MessageArgs... *params*) [static]

4.17.1.5 struct telux::common::SdkVersion

Structure of major, minor and patch version

Data fields

Type	Field	Description
int	major	Major Version: This number will be incremented whenever significant changes or features are introduced
int	minor	Minor Version: This number will be incremented when smaller features with some new APIs are introduced.
int	patch	Patch Version: If the release only contains bug fixes, but no API change then the patch version would be incremented.

4.17.1.6 class telux::common::Version

Provides version of SDK.

Static Public Member Functions

- static std::string getReleaseName ()
- static SdkVersion getSdkVersion ()

4.17.1.6.1 Member Function Documentation

4.17.1.6.1.1 static std::string telux::common::Version::getReleaseName ( ) [static]

Get the release name.

Returns

String contains release name

4.17.1.6.1.2 static SdkVersion telux::common::Version::getSdkVersion ( ) [static]

Get the Telematics SDK version, for example: 01.00.00

Returns

SdkVersion structure of major, minor and patch version

4.17.2 Enumeration Type Documentation

4.17.2.1 enum telux::common::Status [strong]

Defines all the status codes that all Telematics SDK APIs can return

Enumerator

SUCCESS API processing is successful, returned parameters are valid
FAILED API processing failure.
NOCONNECTION Connection to Socket server has not been established
NOSUBSCRIPTION Subscription not available
INVALIDPARAM Input parameters are invalid
INVALIDSTATE Invalid State
NOTREADY Subsystem is not ready
NOTALLOWED Operation not allowed
NOTIMPLEMENTED Functionality not implemented
CONNECTIONLOST Connection to Socket server lost
EXPIRED Expired
ALREADY Already registered handler
NOSUCH No such object
NOTSUPPORTED Not supported on target platform
NOMEMORY Not sufficient memory to process the request

4.17.2.2 enum telux::common::ErrorCode [strong]

Generic Error code for each API responses

Enumerator

SUCCESS No error
RADIO_NOT_AVAILABLE If radio did not start or is resetting
GENERIC_FAILURE Generic Failure
PASSWORD_INCORRECT For PIN/PIN2 methods only
SIM_PIN2 Operation requires SIM PIN2 to be entered
SIM_PUK2 Operation requires SIM PIN2 to be entered
REQUEST_NOT_SUPPORTED Not Supported request
CANCELLED Cancelled
OP_NOT_ALLOWED_DURING_VOICE_CALL Data operation are not allowed during voice call on a Class C GPRS device
OP_NOT_ALLOWED_BEFORE_REG_TO_NW Data operation are not allowed before device registers in network
SMS_SEND_FAIL_RETRY Fail to send SMS and need retry
SIM_ABSENT Fail to set the location where CDMA subscription shall be retrieved because of SIM or RUIM are absent
SUBSCRIPTION_NOT_AVAILABLE Fail to find CDMA subscription from specified location
MODE_NOT_SUPPORTED Hardware does not support preferred network type
FDN_CHECK_FAILURE Command failed because recipient is not on FDN list
ILLEGAL_SIM_OR_ME Network selection failed due to illegal SIM or ME
MISSING_RESOURCE No logical channel available
NO_SUCH_ELEMENT Application not found on SIM
DIAL_MODIFIED_TO_USSD DIAL request modified to USSD
DIAL_MODIFIED_TO_SS DIAL request modified to SS
DIAL_MODIFIED_TO_DIAL DIAL request modified to DIAL with different data
USSD_MODIFIED_TO_DIAL USSD request modified to DIAL
USSD_MODIFIED_TO_SS USSD request modified to SS
USSD_MODIFIED_TO_USSD USSD request modified to different USSD request
SS_MODIFIED_TO_DIAL SS request modified to DIAL
SS_MODIFIED_TO_USSD SS request modified to USSD
SUBSCRIPTION_NOT_SUPPORTED Subscription not supported
SS_MODIFIED_TO_SS SS request modified to different SS request
LCE_NOT_SUPPORTED LCE service not supported
NO_MEMORY Not sufficient memory to process the request
INTERNAL_ERR Hit unexpected vendor internal error scenario
SYSTEM_ERR Hit platform or system error
MODEM_ERR Hit unexpected modem error
INVALID_STATE Unexpected request for the current state
NO_RESOURCES Not sufficient resource to process the request
SIM_ERR Received error from SIM card
INVALID_ARGUMENTS Received invalid arguments in request
INVALID_SIM_STATE Cannot process the request in current SIM state
INVALID_MODEM_STATE Cannot process the request in current Modem state
INVALID_CALL_ID Received invalid call id in request
NO_SMS_TO_ACK ACK received when there is no SMS to ack

NETWORK_ERR Received error from network
REQUEST_RATE_LIMITED Operation denied due to overly-frequent requests
SIM_BUSY SIM is busy
SIM_FULL The target EF is full
NETWORK_REJECT Request is rejected by network
OPERATION_NOT_ALLOWED Not allowed the request now
EMPTY_RECORD The request record is empty
INVALID_SMS_FORMAT Invalid SMS format
ENCODING_ERR Message not encoded properly
INVALID_SMSC_ADDRESS SMSC address specified is invalid
NO_SUCH_ENTRY No such entry present to perform the request
NETWORK_NOT_READY Network is not ready to perform the request
NOT_PROVISIONED Device does not have this value provisioned
NO_SUBSCRIPTION Device does not have subscription
NO_NETWORK_FOUND Network cannot be found
DEVICE_IN_USE Operation cannot be performed because the device is currently in use
ABORTED Operation aborted
INCOMPATIBLE_STATE Operation cannot be performed because the device is in incompatible state
NO_EFFECT Given request had to no effect
DEVICE_NOT_READY Device not ready
MISSING_ARGUMENTS Missing one or more arguments
PIN_PERM_BLOCKED PIN is permanently blocked. The SIM is unusable.
PIN_BLOCKED PIN is blocked. Unblock operation must be issued.
MALFORMED_MSG Message was not formulated correctly by the control point or the message was corrupted during transmission
INTERNAL Internal error
CLIENT_IDS_EXHAUSTED Client IDs exhausted
UNABORTABLE_TRANSACTION The specified transaction could not be aborted
INVALID_CLIENT_ID Could not find client's request
NO_THRESHOLDS No thresholds specified in enable signal strength
INVALID_HANDLE Invalid client handle was received
INVALID_PROFILE Invalid profile index specified
INVALID_PINID PIN in the request is invalid.
INCORRECT_PIN PIN in the request is incorrect.
CALL_FAILED Call origination failed in the lower layers
OUT_OF_CALL Request issued when packet data session disconnected
MISSING_ARG TLV was missing in the request.
ARG_TOO_LONG Path in the request was too long.
INVALID_TX_ID The transaction ID supplied in the request does not match any pending transaction i.e. either the transaction was not received or it is already executed by the device
OP_NETWORK_UNSUPPORTED Selected operation is not supported by the network
OP_DEVICE_UNSUPPORTED Operation is not supported by device or SIM card
NO_FREE_PROFILE Maximum number of profiles are stored in the device and there is no more storage available to create a new profile
INVALID_PDP_TYPE PDP type specified is not supported
INVALID_TECH_PREF Invalid technology preference
INVALID_PROFILE_TYPE Invalid profile type is specified
INVALID_SERVICE_TYPE Invalid service type
INVALID_REGISTER_ACTION Invalid register action value specified in request
INVALID_PS_ATTACH_ACTION Invalid PS attach action value specified in request

AUTHENTICATION_FAILED Authentication error.
SIM_NOT_INITIALIZED PIN is not yet initialized because the SIM initialization has not finished. Try the PIN operation later.
MAX_QOS_REQUESTS_IN_USE Maximum QoS requests in use
INCORRECT_FLOW_FILTER Incorrect flow filter
NETWORK_QOS_UNAWARE Network QoS unaware
INVALID_ID Invalid call ID was sent in the request
REQUESTED_NUM_UNSUPPORTED Requested message ID is not supported by the currently running software
INTERFACE_NOT_FOUND Cannot retrieve the FMC interface
FLOW_SUSPENDED Flow suspended
INVALID_DATA_FORMAT Invalid data format
GENERAL General error
UNKNOWN Unknown error
INVALID_ARG Parameters passed as input were invalid
INVALID_INDEX MIP profile index is not within the valid range
NO_ENTRY No message exists at the specified memory storage designation
DEVICE_STORAGE_FULL Memory storage specified in the request is full
CAUSE_CODE There was an error in the request
MESSAGE_NOT_SENT Message could not be sent
MESSAGE_DELIVERY_FAILURE Message could not be delivered
INVALID_MESSAGE_ID Message ID specified for the message is invalid
ENCODING Message is not encoded properly
AUTHENTICATION_LOCK Maximum number of authentication failures has been reached
INVALID_TRANSITION Selected operating mode transition from the current operating mode is invalid
NOT_A_MCAST_IFACE Not a MCAST interface
MAX_MCAST_REQUESTS_IN_USE MCAST request in use
INVALID_MCAST_HANDLE An invalid MCAST handle
INVALID_IP_FAMILY_PREF IP family preference is invalid
SESSION_INACTIVE Session inactive
SESSION_INVALID Session not valid
SESSION_OWNERSHIP Session ownership error
INSUFFICIENT_RESOURCES Response is longer than the maximum supported size
DISABLED Disabled
INVALID_OPERATION Device is not expecting the request.
INVALID_QMI_CMD Invalid QMI command
TPDU_TYPE Message in memory contains a TPDU type that cannot be read
SMSC_ADDR SMSC address specified is invalid
INFO_UNAVAILABLE Information is not available
SEGMENT_TOO_LONG PRL segment size is too large
SEGMENT_ORDER PRL segment order is incorrect
BUNDLING_NOT_SUPPORTED Bundling not supported
OP_PARTIAL_FAILURE Some personalization codes were set but an error prevented
POLICY_MISMATCH Network policy does not match a valid NAT
SIM_FILE_NOT_FOUND File is not present on the card.
EXTENDED_INTERNAL Error from the the DS profile module, the extended error
ACCESS_DENIED Access to the requested file is denied. This can occur when there is an attempt to access a PIN-protected file.
HARDWARE_RESTRICTED Selected operating mode is invalid with the current wireless disable setting

ACK_NOT_SENT ACK could not be sent
INJECT_TIMEOUT Inject timeout
FDN_RESTRICT FDN restriction
SUPS_FAILURE_CAUSE Indicates supplementary services failure information;
NO_RADIO Radio is not available
NOT_SUPPORTED Operation is not supported
CARD_CALL_CONTROL_FAILED SIM/R-UIM call control failed
NETWORK_ABORTED Operation was released abruptly by the network
MSG_BLOCKED Message blocked
INVALID_SESSION_TYPE Invalid session type
INVALID_PB_TYPE Invalid Phone Book type
NO_SIM Action is being performed on a SIM that is not initialized.
PB_NOT_READY Phone Book not ready
PIN_RESTRICTION PIN restriction
PIN2_RESTRICTION PIN2 restriction
PUK_RESTRICTION PUK restriction
PUK2_RESTRICTION PUK2 restriction
PB_ACCESS_RESTRICTED Phone Book access restricted
PB_DELETE_IN_PROG Phone Book delete in progress
PB_TEXT_TOO_LONG Phone Book text too long
PB_NUMBER_TOO_LONG Phone Book number too long
PB_HIDDEN_KEY_RESTRICTION Phone Book hidden key restriction
PB_NOT_AVAILABLE Phone Book not available
DEVICE_MEMORY_ERROR Device memory error
NO_PERMISSION No permission
TOO_SOON Too soon
TIME_NOT_ACQUIRED Time not acquired
OP_IN_PROGRESS Operation is in progress
DS_PROFILE_REG_RESULT_FAIL General failure
DS_PROFILE_REG_RESULT_ERR_INVAL_HNDL Request contains an invalid profile handle
DS_PROFILE_REG_RESULT_ERR_INVAL_OP Invalid operation was requested
DS_PROFILE_REG_RESULT_ERR_INVAL_PROFILE_TYPE Request contains an invalid technology type
DS_PROFILE_REG_RESULT_ERR_INVAL_PROFILE_NUM Request contains an invalid profile number
DS_PROFILE_REG_RESULT_ERR_INVAL_IDENT Request contains an invalid profile identifier
DS_PROFILE_REG_RESULT_ERR_INVAL Request contains an invalid argument other than profile number and profile identifier received
DS_PROFILE_REG_RESULT_ERR_LIB_NOT_INITED Profile registry has not been initialized yet
DS_PROFILE_REG_RESULT_ERR_LEN_INVALID Request contains a parameter with invalid length
DS_PROFILE_REG_RESULT_LIST_END End of the profile list was reached while searching for the requested profile
DS_PROFILE_REG_RESULT_ERR_INVAL_SUBS_ID Request contains an invalid subscription identifier
DS_PROFILE_REG_INVAL_PROFILE_FAMILY Request contains an invalid profile family
DS_PROFILE_REG_PROFILE_VERSION_MISMATCH Version mismatch
REG_RESULT_ERR_OUT_OF_MEMORY Out of memory
DS_PROFILE_REG_RESULT_ERR_FILE_ACCESS File access error
DS_PROFILE_REG_RESULT_ERR_EOF End of field
REG_RESULT_ERR_VALID_FLAG_NOT_SET A valid flag is not set

REG_RESULT_ERR_OUT_OF_PROFILES Out of profiles
REG_RESULT_NO_EMERGENCY_PDN_SUPPORT No emergency PDN support
V2X_ERR_EXCEED_MAX Exceed max allowed number
V2X_ERR_V2X_DISABLED V2x mode was not enabled
V2X_ERR_UNKNOWN_SERVICE_ID The service id unknown
V2X_ERR_SRV_ID_L2_ADDRS_NOT_COMPATIBLE The service Id mismatch with L2 addr
V2X_ERR_PORT_UNAVAIL The port was occupied by others
DS_PROFILE_3GPP_INVAL_PROFILE_FAMILY Request contains an invalid 3GPP profile family
DS_PROFILE_3GPP_ACCESS_ERR Error was encountered while accessing the 3GPP profiles
DS_PROFILE_3GPP_CONTEXT_NOT_DEFINED Specified 3GPP profile does not have a valid context
DS_PROFILE_3GPP_VALID_FLAG_NOT_SET Specified 3GPP profile is marked invalid
DS_PROFILE_3GPP_READ_ONLY_FLAG_SET Specified 3GPP profile is marked read-only
DS_PROFILE_3GPP_ERR_OUT_OF_PROFILES Creation of a new 3GPP profile failed because the limit of 16 profiles has already been reached
DS_PROFILE_3GPP2_ERR_INVALID_IDENT_FOR_PROFILE Invalid profile identifier was received as part of the 3GPP2 profile modification request
DS_PROFILE_3GPP2_ERR_OUT_OF_PROFILE Creation of a new 3GPP2 profile failed because the limit has already been reached
INTERNAL_ERROR Internal error
SERVICE_ERROR Service error
TIMEOUT_ERROR Timeout error
EXTENDED_ERROR Extended error
PORT_NOT_OPEN_ERROR Port not open
MEMCOPY_ERROR Memory copy error
INVALID_TRANSACTION Invalid transaction
ALLOCATION_FAILURE Allocation failure
TRANSPORT_ERROR Transport error
PARAM_ERROR Parameter error
INVALID_CLIENT Invalid client
FRAMEWORK_NOT_READY Framework not ready
INVALID_SIGNAL Invalid signal
TRANSPORT_BUSY_ERROR Transport busy error
SUBSYSTEM_UNAVAILABLE Underlying service currently unavailable
OPERATION_TIMEOUT Timeout error
ROLLBACK_FAILED Rollback to initial state failed
ROT_ALREADY_SET Root of trust already configured
UNSUPPORTED_PURPOSE Unsupported use of the key
INCOMPATIBLE_PURPOSE Incompatible purpose
UNSUPPORTED_ALGO Unsupported algorithm
INCOMPATIBLE_ALGO Incompatible algorithm
UNSUPPORTED_KEY_SIZE Unsupported key size
UNSUPPORTED_BLOCK_MODE Unsupported block mode
INCOMPATIBLE_BLOCK_MODE Incompatible block mode
UNSUPPORTED_MAC_LEN Unsupported MAC length
UNSUPPORTED_PADDING_MODE Unsupported padding mode
UNSUPPORTED_DIGEST Unsupported digest
INCOMPATIBLE_DIGEST Incompatible digest
INVAL_EXP_TIME Invalid expiration time
INVAL_USR_ID Invalid user ID
INVAL_AUTH_TIMEOUT Invalid authorization timeout

UNSUPPORTED_KEY_FMT Unsupported key format
INCOMPATIBLE_KEY_FMT Incompatible key format
UNSUPPORTED_KEY_ENC_ALGO Unsupported key encryption algorithm (for PKCS8 & PKCS12)
UNSUPPORTED_KEY_VRFY_ALGO Unsupported key verification algorithm (for PKCS8 & PKCS12)
INVAL_IN_LEN Invalid input length
INVAL_KEY_EXPRT_OPTNS Invalid oprions for key export
DELEGATION_NOT_ALLOWED Delegation not allowed
KEY_NOT_YET_VALID Key still not valid
KEY_EXPIRED Key has expired
KEY_USR_NOT_AUTHENTICATED Key user not authenticated
OUT_PARAMETER_NULL Null output argument
INVAL_OPERATION_HNDL Invalid operation handle
INSUFFICIENT_BUF_SPACE Insufficient buffer space
VERIFICATION_FAILED Verifcation failed
TOO_MANY_OPS Too many operations
UNEXPECTED_NULL_PTR Unexpected null pointer
INVAL_KEY_BLOB Invalid key blob
IMPORTED_KEY_NOT_ENC Imported key not encrypted
IMPORTED_KEY_DEC_FAIL Imported key decryption failed
IMPORTED_KEY_NOT_SIGNED Imported key not signed
IMPORTED_KEY_VRFY_FAIL Imported key verification failed
UNSUPPORTED_TAG Unsupported tag
INVAL_TAG Invalid TAG
IMPORT_PARAM_MISMATCH Mismatch in import parameters
SEC_HW_ACCESS_DENIED Secure hardware access denied
CONCUR_ACCESS_CONFLICT Concurrent access conflict
SEC_HW_BUSY Secure hardware busy
SEC_HW_COM_FAIL Secure hardware communication failed
UNSUPPORTED_EC_FIELD Unsupported EC field
MISSING_NONCE Missing nonce
INVAL_NONCE Invalid nonce
MISSING_MAC_LEN Missing MAC length
KEY_RATE_LIMIT_EXCEEDED Key limit exceeded
CALLER_NONCE_PROHIBITED Caller nonce proibited
KEY_MAX_OPS_EXCEEDED Key maximum operations exceeded
INVAL_MAC_LEN Invalid MAC length
MISSING_MIN_MAC_LEN Missing minimum MAC length
UNSUPPORTED_MIN_MAC_LEN Unsupported minimum MAC length
UNSUPPORTED_KDF Unsupported KDF
UNSUPPORTED_EC_CURVE Unsupported EC curve
KEY_REQ_UPGRADE Key requires upgrade
ATTESTATION_CHLNG_MIS Attestation challenge missing
KM_NOT_CONFGRD Keymaster not configured
ATTESTATION_APPID_MIS Attestation app ID missing
CANNOT_ATTEST_IDS Can not attest IDs
UNIMPLEMENTED Unimplemented
VER_MISMATCH Version mismatch
SOTER_ERR Soter error
DMA_ERR HSDMA error

DIV_ERR Divided by error
OVERFLOW_UNDERFLOW Arithmetic overflow or underflow
RNG_UNSEEDED Read from unseeded ring
MEM_ERR Memory read
MODULUS_ERR Modulus error
DECODING_ERR Decode error
INVALID_LENGTH Invalid length of data

4.17.2.3 enum telux::common::ServiceStatus [strong]

Service status.

Enumerator

SERVICE_UNAVAILABLE
SERVICE_AVAILABLE
SERVICE_FAILED

4.17.2.4 enum telux::common::ProcType [strong]

This applies in system architectures where the modem is attached to an External Application Processor(EAP). The operations associated with the ProcType can be performed by SDK either on EAP or the modem's Internal Application Processor(IAP). This type specifies where the operation is carried out.

Enumerator

LOCAL_PROC Perform the operation on the processor where the API is invoked.
REMOTE_PROC Perform the operation on the application processor other than where the API is invoked.

4.17.2.5 enum telux::common::LogLevel [strong]

Indicates supported logging levels.

Enumerator

LEVEL_NONE
LEVEL_PERF Prints messages with nanoseconds precision timestamp
LEVEL_ERROR Prints perf and error messages only
LEVEL_WARNING Prints perf, error and warning messages
LEVEL_INFO Prints perf, errors, warning and information messages
LEVEL_DEBUG Full logging including debug messages

4.18 C APIs

- C Common APIs
- C Kinematics APIs
- C Radio APIs
- C Vehicle APIs
- C Config APIs
- C Packet APIs

This section contains C APIs related to Cellular-V2X operation.

4.19 C Common APIs

This section contains C Common APIs related to Cellular-V2X operation.

The following common typedefs and macros are used by the C-V2X C APIs.

4.19.1 Data Structure Documentation

4.19.1.1 struct v2x_api_ver_t

Contains retrieved information about the SDK API library that is called. Each SDK component (Kinematics, Radio, Vehicle Data) implements a method to return this structure.

Data fields

Type	Field	Description
uint32_t	version_num	Version number of the interface.
char	build_date_str[128]	Date of the build (part of the data string).
char	build_time_str[128]	Time of the build (part of the data string).
char	build_details_str[128]	Build details (part of the data string).

4.19.2 Enumeration Type Documentation

4.19.2.1 enum v2x_status_enum_type

Valid types for subsystem status, and return status codes for API function calls and callbacks.

Enumerator

V2X_STATUS_SUCCESS Operation is successful.

V2X_STATUS_FAIL Operation is unsuccessful. This is a generic error failure status that can be due to radio hardware resource limitations, geofencing, and so on.

V2X_STATUS_ENO_MEMORY Failure due to a memory allocation issue.

V2X_STATUS_EBADPARM One of the supplied parameters is bad.

V2X_STATUS_EALREADY Attempted step was already issued, and this call is not required.

V2X_STATUS_KINETICS_PLACEHOLDER Begin the return codes associated with the Kinematics interface.

V2X_STATUS_RADIO_PLACEHOLDER Begin the return codes associated with the Radio interface.

V2X_STATUS_ECHANNEL_UNAVAILABLE Requested radio frequency cannot be used at this time.

V2X_STATUS_RADIO_NOT_READY Radio initialization failed due to v2x status.

V2X_STATUS_VEHICLE_PLACEHOLDER Begin the return codes associated with the Vehicle Data interface.

4.20 C Kinematics APIs

This section contains C Kinematics APIs related to Cellular-V2X operation. For any new CV2x development, it is recommended to use the C++ telux::loc::ILocationManager APIs.

Abstraction of the system GNSS + DR solution for returning precision fixes with low latency via callbacks. This solution is used each time a fix is available, and it supports multiple callbacks to a short list of clients.

Common types are used for the motion and location reporting system of the platform. These types include the structures that are used to both configure the Kinematics subsystem and to report periodic fixes. The fixes are combinations of inertial/motion data and GNSS solutions determined (possibly directly) from satellite processing or dead-reckoning in degraded SV reception.

4.20.1 Define Documentation

4.20.1.1 #define V2X_KINEMATICS_HANDLE_BAD (-1)

Invalid handle returned by v2x_kinematics_init() upon an error.

4.20.2 Data Structure Documentation

4.20.2.1 struct v2x_GNSSstatus_t

Contains status information for the GNSS satellite.

This structure is used for each reported fix to indicate the quality of the available constellation (or whether the constellation is not available).

Data fields

Type	Field	Description
bool	unavailable	Specifies whether a constellation is not equipped or is unavailable. **Supported values:** • 0 – GNSS is available • 1 – GNSS is unavailable
bool	aPDOPof↩ Under5	Specifies whether dilution of precision is greater than 5. **Supported values:** • 0 – Not greater than 5 • 1 – Greater than 5
bool	inViewOf↩ Under5	Specifies whether fewer than five satellites are in view. **Supported values:** • 0 – Five or more satellites are in view • 1 – Fewer than five satellites are in view
bool	local↩ Corrections↩ Present	Specifies whether DGPS type corrections are used. **Supported values:** • 0 – Not used • 1 – Used

Type	Field	Description
bool	network↩ Corrections↩ Present	Specifies whether RTK type corrections are used. **Supported values:** • 0 – Not used • 1 – Used

4.20.2.2 struct v2x_gnss_fix_rates_supported_list_t

Defines supported GNSS fix generation rates (such as 1 Hz, 5 Hz, 10 Hz).

Data fields

Type	Field	Description
uint32_t	qty_rates_↩ supported	Specify whether the listing or discovery of the supported rates is supported. **Supported values:** • 0 – Not supported • 1 – Supported
pb_size_t	rates_↩ supported_hz↩ _array_count	Number of supported rates.
uint32_t	rates_↩ supported_hz↩ _array[32]	Array of data rates supported by the API.

4.20.2.3 struct v2x_init_t

Defines client initialization.

Data fields

Type	Field	Description
uint32_t	log_level_mask	Log levels as defined in syslog.h.
char	server_ip_↩ addr[32]	IP address of the server.

4.20.2.4 struct v2x_kinematics_capabilities_t_feature_flags_t

Defines Kinematics features supported by the hardware.

Data fields

Type	Field	Description
bool	has_3_axis_↩ gyro	Specifies whether the hardware supports 3-axis gyro. **Supported values:** • 0 – Not supported • 1 – Supported
bool	has_3_axis_↩ accelerometer	Specifies whether the hardware supports the 3-axis accelerometer. **Supported values:** • 0 – Not supported • 1 – Supported
bool	has_imu_↩ supplemented↩ _dead_↩ reckoning	Specifies whether a dead reckoning (DR) solution is available and enabled or only GNSS is the result. **Supported values:** • 0 – GNSS is available • 1 – DR is available
bool	has_yaw_rate↩ _sensor	Specifies whether the IMU includes a yaw rate sensor. **Supported values:** • 0 – Does not include sensor • 1 – Includes sensor
bool	used_vehicle↩ _speed	Specifies whether the DR algorithm uses the vehicle speed sensor. **Supported values:** • 0 – Does not use sensor • 1 – Uses sensor
bool	used_single_↩ wheel_ticks	Specifies whether the DR algorithm uses the single wheel ticks. **Supported values:** • 0 – Does not use ticks • 1 – Uses ticks
bool	used_front_↩ differential_↩ wheel_ticks	Specifies whether the DR algorithm uses two front differential wheel ticks. **Supported values:** • 0 – Does not use ticks • 1 – Uses ticks
bool	used_rear_↩ differential_↩ wheel_ticks	Specifies whether the DR algorithm uses two rear differential wheel ticks. **Supported values:** • 0 – Does not use ticks • 1 – Uses ticks

Type	Field	Description
bool	used_vehicle↩ _dynamic_↩ model	Specifies whether the DR algorithm uses vehicle dynamic model factoring in differential ticks, steering, and so on. **Supported values:** • 0 – Does not use factoring • 1 – Uses factoring

4.20.2.5 struct v2x_rates_t

Defines the rate type.

Data fields

Type	Field	Description
uint32_t	rate_report_hz	Requested or reported current number of fixes per second.
uint32_t	offset_↩ nanoseconds	Currently unsupported.

4.20.2.6 struct v2x_kinematics_capabilities_t

Returned via v2x_kinematics_get_capabilities() for the client to discover the lower level function that this system supports.

Data fields

Type	Field	Description
v2x_↩ kinematics↩ _capabilities↩ _t_feature_↩ flags_t	feature_flags	Features supported by the API.
uint32_t	max_fix_rate↩ _supported_hz	Highest rate platform that supports GNSS fix generation (in Hz).
v2x_gnss_↩ fix_rates_↩ supported_↩ list_t	rates_list	Supported fix rates.

4.20.2.7 struct v2x_location_fix_t

Contains a standardized set of parameters that are used for ITS applications.

The contents of this structure do not include every possible GNSS element (for example, raw range data is not included). These fields are via low latency for safety applications, both to use locally and to load into a J2945/1 or ETSI G5 EN302.637-2 CAM. For example, the fields are used for CAM-type and BSM-type safety beacons.

This structure is populated for each location or dead-reckoning fix, and it is supplied on the fix available callback.

Predefined J2735s can be used to communicate raw SV observations and RTK correction data. Currently, however, they are not supplied from this structure.

Data fields

Type	Field	Description
double	utc_fix_time	UTC time in seconds.
v2x_fix_mode_t	fix_mode	Location engine used to produce this record. **Supported values:** No fix, 2D, 3D, RTCM
double	latitude	Latitude in degrees.
double	longitude	Longitude in degrees.
double	altitude	Altitude in meters above the geoid (mean sea level).
uint32_t	qty_SV_in_view	Number of usable space vehicles (SV) that should be in view.
uint32_t	qty_SV_used	Actual number of SVs used in this fix calculation.
v2x_GNSSstatus_t	gnss_status	Status of the GNSS data.
bool	has_SemiMajorAxisAccuracy	Specifies whether the value of the SemiMajorAxisAccuracy field is valid. **Supported values:** • 0 – Not valid • 1 – Valid
double	SemiMajorAxisAccuracy	Accuracy of the major axis, in meters.
bool	has_SemiMinorAxisAccuracy	Specifies whether the value of the SemiMinorAxisAccuracy field is valid. **Supported values:** • 0 – Not valid • 1 – Valid
double	SemiMinorAxisAccuracy	Accuracy of the minor axis, in meters.
bool	has_SemiMajorAxisOrientation	Specifies whether the value of the SemiMajorAxisOrientation field is valid. **Supported values:** • 0 – Not valid • 1 – Valid

Type	Field	Description
double	SemiMajor↩ AxisOrientation	Orientation of the major axis, in meters.
bool	has_heading	Specifies whether the value of the heading field is valid. **Supported values:** • 0 – Not valid • 1 – Valid
double	heading	Track degrees relative to true north.
bool	has_velocity	Specifies whether the value of the velocity field is valid. **Supported values:** • 0 – Not valid • 1 – Valid
double	velocity	Speed over ground in meters/second.
bool	has_climb	Specifies whether the value of the climb field is valid. **Supported values:** • 0 – Not valid • 1 – Valid
double	climb	Vertical speed in meters/second.
bool	has_lateral_↩ acceleration	Specifies whether the value of the lateral_acceleration field is valid. **Supported values:** • 0 – Not valid • 1 – Valid
double	lateral_↩ acceleration	Acceleration in a latitudinal direction, in meters/second^2.
bool	has_↩ longitudinal_↩ acceleration	Specifies whether the value of the longitudinal acceleration field is valid. **Supported values:** • 0 – Not valid • 1 – Valid
double	longitudinal_↩ acceleration	Acceleration in a longitudinal direction, in meters/second^2.
bool	has_vehicle_↩ vertical_↩ acceleration	Specifies whether the value of the vehicle_vertical_acceleration field is valid. **Supported values:** • 0 – Not valid • 1 – Valid
double	vehicle_↩ vertical_↩ acceleration	Vertical acceleration of the vehicle in G force.
bool	has_yaw_rate↩ _degrees_per↩ _second	Specifies whether the value of the yaw_rate_degrees_per_second field is valid. **Supported values:** • 0 – Not valid • 1 – Valid

Type	Field	Description
double	yaw_rate_↩ degrees_per_↩ second	Yaw rate in degrees/second, per SAE J2735.
bool	has_yaw_rate↩ _95pct_↩ confidence	Specifies whether the value of the yaw_rate_95pct_confidence field is valid. **Supported values:** • 0 – Not valid • 1 – Valid
double	yaw_rate_↩ 95pct_↩ confidence	95% confidence (2 sigma) on the yaw rate in degrees/second.
bool	has_lane_↩ position_↩ number	Specifies whether the value of the lane_position_number field is valid. **Supported values:** • 0 – Not valid • 1 – Valid
double	lane_position↩ _number	Current lane number, where 0 is either the outer-most edge of the hard shoulder or off-road.
bool	has_lane_↩ position_↩ 95pct_↩ confidence	Specifies whether the value of the lane_position_95pct_confidence field is valid. **Supported values:** • 0 – Not valid • 1 – Valid
double	lane_position↩ _95pct_↩ confidence	95% confidence range on the lane position.
bool	has_time_↩ confidence	Specifies whether the value of the time_confidence field is valid. **Supported values:** • 0 – Not valid • 1 – Valid
float	time_↩ confidence	95% (2 sigma) confidence in number of seconds.
bool	has_heading_↩ confidence	Specifies whether the value of the heading_confidence field is valid. **Supported values:** • 0 – Not valid • 1 – Valid
float	heading_↩ confidence	95% heading confidence in degrees.
bool	has_velocity_↩ confidence	Specifies whether the value of the velocity_confidence field is valid. **Supported values:** • 0 – Not valid • 1 – Valid
float	velocity_↩ confidence	95% velocity confidence in meters/second.

Type	Field	Description
bool	has_elevation↩ _confidence	Specifies whether the value of the elevation_confidence field is valid. **Supported values:** • 0 – Not valid • 1 – Valid
float	elevation_↩ confidence	95% uncertainty range (2 sigma) confidence in meters.
uint32_t	leap_seconds	Indicates that both UTC and GPS are required because IEEE 1609.2 security requires operations to be performed in raw GPS time.

4.20.3 Enumeration Type Documentation

4.20.3.1 enum v2x_fix_mode_t

Valid GNSS fix modes.

Enumerator

V2X_GNSS_MODE_NOT_SEEN SV is unavailable or not in view.
V2X_GNSS_MODE_NO_FIX No SV fix.
V2X_GNSS_MODE_2D 2D fix with latitude and longitude information.
V2X_GNSS_MODE_3D 3D fix with latitude, longitude, and altitude information.

4.20.4 Function Documentation

4.20.4.1 v2x_api_ver_t v2x_kinematics_api_version (void)

Gets the compiled API version interface (as an integer number).

Returns

v2x_api_ver_t – Filled with the version number, build date, and detailed build information.

4.20.4.2 v2x_kinematics_handle_t v2x_kinematics_init (v2x_init_t * *param*, v2x_kinematics_init_callback_t *cb*, void * *context*)

Initializes the Kinematics library.

Associated data types

v2x_init_t
v2x_kinematics_init_callback_t

Parameters

in	*param*	Pointer to the structure that contains parameters for the IP address of the server, logging level, and so on.
in	*cb*	Callback function called when initialization is complete.
in	*context*	Pointer to the application context for use with the callbacks, which can help the caller code.

Returns

Handle number to use with subsequent calls.

V2X_KINEMATICS_HANDLE_BAD – Upon an error.

4.20.4.3 v2x_status_enum_type v2x_kinematics_start_rate_notification (v2x_↩ kinematics_handle_t *handle,* v2x_kinematics_rate_notification_listener_t *cb,* void ∗ *context*)

Gets the current rate and offset from the Kinematics library.

Associated data types

v2x_kinematics_handle_t
v2x_kinematics_rate_notification_listener_t

Parameters

in	*handle*	Handle number to use with subsequent calls. If there is an error in initialization, the value is -1.
in	*cb*	Callback function used to report rate notification changes.
in	*context*	Pointer to the application context for use with the callbacks, which can help the caller code.

Returns

Indication of success or failure from v2x_status_enum_type.

4.20.4.4 v2x_status_enum_type v2x_kinematics_set_rate (v2x_kinematics_handle_t *handle*, v2x_rates_t * *rate*, v2x_kinematics_set_rate_callback_t *cb*, void * *context*)

Sets the current rate and offset from the Kinematics library.

Associated data types

v2x_kinematics_handle_t
v2x_rates_t
v2x_kinematics_set_rate_callback_t

Parameters

in	*handle*	Handle number to use with subsequent calls. If there is an error in initialization, the value is -1.
in	*rate*	Pointer to the rate structure filled with the fix timing parameters.
in	*cb*	Callback function called when rates and offsets are successfully set. This parameter can be NULL.
in	*context*	Pointer to the application context for use with the callbacks, which can help the caller code.

Returns

Indication of success or failure from v2x_status_enum_type.

4.20.4.5 v2x_status_enum_type v2x_kinematics_register_listener (v2x_kinematics_handle_t *handle,* v2x_kinematics_newfix_listener_t *listener,* void * *context*)

Registers for a Kinematics result listener callback at the requested rate.

Associated data types

v2x_kinematics_handle_t
v2x_kinematics_newfix_listener_t

Parameters

in	*handle*	Handle number to use with subsequent calls. If there is an error in initialization, the value is -1.
in	*listener*	Callback function to use for this listener.
in	*context*	Pointer to the application context for use with the callbacks, which can help the caller code.

Detailed description

This function requests GNSS fix/motion callbacks at a specified rate (Hz) with a specified offset.

Only certain rates are supported (such as 1 Hz, 2 Hz, 5 Hz, 10 Hz), which are obtained from v2x_kinematics_get_capabilities().

Currently, a request cannot be made for a rate slower than 1 Hz.

Returns

Indication of success or failure from v2x_status_enum_type.

4.20.4.6 v2x_status_enum_type v2x_kinematics_deregister_listener (v2x_kinematics_handle_t *handle,* v2x_kinematics_deregister_callback_t *cb,* void * *context*)

Deregisters a previously registered GNSS fix that the listener established earlier via v2x_kinematics_register_listener().

Associated data types

v2x_kinematics_handle_t
v2x_kinematics_deregister_callback_t

Parameters

in	*handle*	Handle number of the registered fix.
in	*cb*	Callback function to use for this listener. This parameter can be NULL.
in	*context*	Pointer to the application context for use with the callbacks, which can help the caller code.

Returns

Indication of success or failure from v2x_status_enum_type.

4.20.4.7 v2x_status_enum_type v2x_kinematics_final (v2x_kinematics_handle_t *handle,* v2x_kinematics_final_callback_t *cb,* void * *context*)

Terminates the Kinematics library.

Associated data types

v2x_kinematics_handle_t
v2x_kinematics_final_callback_t

Parameters

in	*handle*	Handle number of the library.
in	*cb*	Callback function called when termination is complete. This parameter can be NULL.
in	*context*	Pointer to the application context for use with the callbacks, which can help the caller code.

Returns

Indication of success or failure from v2x_status_enum_type.

4.20.4.8 void v2x_kinematics_enable_fixes (v2x_kinematics_handle_t *handle*)

Enables the Kinematics fixes from GNSS.

Associated data types

v2x_kinematics_handle_t

Parameters

in	*handle*	Unique identifier for the library.

Returns

None.

4.20.4.9 void v2x_kinematics_disable_fixes (v2x_kinematics_handle_t *handle*)

Disables the Kinematics fixes from GNSS.

Associated data types

v2x_kinematics_handle_t

Parameters

in	*handle*	Unique identifier for the library.

Returns

None.

4.21 C Radio APIs

This section contains C Radio APIs related to Cellular-V2X operation. Applications need to have "radio" Linux group permissions to be able to operate successfully with underlying services. For any new CV2x development, it is recommended to use the C++ telux::cv2x::∗ APIs.

Abstraction of the radio driver parameters for a V2X broadcast socket interface, including 3GPP CV2X QoS bandwidth contracts.

4.21.1 Define Documentation

4.21.1.1 #define V2X_RADIO_HANDLE_BAD (-1)

Invalid handle returned by v2x_radio_init() and v2x_radio_init_v2() upon an error.

4.21.1.2 #define V2X_MAX_RADIO_SESSIONS (10)

Limit on the number of simultaneous RmNet Radio interfaces this library can have open at once.

Typically, there are only a few actual radios. On the same radio however, one interface can be for IP traffic, and another interface can be for non-IP traffic.

4.21.1.3 #define V2X_RX_WILDCARD_PORTNUM (9000)

Wildcard value for a port number. When the wildcard is used, all V2X received traffic is routed.

4.21.1.4 #define MAX_POOL_IDS_LIST_LEN (20)

Maximum length of the pool ID list that is returned in v2x_iface_capabilities_t.

4.21.1.5 #define MAX_MALICIOUS_IDS_LIST_LEN (50)

Maximum length of the malicious ID list that can be passed in v2x_radio_update_trusted_ue_list().

4.21.1.6 #define MAX_TRUSTED_IDS_LIST_LEN (50)

Maximum length of the trusted ID list that can be passed in v2x_radio_update_trusted_ue_list().

4.21.1.7 #define MAX_SUBSCRIBE_SIDS_LIST_LEN (10)

Maximum length for the subscribed service ID list that can be passed in v2x_radio_rx_sock_create_and_bind_v2().

4.21.1.8 #define MAX_FILTER_IDS_LIST_LEN (50)

Maximum length for the L2 ID list that can be passed in v2x_set_l2_filters() and v2x_cancel_l2_filters.

4.21.1.9 #define V2X_MAX_ANTENNAS_SUPPORTED (2)

Maximum number of antennas that is supported. Used in v2x_tx_status_report_t

4.21.1.10 #define V2X_MAX_TX_POOL_NUM (2)

Maximum number of V2X Tx pools that is supported. Used in v2x_radio_status_ex_t

4.21.1.11 #define V2X_MAX_RX_POOL_NUM (4)

Maximum number of V2X Rx pools that is supported. Used in v2x_radio_status_ex_t

4.21.1.12 #define V2X_MAX_SLSS_SYNC_REF_UE_NUM (16)

Maximum number of detected SLSS sync reference UEs. Used in v2x_slss_rx_info_t

4.21.2 Data Structure Documentation

4.21.2.1 struct v2x_status_info_t

Encapsulates CV2X Tx/Rx status and cause of failure.

Data fields

Type	Field	Description
v2x_radio_↩ status_type_t	status	Tx/Rx status
v2x_radio_↩ cause_type_t	cause	Cause of failure

4.21.2.2 struct v2x_radio_status_t

Encapsulates status of CV2X radio.

Data fields

Type	Field	Description
v2x_status_↩ info_t	tx_status	TX status
v2x_status_↩ info_t	rx_status	RX status

4.21.2.3 struct v2x_pool_status_t

Encapsulates status for single TX/RX pool.

Data fields

Type	Field	Description
uint8_t	pool_id	pool ID
v2x_status_↩ info_t	status	Tx/Rx pool status

4.21.2.4 struct v2x_radio_status_ex_t

V2X overall radio status and per pool status.

Data fields

Type	Field	Description
v2x_radio_↩ status_t	status	CV2X overall TX/RX status
uint8_t	tx_pool_size	Number of Tx pools in array of pool_status.
v2x_pool_↩ status_t	tx_pool_↩ status[V2X_↩ MAX_TX_P↩ OOL_NUM]	CV2X Tx pool status.
uint8_t	rx_pool_size	Number of Rx pools in array of pool_status.
v2x_pool_↩ status_t	rx_pool_↩ status[V2X_↩ MAX_RX_P↩ OOL_NUM]	CV2X Rx pool status.

4.21.2.5 struct trusted_ue_info_t

Contains time confidence, position confidence, and propagation delay for a trusted UE.

Data fields

Type	Field	Description
uint32_t	source_l2_id	L2 ID of the trusted source
float	time_↩ uncertainty	Time uncertainty in milliseconds.
uint16_t	time_↩ confidence_↩ level	Deprecated. Use time_uncertainty instead. Confidence level of the time period. **Supported values:** 0 through 127, where 0 is invalid or unavailable and 127 is the most confident
uint16_t	position_↩ confidence_↩ level	Confidence level of the position. **Supported values:** 0 through 127, where 0 is invalid or unavailable and 127 is the most confident
uint32_t	propagation_↩ delay	Propagation delay in microseconds.

4.21.2.6 struct tx_pool_id_info_t

Contains minimum and maximum EARFCNs for a Tx pool ID. Multiple Tx Pools allow the same radio and overall frequency range to be shared for multiple types of traffic like V2V and V2X. Each pool ID and frequency range corresponds to a certain type of traffic. Both edge guard bands are not included in the EARFCN range reported. The calculation for the full bandwidth includes both edge guard bands is: bandwidth(MHz) = (max_freq-min_freq)/9. This struct is used in v2x_iface_capabilities_t.

Data fields

Type	Field	Description
uint8_t	pool_id	ID of the Tx pool.
uint16_t	min_freq	Minimum EARFCN of this pool.
uint16_t	max_freq	Maximum EARFCN of this pool.

4.21.2.7 struct v2x_iface_capabilities_t

Contains information on the capabilities of a Radio interface.

Data fields

Type	Field	Description
int	link_ip_MTU_bytes	Maximum data payload length (in bytes) of a packet supported by the IP Radio interface.
int	link_non_ip_MTU_bytes	Maximum data payload length (in bytes) of a packet supported by the non-IP Radio interface.
v2x_concurrency_sel_t	max_supported_concurrency	Indicates whether this interface supports concurrent WWAN with V2X (PC5).
uint16_t	non_ip_tx_payload_offset_bytes	Byte offset in a non-IP Tx packet before the actual payload begins. In 3GPP CV2X, the first byte after the offset is the 1-byte V2X Family ID. This offset is to the left for a per-packet Tx header that includes Tx information that might be inserted in front of the packet payload (in subsequent releases). An example of Tx information is MAC/Phy parameters (power, rate, retransmissions policy, and so on). Currently, this value is expected to be 0. But it is reserved to support possible per-packet Tx/Rx headers that might be added in future releases of this API.
uint16_t	non_ip_rx_payload_offset_bytes	Byte offset in a non-IP Rx packet before the actual payload begins. Initially, this value is zero. But it allows for later insertion of per-packet Rx information (sometimes called metadata) to be added to the front of the data payload. An example of Rx information is MAC/Phy measurements (receive signal strength, timestamps, and so on). The V2X Family ID is considered as part of the payload in the 3GPP CV2X. Higher layers (applications that are clients to this API) must remove or advance past that 1 byte to get to the more familiar actual WSMP/Geonetworking payload.
uint16_t	int_min_periodicity_multiplier_ms	Lowest number of milliseconds requested for a bandwidth. This value is also the basis for all possible bandwidth reservation periods. For example, if this multiplier=100 ms, applications can only reserve bandwidths of 100 ms, 200 ms, up to 1000 ms.
uint16_t	int_maximum_periodicity_ms	Least frequent bandwidth periodicity that is supported. Above this value, use event-driven periodic messages of a period larger than this value.
unsigned	supports_10ms_periodicity: 1	Indicates whether n∗10 ms periodicities are supported. **Supported values:** • 0 – Not supported • 1 – Supported

Type	Field	Description
unsigned	supports_↩ 20ms_↩ periodicity: 1	Indicates whether an n∗20 ms bandwidth reservation is supported. **Supported values:** • 0 – Not supported • 1 – Supported
unsigned	supports_↩ 50ms_↩ periodicity: 1	Indicates whether 50 ms periodicity is supported. **Supported values:** • 0 – Not supported • 1 – Supported
unsigned	supports_↩ 100ms_↩ periodicity: 1	Indicates whether the basic minimum periodicity of 100 ms is supported. **Supported values:** • 0 – Not supported • 1 – Supported
unsigned	max_quantity↩ _of_auto_↩ retrans: 4	Maximum number automatic retransmissions. **Supported values:** 0 through 15
unsigned	size_of_↩ layer2_mac_↩ address: 4	Size of the L2 MAC address. Different Radio Access Technologies have different-sized L2 MAC addresses: 802.11 has 6 bytes, whereas 3GPP PC5 has only 3 bytes. Because a randomized MAC address comes from an HSM with good pseudo random entropy, higher layers must know how many bytes of the MAC address to generate.
uint16_t	v2x_number_↩ of_priority_↩ levels	Number of different priority levels supported. For example, 8 is the current 3GPP standard (where a lower number means a higher priority).
uint16_t	highest_↩ priority_value	Least urgent priority number supported by this radio. Higher numbers are lower priority, so if the full range is supported, this value is V2X_PRIO_BACKGROUND.
uint16_t	lowest_↩ priority_value	Highest priority value (most urgent traffic). Lower numbers are higher priority, so if the highest level supported this value is V2X_PRIO_MOST_URGENT.
uint16_t	max_qty_SP↩ S_flows	Maximum number of supported SPS reservations.
uint16_t	max_qty_non↩ _SPS_flows	Maximum number of supported event flows (non-SPS ports).
int32_t	max_tx_pwr	Maximum supported transmission power in dBm.
int32_t	min_tx_pwr	Minimum supported transmission power in dBm.
uint32_t	tx_pool_ids_↩ supported_len	Length of the tx_pool_ids_supported array.
tx_pool_id_↩ info_t	tx_pool_ids_↩ supported[M↩ AX_POOL_I↩ DS_LIST_L↩ EN]	Array of Tx pool IDs and their associated minimum and maximum frequencies.

4.21.2.8 struct v2x_tx_bandwidth_reservation_t

Used when requesting a QoS bandwidth contract, which is implemented in PC5 3GPP V2-X radio as a *Semi Persistent Flow* (SPS).

The underlying radio providing the interface might support periodicities of various granularity in 100 ms integer multiples (such as 200 ms, 300 ms, and 400 ms).

The reservation is also used internally as a handle.

Data fields

Type	Field	Description
int	v2xid	Variable length 4-byte PSID or ITS_AID, or another application ID.
v2x_priority_et	priority	Specifies one of the 3GPP levels of priority for the traffic that is pre-reserved on the SPS flow. Use v2x_radio_query_capabilities() to get the exact number of supported priority levels.
int	period_↩ interval_ms	Bandwidth-reserved periodicity interval in milliseconds. There are limits on which intervals the underlying radio supports. Use the capabilities query method to discover the int_min_periodicity_multiplier_ms and int_maximum_periodicity_ms supported intervals.
int	tx_↩ reservation_↩ size_bytes	Number of Tx bandwidth bytes that are sent every periodicity interval.

4.21.2.9 struct v2x_chan_meas_params_t

Contains the measurement parameters for configuring the MAC/Phy radio channel measurements (such as CBR utilization).

The radio chip contains requests on radio measurement parameters that API clients can use to specify the following:

- How their higher-level application requires the CBR/CBP to be measured
- Over which time window
- When to send a report

Data fields

Type	Field	Description
int	channel_↩ measurement↩ _interval_us	Duration in microseconds of the sliding window size.
int	rs_threshold_↩ decidbm	Parameter to the radio CBR measurement that is used for determining how busy the channel is. Signals weaker than the specified receive strength (RSRP, or RSSI) are not considered to be in use (busy).

4.21.2.10 struct v2x_chan_measurements_t

Periodically returned by the radio with all measurements about the radio channel, such as the amount of noise and bandwidth saturation (channel_busy_percentage, or CBR).

Data fields

Type	Field	Description
float	channel_busy↩ _percentage	No measurement parameters are supplied.
float	noise_floor	Measurement of the background noise for a quiet channel.
float	time_↩ uncertainty	V2X time uncertainty in milliseconds.

4.21.2.11 struct v2x_radio_calls_t

Contains callback functions used in a v2x_radio_init() and v2x_radio_init_v2 call.

The radio interface uses these callback functions for events such as completion of initialization, a Layer-02 MAC address change, or a status event (loss of sufficient GPS time precision to transmit).

These callbacks are related to a specific radio interface, and its MAC/Phy parameters, such as transmit power, bandwidth utilization, and changes in radio status.

Data Fields

- void(∗ v2x_radio_init_complete)(v2x_status_enum_type status, void ∗context)
- void(∗ v2x_radio_status_listener)(v2x_event_t event, void ∗context)
- void(∗ v2x_radio_chan_meas_listener)(v2x_chan_measurements_t ∗measurements, void ∗context)
- void(∗ v2x_radio_l2_addr_changed_listener)(int new_l2_address, void ∗context)
- void(∗ v2x_radio_macphy_change_complete_cb)(void ∗context)
- void(∗ v2x_radio_capabilities_listener)(v2x_iface_capabilities_t ∗caps, void ∗context)
- void(∗ v2x_service_status_listener)(v2x_service_status_t status, void ∗context)

4.21.2.11.1 Field Documentation

4.21.2.11.1.1 void(∗ v2x_radio_calls_t::v2x_radio_init_complete) (v2x_status_enum_type status, void ∗context)

Callback that indicates initialization is complete.

Associated data types

v2x_status_enum_type

Parameters

in	*status*	Updated current radio status that indicates whether transmit and receive are ready.
in	*context*	Pointer to the context that was supplied during initial registration.

4.21.2.11.1.2 void(∗ v2x_radio_calls_t::v2x_radio_status_listener) (v2x_event_t event, void ∗context)

Callback made when the status in the radio changes. For example, in response to a fault when there is a loss of GPS timing accuracy.

Deprecated

This callback is deprecated, please consider use v2x_ext_radio_status_listener instead.

Associated data types

v2x_event_t

Parameters

in	*event*	Delivery of the event that just occurred, such losing the ability to transmit.
in	*context*	Pointer to the context of the caller who originally registered for this callback.

4.21.2.11.1.3 void(∗ v2x_radio_calls_t::v2x_radio_chan_meas_listener) (v2x_chan_measurements_t ∗measurements, void ∗context)

Callback made from lower layers when periodic radio measurements are prepared.

Associated data types

v2x_chan_measurements_t

Parameters

in	*measurements*	Pointer to the periodic measurements.
in	*context*	Pointer to the context of the caller who originally registered for this callback.

4.21.2.11.1.4 void(∗ v2x_radio_calls_t::v2x_radio_l2_addr_changed_listener) (int new_l2_address, void ∗context)

Callback made by the platform SDK when the MAC address (L2 SRC address) changes.

Parameters

in	*new_l2_address*	New L2 source address as an integer (because the L2 address is 3 bytes).
in	*context*	Pointer to the context of the caller who originally registered for this callback.

4.21.2.11.1.5 void(∗ v2x_radio_calls_t::v2x_radio_macphy_change_complete_cb) (void ∗context)

Callback made to indicate that the requested radio MAC/Phy change (such as channel/frequency and power) has completed.

Parameters

in	*context*	Pointer to the context of the caller who originally registered for this callback.

4.21.2.11.1.6 void(∗ v2x_radio_calls_t::v2x_radio_capabilities_listener) (v2x_iface_capabilities_t ∗caps, void ∗context)

Callback made when V2X capabilities change.

Associated data types

v2x_iface_capabilities_t

Parameters

in	*caps*	Pointer to the capabilities of this interface.
in	*context*	Pointer to the context of the caller who originally registered for this callback.

4.21.2.11.1.7 void(∗ v2x_radio_calls_t::v2x_service_status_listener) (v2x_service_status_t status, void ∗context)

Callback made when the service status changes.

Associated data types

v2x_service_status_t

Parameters

in	*status*	Service status.
in	*context*	Pointer to the context of the caller who originally registered for this callback.

4.21.2.12 struct v2x_sps_mac_details_t

Contains MAC information that is reported from the actual MAC SPS in the radio. The offsets can periodically change on any given transmission report.

Data fields

Type	Field	Description
uint32_t	periodicity_↩ in_use_ns	Actual transmission interval period (in nanoseconds) scheduled relative to 1PP 0:00.00 time.
uint16_t	currently_↩ reserved_↩ periodic_bytes	Actual number of bytes currently reserved at the MAC layer. This number can be slightly larger than original request.
uint32_t	tx_↩ reservation_↩ offset_ns	Actual offset, from a 1PPS pulse and Tx flow periodicity, that the MAC selected and is using for the transmit reservation. If data goes to the radio with enough time, it can be transmitted on the medium in the next immediately scheduled slot.
uint64_t	utc_time_ns	Absolute UTC start time of next selected grant, in nanoseconds.

4.21.2.13 struct v2x_per_sps_reservation_calls_t

Callback functions used in v2x_radio_tx_sps_sock_create_and_bind() calls.

Data Fields

- void(∗ v2x_radio_l2_reservation_change_complete_cb)(void ∗context, v2x_sps_mac_details_t ∗details)
- void(∗ v2x_radio_sps_offset_changed)(void ∗context, v2x_sps_mac_details_t ∗details)

4.21.2.13.1 Field Documentation

4.21.2.13.1.1 void(∗ v2x_per_sps_reservation_calls_t::v2x_radio_l2_reservation_change_complete_cb) (void ∗context, v2x_sps_mac_details_t ∗details)

Callback made upon completion of a reservation change that a v2x_radio_tx_reservation_change() call initiated for a MAC/Phy contention.

The current SPS offset and reservation parameter are passed in the details structure returned by the pointer details.

Associated data types

v2x_sps_mac_details_t

Parameters

in	*context*	Pointer to the application context.
in	*details*	Pointer to the MAC information.

4.21.2.13.1.2 void(∗ v2x_per_sps_reservation_calls_t::v2x_radio_sps_offset_changed) (void ∗context, v2x_sps_mac_details_t ∗details)

Callback periodically made when the MAC SPS timeslot changes. The new reservation offset is in the details structure returned by pointer details.

Associated data types

v2x_sps_mac_details_t

Parameters

in	*measurements*	Pointer to the channel measurements.
in	*context*	Pointer to the context.

Detailed description

This callback can occur when a MAC contention triggers a new reservation time slot to be selected. It is relevant only to connections opened with v2x_radio_tx_sps_sock_create_and_bind().

4.21.2.14 struct v2x_slss_sync_ref_ue_info_t

Encapsulates parameters of an SLSS sync reference UE. Used in v2x_slss_rx_info_t.

Data fields

Type	Field	Description
uint16_t	slss_id	The SLSS ID of the sync reference UE that is defined in 3GPP TS 36.331 chapter 6.3.8.
bool	in_coverage	Indicates whether or not the UE is in coverage of GNSS that is defined in 3GPP TS 36.331 chapter 6.5.2.
v2x_slss_↩ sync_pattern↩ _t	pattern	Indicates the SLSS sync pattern of the UE that is defined in 3GPP TS 36.331 chapter 6.3.8.
uint8_t	rsrp	SLSS RSRP value of the UE in dBm is ((float)rsrp - 256)/2.
bool	selected	Indicates whether or not the sync reference UE has been selected as the timing source.

4.21.2.15 struct v2x_slss_rx_info_t

Encapsulates parameters of CV2X SLSS Rx Information.

Used in v2x_get_slss_rx_info and v2x_slss_rx_info_listener.

Data fields

Type	Field	Description
uint32_t	num_ue	The number of SLSS sync reference UEs in array ueInfo.
v2x_slss_↩ sync_ref_ue_↩ info_t	ue_info[V2X↩ _MAX_SLS↩ S_SYNC_RE↩ F_UE_NUM]	Array of detected SLSS sync reference UEs.

4.21.2.16 struct v2x_tx_flow_info_t

Advanced parameters that can be specified for Tx SPS and event-driven flows.

Data fields

Type	Field	Description
v2x_auto_↩ retransmit_↩ policy_t	retransmit_↩ policy	V2X retransmit policy.
uint8_t	default_tx_↩ power_valid	Indicates whether the default Tx power is specified. **Supported values:** • 0 – Default power is not specified • 1 – Default power is specified and is valid
int32_t	default_tx_↩ power	Default power used for transmission.

Type	Field	Description
uint8_t	mcs_index_↩ valid	Indicates whether the MCS index is specified. **Supported values:** • 0 – Index is not specified • 1 – Index is specified and is valid
uint8_t	mcs_index	MCS index number
uint8_t	tx_pool_id_↩ valid	Indicates whether the Tx pool ID is valid. **Supported values:** • 0 – ID is not specified • 1 – ID is specified and is valid
uint8_t	tx_pool_id	ID of the Tx pool.
uint8_t	is_unicast_↩ valid	Indicates whether is_unicast is specified. **Supported values:** • 0 – Is unicast is not specified • 1 – Is unicast is specified and is valid
uint8_t	is_unicast	Non zero if requested flow is unicast. Note: Unicast flows ignore subscribed Service Ids

4.21.2.17 struct v2x_sock_info_t

Parameters to identify a Tx or Rx socket.

Data fields

Type	Field	Description
int	sock	Pointer to the file descriptor for the socket.
struct sockaddr_↩ in6	sockaddr	IPv6 socket address. The sockaddr_in6 buffer is initialized with the IPv6 source address and source port that are used for the bind() function.

4.21.2.18 struct v2x_sid_list_t

Parameters to identify a service ID list.

Data fields

Type	Field	Description
int	length	number of services IDs included in the array of sid.
uint32_t	sid[MAX_S↩ UBSCRIBE_↩ SIDS_LIST_↩ LEN]	array of service IDs.

4.21.2.19 struct v2x_tx_sps_flow_info_t

Advanced parameters that can be specified for Tx SPS flows.

Data fields

Type	Field	Description
v2x_tx_↩ bandwidth↩ _reservation_t	reservation	Transmit reservation information.
v2x_tx_flow_↩ info_t	flow_info	Transmit resource information about the SPS Tx flow.

4.21.2.20 struct socket_info_t

Parameters that can be specified for the creation of CV2X socket.

Data fields

Type	Field	Description
uint32_t	service_id	V2X service ID bound to the CV2X socket.
uint16_t	local_port	Local port number of the CV2X socket used for binding.

4.21.2.21 struct src_l2_filter_info_t

Contains remote UE source L2 ID that expecting to filter.

Data fields

Type	Field	Description
uint32_t	src_l2_id	< remote UE L2 addr to filter. Duration, in millisec (resolution 100 msec).
uint32_t	duration_ms	/∗ Proximity service per packet priority (PPPP), packets with priority above this value will be dropped. Range 0-7, 0 mean all of the pkts will be dropped.
uint8_t	pppp	

4.21.2.22 struct v2x_rf_tx_info_t

Tx status per Tx chain and Tx power per Tx antenna for a specific transport block.

Data fields

Type	Field	Description
rf_status_t	status	Fault detection status for a specific Tx chain.
int32_t	power	The target Tx power after MPR/AMPR reduction for a specific Tx antenna in dBm∗10 format. Invalid value is -700, it means the corresponding antenna is not being used for the transmission of this transport block.

4.21.2.23 struct v2x_tx_status_report_t

Information on Tx status of a V2X transport block that is reported from low layer.

1. A V2X Tx packet might trigger multiple reports because of the segmentaion and re-Tx in low layer.
2. If a transport block is dropped in low layer, no report will be triggered for that transport block.
3. The power in the array of rfInfo is the target Tx power value in dBm∗10 after MPR/AMPR reduction for a specific Tx antenna. The status in the array of rfInfo is the fault detection status for a specific Tx chain.
   - In CDD mode, two antennas have transmission for a specific transport block, both rfInfo[0].power and rfInfo[1].power are valid (not -700), rfInfo[i].status is reflecting the status of Tx chain/Tx antenna i.
   - In TXD mode, data transmission swtiches between two antennas/chains and only one antenna/chain has transmission for a specific transport block, the Tx antenna being used has valid power (not -700) in the array of rfInfo, rfInfo[i].status is reflecting the status of Tx chain i or the status of the Tx antenna i whose power is valid (not -700) in the array of rfInfo. Used in v2x_tx_status_report_listener

Data fields

Type	Field	Description
v2x_rf_tx_↩ info_t	rf_info[V2X_↩ MAX_ANTE↩ NNAS_SUP↩ PORTED]	Tx status per Tx chain and Tx power per Tx antenna.
uint8_t	num_rb	Number of resource blocks used for the transport block.
uint8_t	start_rb	Start resource block index used for the transport block.
uint8_t	mcs	Modulation and coding scheme used for the transport block that is defined in 3GPP TS 36.213.
uint8_t	seg_num	Total number of segments of a V2X packet.
v2x_segment↩ _type_t	seg_type	Segment type of the transport block.
v2x_tx_type_t	tx_type	Indication of new Tx or re-Tx of the transport block.
uint16_t	ota_timing	OTA timing in format of system frame number∗10 + subframe number.
uint16_t	port	Port number that can be used to link the report to a specific Tx flow which has the same source port number.

4.21.3 Enumeration Type Documentation

4.21.3.1 enum v2x_concurrency_sel_t

Describes whether the radio chip modem should attempt or support concurrent 3GPP CV2X operation with a WWAN 4G/5G data call.

Some chips are only capable of operating on CV2X. In this case:

- The callers can only request V2X_WWAN_NONCONCURRENT.

- The interface parameters that are returned list v2x_concurrency_sel_t as the value for V2X_WAN_NONCONCURRENT.

Enumerator

V2X_WWAN_NONCONCURRENT No simultaneous WWAN + CV2X on this interface.
V2X_WWAN_CONCURRENT Interface allows requests for concurrent support of WWAN + CV2X connections.

4.21.3.2 enum v2x_event_t

Event indications sent asynchronously from the radio via callbacks that indicate the state of the radio. The state can change in response to the loss of timing precision or a geofencing change.

Deprecated

This enum type is deprecated, please consider use v2x_radio_status_ex_t instead.

Enumerator

V2X_INACTIVE V2X communication is disabled.
V2X_ACTIVE V2X communication is enabled. Transmit and receive are possible.
V2X_TX_SUSPENDED Small loss of timing precision occurred. Transmit is no longer supported.
V2X_RX_SUSPENDED Radio can no longer receive any messages.
V2X_TXRX_SUSPENDED Radio can no longer transmit or receive for some reason.

4.21.3.3 enum v2x_priority_et

Range of supported priority levels, where a lower number means a higher priority. For example, 8 is the current 3GPP standard.

Enumerator

V2X_PRIO_MOST_URGENT Highest priority.
V2X_PRIO_1
V2X_PRIO_2
V2X_PRIO_3
V2X_PRIO_4
V2X_PRIO_5
V2X_PRIO_6
V2X_PRIO_BACKGROUND Lowest priority.

4.21.3.4 enum v2x_service_status_t

Valid service availability states.

Enumerator

SERVICE_UNAVAILABLE
SERVICE_AVAILABLE
SERVICE_FAILED

4.21.3.5 enum v2x_radio_status_type_t

Defines possible values for CV2X radio RX/TX status.

1. If Rx is in inactive state, Tx should also be in inactive state.
2. If Rx is in active state, Tx should be in active(normal case) or suspended state(sensing or tunnel mode).
3. If Rx is in suspended state, Tx should be in suspended state. Used in v2x_status_info_t

Enumerator

V2X_RADIO_STATUS_INACTIVE RX/TX is inactive
V2X_RADIO_STATUS_ACTIVE RX/TX is active
V2X_RADIO_STATUS_SUSPENDED RX/TX is suspended
V2X_RADIO_STATUS_UNKNOWN RX/TX status unknown

4.21.3.6 enum v2x_radio_cause_type_t

Defines possible values for cause of CV2X radio failure. The cause code is only associated with cv2x suspend/inactive status, if cv2x is active, the cause code has no meaning. Used in v2x_status_info_t

Enumerator

V2X_RADIO_CAUSE_TIMING CV2X is suspended due to the outage of timing reference.
V2X_RADIO_CAUSE_CONFIG CV2X is inactive due to v2x.xml is missing, invalid, or expired.

V2X_RADIO_CAUSE_UE_MODE CV2X is inactive due to CV2X mode is not started.
V2X_RADIO_CAUSE_GEOPOLYGON CV2X is inactive due to UE enters a geo-polygon that does not support cv2x.
V2X_RADIO_CAUSE_THERMAL CV2X is suspended when the device's temperature is high.
V2X_RADIO_CAUSE_THERMAL_ECALL CV2X is suspended when the device's temperature is high and emergency call is ongoing.
V2X_RADIO_CAUSE_GEOPOLYGON_SWITCH CV2X is suspended when UE switches to a new geopolygon that also supports CV2X and UE is already in CV2X active status, CV2X status will change to active after the update is done.
V2X_RADIO_CAUSE_SENSING CV2X Tx is suspended when GNSS signal recovers or CV2X mode just starts. UE needs sensing for 1 second before Tx can begin, Tx status will change to active after sensing is done.
V2X_RADIO_CAUSE_LPM CV2X is inactive when UE enters Low Power Mode.
V2X_RADIO_CAUSE_DISABLED CV2X is inactive due to CV2X is disabled in the EFS.
V2X_RADIO_CAUSE_NO_GNSS CV2X is inactive due to GNSS signal is not available when starting CV2X.
V2X_RADIO_CAUSE_INVALID_LICENSE CV2X is inactive due to invalid license.
V2X_RADIO_CAUSE_UNKNOWN Invalid cause type only used internally.

4.21.3.7 enum v2x_auto_retransmit_policy_t

V2X Tx retransmission policies supported by the modem.

Enumerator

V2X_AUTO_RETRANSMIT_DISABLED Retransmit mode is disabled.
V2X_AUTO_RETRANSMIT_ENABLED Retransmit mode is enabled.
V2X_AUTO_RETRANSMIT_DONT_CARE Modem falls back to its default behavior.

4.21.3.8 enum v2x_slss_sync_pattern_t

Defines possible values for SLSS sync pattern. Used in v2x_slss_sync_ref_ue_info_t

Enumerator

V2X_SLSS_SYNC_PATTERN_OFFSET_IND_1 UE transmits SLSS in subframes indicated by the syncOffsetIndicator1 specified in V2X configuration.
V2X_SLSS_SYNC_PATTERN_OFFSET_IND_2 UE transmits SLSS in subframes indicated by the syncOffsetIndicator2 specified in V2X configuration.
V2X_SLSS_SYNC_PATTERN_OFFSET_IND_3 UE transmits SLSS in subframes indicated by the syncOffsetIndicator3 specified in V2X configuration.
V2X_SLSS_SYNC_PATTERN_ODD_RESERVED UE transmits SLSS in odd-numbered reserved subframes.
V2X_SLSS_SYNC_PATTERN_EVEN_RESERVED UE transmits SLSS in even-numbered reserved subframes.
V2X_SLSS_SYNC_PATTERN_UNKNOWN Unkown SLSS sync pattern.

4.21.3.9 enum traffic_ip_type_t

V2X Ip Types

Enumerator

TRAFFIC_IP Use Ip type traffic.
TRAFFIC_NON_IP Use Non-Ip type traffic.

4.21.3.10 enum rf_status_t

Fault detection for Tx chain that including PA and front end.

Enumerator

INACTIVE The Tx chain is not working.
OPERATIONAL The Tx chain is operational.
FAULT Fault detected on the Tx chain.

4.21.3.11 enum v2x_segment_type_t

Defines possible values for the segment type of a transport block.

Enumerator

FIRST V2X packet is segmented, it's the first transport block.
LAST V2X packet is segmented, it's the last transport block.
MIDDLE V2X packet is segmented, it's a transport block between first and last.
ONLY_ONE V2X packet is not segmented, it's the only one transport block.

4.21.3.12 enum v2x_tx_type_t

Defines new Tx or re-Tx type relevant to a transport block.

Enumerator

V2X_NEW_TX New Tx of the V2X transport block.
V2X_RE_TX Re-Tx of the V2X transport block.
V2X_SLSS_TX Tx of SLSS.

4.21.4 Function Documentation

4.21.4.1 uint16_t v2x_convert_priority_to_traffic_class (v2x_priority_et *priority*)

Converts a traffic priority to one of the 255 IPv6 traffic class bytes that are used in the data plane to indicate per-packet priority on non-SPS (event driven) data ports.

Associated data types

v2x_priority_et

Parameters

`in`	*priority*	Packet priority that is to be converted to an IPv6 traffic class. This priority is between the lowest and highest priority values returned in v2x_iface_capabilities_t.

Detailed description

This function is symmetric and is a reverse operation.

The traffic priority is one of the values between min_priority_value and max_priority_value returned in the v2x_iface_capabilities_t query.

Returns

IPv6 traffic class for achieving the calling input parameter priority level.

4.21.4.2 v2x_priority_et v2x_convert_traffic_class_to_priority (uint16_t *traffic_class*)

Maps an IPv6 traffic class to a V2X priority value.

Parameters

in	*traffic_class*	IPv6 traffic classification that came in a packet from the radio.

Detailed description

This function is the inverse of the v2x_convert_priority_to_traffic_class() function. It is symmetric and is a reverse operation.

Returns

Priority level (between highest and lowest priority values) equivalent to the input IPv6 traffic class parameter.

4.21.4.3 v2x_api_ver_t v2x_radio_api_version ( )

Method used to query the platform SDK for its version number, build information, and build date.

Returns

v2x_api_ver_t – Contains the build date and API version number.

4.21.4.4 v2x_status_enum_type v2x_radio_query_capabilities (v2x_iface_↩ capabilities_t * *caps*)

Gets the capabilities of CV2X radio.

Associated data types

v2x_iface_capabilities_t

Parameters

out	*caps*	Pointer to the v2x_iface_capabilities_t structure, which contains the capabilities of this specific interface.

Returns

V2X_STATUS_SUCCESS – The radio is ready for data-plane sockets to be created and bound.

Error code – If there is a problem (see v2x_status_enum_type).

4.21.4.5 v2x_status_enum_type v2x_radio_deinit (v2x_radio_handle_t *handle*)

De-initializes a specific Radio interface.

Associated data types

v2x_radio_handle_t

Parameters

in	*handle*	Handle to the Radio that was initialized.

Returns

Indication of success or failure (see v2x_status_enum_type).

Dependencies

The interface must be pre-initialized with v2x_radio_init() or v2x_radio_init_v2(). The handle from that function must be used as the parameter in this function.

4.21.4.6 int v2x_radio_rx_sock_create_and_bind (v2x_radio_handle_t *handle,* int * *sock,* struct sockaddr_in6 * *rx_sockaddr*)

Opens a new V2X radio receive socket, and initializes the given sockaddr buffer. The socket is also bound as an AF_INET6 UDP type socket.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_FLOW_OPS permission to successfully invoke this API.

Associated data types

v2x_radio_handle_t

Parameters

in	*handle*	Identifies the initialized Radio interface.
out	*sock*	Pointer to the socket that, on success, returns the socket descriptor. The caller must release this socket with v2x_radio_sock_close().
out	*rx_sockaddr*	Pointer to the IPv6 UDP socket. The sockaddr_in6 buffer is initialized with the IPv6 source address and source port that are used for the bind.

Detailed description

You can execute any sockopts that are appropriate for this type of socket (AF_INET6).

The port number for the receive path is not exposed, but it is in the sockaddr_in6 structure (if the caller is interested).

Returns

0 – On success.

Otherwise:

- EPERM – Socket creation failed; for more details, check errno.h.
- EAFNOSUPPORT – On failure to find the interface.
- EACCES – On failure to get the MAC address of the device.

Dependencies

The interface must be pre-initialized with v2x_radio_init() or v2x_radio_init_v2(). The handle from that function must be used as the parameter in this function.

4.21.4.7 int v2x_radio_rx_sock_create_and_bind_v2 (v2x_radio_handle_t *handle*, int *id_ist_len*, uint32_t * *id_list*, int * *sock*, struct sockaddr_in6 * *rx_sockaddr*)

Opens a new V2X radio receive socket with specific service IDs for subscription, and initializes the given sockaddr buffer. The socket is also bound as an AF_INET6 UDP type socket.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_FLOW_OPS permission to successfully invoke this API.

Associated data types

v2x_radio_handle_t

Parameters

in	*handle*	Identifies the initialized Radio interface.
in	*id_ist_len*	Identifies the length of service ID list.
in	*id_list*	Pointer to the service ID list for subscription, subscribe wildcard if input nullptr.
out	*sock*	Pointer to the socket that, on success, returns the socket descriptor. The caller must release this socket with v2x_radio_sock_close().
out	*rx_sockaddr*	Pointer to the IPv6 UDP socket. The sockaddr_in6 buffer is initialized with the IPv6 source address and source port that are used for the bind.

Detailed description

You can execute any sockopts that are appropriate for this type of socket (AF_INET6).

This API can be used to subscribe wildcard, catchall port, or specifc service IDs. The Rx port should be set with v2x_set_rx_port() before any subscription via this API, otherwise a default port number 9000 will be used.

Wildcard is used to receive all traffic. Only one port can be registered as wildcard port. Once wildcard is registered successfully, all received packets will be directed to wildcard port, and any subscription for specific service IDs or catchall port at other ports will be invalid. The parameter id_list of this API should be set to a null list for wildcard subscription.

Catchall port is used to receive packets with non-registered service IDs (via specific service IDs subscription). Only one port can be registered as catchall port. If catchall port is registered successfully, received packets with non-registered service ID will be directed to catchall port. All specific service IDs subscription (if any) should be performed before catchall port subscription. The parameter id_list of this API should include all non-registered service IDs for catchall port subscription.

Any port different from catchall port can be used to receive packets with specific service IDs. Only one port can be registered for a single service ID, a list of service IDs can be registered at a single port. To subscribe specific service IDs at a given Rx port, a Tx flow must be pre-setup with the Tx service ID set to any service ID included in the list of specific service IDs and the Tx source port set to the same port number as Rx port. The parameter id_list of this API should include all interested service IDs for the given Rx port.

Returns

0 – On success.

Otherwise:

- EPERM – Socket creation failed; for more details, check errno.h.
- EAFNOSUPPORT – On failure to find the interface.
- EACCES – On failure to get the MAC address of the device.

Dependencies

The interface must be pre-initialized with v2x_radio_init(). The handle from that function must be used as the parameter in this function. The Rx port must be pre-set with v2x_set_rx_port(), otherwise a default port number will be used. For any specific service ID subscription, a Tx flow must be pre-setup using one of the following methods:

- v2x_radio_tx_sps_sock_create_and_bind()
- v2x_radio_tx_sps_sock_create_and_bind_v2()
- v2x_radio_tx_sps_only_create()
- v2x_radio_tx_sps_only_create_v2()
- v2x_radio_tx_event_sock_create_and_bind()
- v2x_radio_tx_event_sock_create_and_bind_v2()
- v2x_radio_tx_event_sock_create_and_bind_v3()

4.21.4.8 int v2x_radio_rx_sock_create_and_bind_v3 (v2x_radio_handle_t *handle*, uint16_t *port_num*, int *id_ist_len*, uint32_t * *id_list*, int * *sock*, struct sockaddr_in6 * *rx_sockaddr*)

Opens a new V2X radio receive socket with specific service IDs for subscription and specifc port number for the receive path, and initializes the given sockaddr buffer. The socket is also bound as an AF_INET6 UDP type socket.

This v2x_radio_rx_sock_create_and_bind_v3() method differs from v2x_radio_rx_sock_create_and_bind_v2() in that you can use the port_num parameter to specify the port number for the receive path.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_FLOW_OPS permission to successfully invoke this API.

Associated data types

v2x_radio_handle_t

Parameters

in	*handle*	Identifies the initialized Radio interface.
in	*port_num*	Identifies the port number for the receive path.
in	*id_ist_len*	Identifies the length of service ID list.
in	*id_list*	Pointer to the service ID list for subscription, subscribe wildcard if input nullptr.
out	*sock*	Pointer to the socket that, on success, returns the socket descriptor. The caller must release this socket with v2x_radio_sock_close().
out	*rx_sockaddr*	Pointer to the IPv6 UDP socket. The sockaddr_in6 buffer is initialized with the IPv6 source address and source port that are used for the bind.

Detailed description

You can execute any sockopts that are appropriate for this type of socket (AF_INET6).

This API can be used to subscribe wildcard, catchall port, or specifc service IDs.

Wildcard is used to receive all traffic. Only one port can be registered as wildcard port. Once wildcard is registered successfully, all received packets will be directed to wildcard port, and any subscription for specific service IDs or catchall port at other ports will be invalid. The parameter id_list of this API should be set to a null list for wildcard subscription.

Catchall port is used to receive packets with non-registered service IDs (via specific service IDs subscription). Only one port can be registered as catchall port. If catchall port is registered successfully, received packets with non-registered service ID will be directed to catchall port. All specific service IDs subscription (if any) should be performed before catchall port subscription. The parameter id_list of this API should include all non-registered service IDs for catchall port subscription.

Any port different from catchall port can be used to receive packets with specific service IDs. Only one port can be registered for a single service ID, a list of service IDs can be registered at a single port. To

subscribe specific service IDs at a given Rx port, a Tx flow must be pre-setup with the Tx service ID set to any service ID included in the list of specific service IDs and the Tx source port set to the same port number as Rx port. The parameter id_list of this API should include all interested service IDs for the given Rx port.

Returns

0 – On success.

Otherwise:

- EPERM – Socket creation failed; for more details, check errno.h.
- EAFNOSUPPORT – On failure to find the interface.
- EACCES – On failure to get the MAC address of the device.

Dependencies

The interface must be pre-initialized with v2x_radio_init(). The handle from that function must be used as the parameter in this function. For any specific service ID subscription, a Tx flow must be pre-setup using one of the following methods:

- v2x_radio_tx_sps_sock_create_and_bind()
- v2x_radio_tx_sps_sock_create_and_bind_v2()
- v2x_radio_tx_sps_only_create()
- v2x_radio_tx_sps_only_create_v2()
- v2x_radio_tx_event_sock_create_and_bind()
- v2x_radio_tx_event_sock_create_and_bind_v2()
- v2x_radio_tx_event_sock_create_and_bind_v3()

4.21.4.9 int v2x_radio_enable_rx_meta_data (v2x_radio_handle_t *handle*, bool *enable*, int *id_list_len*, uint32_t * *id_list*)

Enable or disable the meta data report for the packets corresponding to the service IDs.

If enabled, the meta data report would be generated in addition to the actual OTA payload packet, and it comes from the same data interface as the OTA packet itself, it consist of RF RSSI (received signal strength indicator) status, 32-bit SCI Format 1 (3GPP TS 36.213, section 14.1), packet delay estimation, L2 destination ID, and the resource blocks used for the packet's transmission: subframe, subchannel index.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_INFO permission to successfully invoke this API.

Associated data types

v2x_radio_handle_t

Parameters

in	*handle*	Identifies the initialized Radio interface.
in	*enable*	enable or disable the meta data
in	*id_list_len*	number of the service IDs provided in the id_list
in	*id_list*	Pointer to the Rx service ID list

Detailed description

This function extracts the received packet's meta data from the payload, currently only NON-IP packets can have the meta data reported, it is not supported yet for IP packets.

If the meta data report is enabled for certain services, call v2x_parse_rx_meta_data to extract the meta data by providing a pointer to a object of type rx_packet_meta_data_t, and the real payload.

Returns

0 – On success.

4.21.4.10 int v2x_radio_sock_create_and_bind (v2x_radio_handle_t *handle*, v2x_tx_sps_flow_info_t * *tx_flow_info*, v2x_per_sps_reservation_calls_t * *calls*, int *tx_sps_portnum*, int *tx_event_portnum*, int *rx_portnum*, v2x_↩ sid_list_t * *rx_id_list*, v2x_sock_info_t * *tx_sps_sock*, v2x_sock_info_t * *tx_event_sock*, v2x_sock_info_t * *rx_sock*)

Creates Tx SPS socket, Tx Event socket and Rx socket with specified parameters. The socket is also bound as an AF_INET6 UDP type socket.

This v2x_radio_sock_create_and_bind() method is the combination of function v2x_radio_tx_sps_sock_create_and_bind_v2()/v2x_radio_tx_event_sock_create_and_bind_v2 in the transmit direction and function v2x_radio_rx_sock_create_and_bind_v3() in the receiving direction.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_FLOW_OPS permission to successfully invoke this API.

Associated data types

v2x_radio_handle_t

Parameters

in	*handle*	Identifies the initialized Radio interface.
in	*tx_flow_info*	Pointer to the Tx SPS or event flow information. To create event flow, set reservation.v2xid and flow_info in this structure.
in	*calls*	Pointer to reservation callbacks or listeners. This parameter is called when underlying radio MAC parameters change related to the SPS bandwidth contract. For example, the callback after a reservation change, or if the timing offset of the SPS adjusts itself in response to traffic. This parameter passes NULL if no callbacks are required.
in	*tx_sps_portnum*	Requested Tx source port number for SPS transmissions, or -1 for no Tx sps flow.
in	*tx_event_portnum*	Requested Tx source port number for event transmissions, or -1 for no Tx event flow.
in	*rx_portnum*	Requested Rx destination port number, or -1 for no Rx subscription.
in	*rx_id_list*	Pointer to the Rx service ID list for subscription, subscribe wildcard if input nullptr.
out	*tx_sps_sock*	Pointer to the Tx sps socket that, on success, returns the socket descriptor and the IPv6 socket address. The caller must release this socket with v2x_radio_sock_close().
out	*tx_event_sock*	Pointer to the Tx event socket that, on success, returns the socket descriptor and the IPv6 socket address. The caller must release this socket with v2x_radio_sock_close().
out	*rx_sock*	Pointer to the Rx socket that, on success, returns the socket descriptor and the IPv6 socket address. The caller must release this socket with v2x_radio_sock_close().

Detailed description

You can execute any sockopts that are appropriate for this type of socket (AF_INET6).

This API can be used for the registeration of both Tx and Rx. It sets up sockets on the requested port numbers. A negative port number corresponds to no actions for Tx or Rx.

Wildcard is used to receive all traffic. Only one port can be registered as wildcard port. Once wildcard is registered successfully, all received packets will be directed to wildcard port, and any subscription for specific service IDs or catchall port at other ports will be invalid. The parameter rx_id_list of this API should be set to a null list for wildcard subscription.

Catchall port is used to receive packets with non-registered service IDs (via specific service IDs subscription). Only one port can be registered as catchall port. If catchall port is registered successfully, received packets with non-registered service ID will be directed to catchall port. All specific service IDs subscription (if any) should be performed before catchall port subscription. The parameter rx_id_list of this API should include all non-registered service IDs for catchall port subscription.

Any port different from catchall port can be used to receive packets with specific service IDs. Only one port can be registered for a single service ID, a list of service IDs can be registered at a single port. To subscribe specific service IDs at a given Rx port, a Tx flow should also be set up using this API. The parameter rx_id_list should include all interested service IDs for the given Rx port, the parameter tx_flow_info.reservation.v2xid should be set to one of the service ID included in rx_id_list, the parmenter tx_sps_portnum or tx_event_portnum should be set to the same port number as rx_portnum.

Returns

0 – On success.

Otherwise:

- EPERM – Socket creation failed; for more details, check errno.h.
- EAFNOSUPPORT – On failure to find the interface.
- EACCES – On failure to get the MAC address of the device.

Dependencies

The interface must be pre-initialized with v2x_radio_init(). The handle from that function must be used as the parameter in this function.

4.21.4.11 int v2x_radio_tx_sps_sock_create_and_bind (v2x_radio_handle_t *handle*, v2x_tx_bandwidth_reservation_t * *res*, v2x_per_sps_reservation_calls_t * *calls*, int *sps_portnum*, int *event_portnum*, int * *sps_sock*, struct sockaddr_in6 * *sps_sockaddr*, int * *event_sock*, struct sockaddr_in6 * *event_sockaddr*)

Creates and binds a socket with a bandwidth-reserved (SPS) Tx flow with the requested ID, priority, periodicity, and size on a specified source port number. The socket is created as an IPv6 UDP socket.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_FLOW_OPS permission to successfully invoke this API.

Associated data types

v2x_radio_handle_t
v2x_tx_bandwidth_reservation_t
v2x_per_sps_reservation_calls_t

Parameters

in	*handle*	Identifies the initialized Radio interface on which this data connection is made.
in	*res*	Pointer to the parameter structure (how often the structure is sent, how many bytes are reserved, and so on).
in	*calls*	Pointer to reservation callbacks or listeners. This parameter is called when underlying radio MAC parameters change related to the SPS bandwidth contract. For example, the callback after a reservation change, or if the timing offset of the SPS adjusts itself in response to traffic. This parameter passes NULL if no callbacks are required.
in	*sps_portnum*	Requested source port number for the bandwidth reserved SPS transmissions.
in	*event_portnum*	Requested source port number for the bandwidth reserved event transmissions, or -1 for no event port.
out	*sps_sock*	Pointer to the socket that is bound to the requested port for Tx with reserved bandwidth.
out	*sps_sockaddr*	Pointer to the IPv6 UDP socket. The sockaddr_in6 buffer is initialized with the IPv6 source address and source port that are used for the bind() function. The caller can then use the buffer for subsequent sendto() function calls.
out	*event_sock*	Pointer to the socket that is bound to the event-driven transmission port.
out	*event_sockaddr*	Pointer to the IPV6 UDP socket. The sockaddr_in6 buffer is initialized with the IPv6 source address and source port that are used for the bind() function. The caller can then use the buffer for subsequent sendto() function calls.

Detailed description

The radio attempts to reserve the flow with the specified size and rate passed in the request parameters.

This function is used only for Tx. It sets up two UDP sockets on the requested two HLOS port numbers.

For only a single SPS flow, indicate the event port number by using a negative number or NULL for the event_sockaddr. For a single event-driven port, use v2x_radio_tx_event_sock_create_and_bind() instead.

Because the modem endpoint requires a specific global address, all data sent on these sockets must have a configurable IPv6 destination address for the non-IP traffic.

The Priority parameter of the SPS reservation is used only for the reserved Tx bandwidth (SPS) flow. The non-SPS/event-driven data sent to the event_portnum parameter is prioritized on the air, based on the IPv67 Traffic Class of the packet.

The caller is expected to identify two unused local port numbers to use for binding: one for the event-driven flow and one for the SPS flow.

This call is a blocking call. When it returns, the sockets are ready to use, assuming there is no error.

Returns

0 – On success.

Otherwise:

- EPERM – Socket creation failed; for more details, check errno.h.
- EAFNOSUPPORT – On failure to find the interface.
- EACCES – On failure to get the MAC address of the device.

Dependencies

The interface must be pre-initialized with v2x_radio_init() or v2x_radio_init_v2(). The handle from that function must be used as the parameter in this function.

4.21.4.12 int v2x_radio_tx_sps_only_create (v2x_radio_handle_t *handle*, v2x_tx_↩ bandwidth_reservation_t ∗ *res*, v2x_per_sps_reservation_calls_t ∗ *calls*, int *sps_portnum*, int ∗ *sps_sock*, struct sockaddr_in6 ∗ *sps_sockaddr*)

Creates a socket with a bandwidth-reserved (SPS) Tx flow.

Only SPS transmissions are to be implemented for the socket, which is created as an IPv6 UDP socket.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_FLOW_OPS permission to successfully invoke this API.

Associated data types

v2x_radio_handle_t
v2x_tx_bandwidth_reservation_t
v2x_per_sps_reservation_calls_t

Parameters

in	*handle*	Identifies the initialized Radio interface on which this data connection is made.
in	*res*	Pointer to the parameter structure (how often the structure is sent, how many bytes are reserved, and so on).
in	*calls*	Pointer to reservation callbacks or listeners. This parameter is called when underlying radio MAC parameters change related to the SPS bandwidth contract. For example, the callback after a reservation change, or if the timing offset of the SPS adjusts itself in response to traffic. This parameter passes NULL if no callbacks are required.
in	*sps_portnum*	Requested source port number for the bandwidth reserved SPS transmissions.
out	*sps_sock*	Pointer to the socket that is bound to the requested port for Tx with reserved bandwidth.
out	*sps_sockaddr*	Pointer to the IPv6 UDP socket. The sockaddr_in6 buffer is initialized with the IPv6 source address and source port that are used for the bind() function. The caller can then use the buffer for subsequent sendto() function calls.

Detailed description

The radio attempts to reserve the flow with the specified size and rate passed in the request parameters.

This function is used only for Tx. It sets up a UDP socket on the requested HLOS port number. Because the modem endpoint requires a specific global address, all data sent on the socket must have a configurable IPv6 destination address for the non-IP traffic.

The Priority parameter of the SPS reservation is used only for the reserved Tx bandwidth (SPS) flow.

The caller is expected to identify an unused local port number for the SPS flow.

This call is a blocking call. When it returns, the socket is ready to use, assuming there is no error.

Returns

0 – On success.

Otherwise:

- EPERM – Socket creation failed; for more details, check errno.h.
- EINVAL – On failure to find the interface or get bad parameters.

Dependencies

The interface must be pre-initialized with v2x_radio_init() or v2x_radio_init_v2(). The handle from that function must be used as the parameter in this function.

4.21.4.13 v2x_status_enum_type v2x_radio_tx_reservation_change (int * *sps_sock,* v2x_tx_bandwidth_reservation_t * *updated_reservation*)

Adjusts the reservation for transmit bandwidth.

Associated data types

v2x_tx_bandwidth_reservation_t

Parameters

out	*sps_sock*	Pointer to the socket bound to the requested port.
in	*updated_reservation*	Pointer to a bandwidth reservation with new reservation information.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_FLOW_OPS permission to successfully invoke this API.

Detailed description

This function will not update reservation priority. Can be used as follows:

- When the bandwidth requirement changes in periodicity (for example, due to an application layer DCC algorithm)
- Because the packet size is increasing (for example, due to a growing path history size in a BSM).

When the reservation change is complete, a callback to the structure is passed in a v2x_radio_init() or v2x_radio_init_v2() call.

Returns

V2X_STATUS_SUCCESS.

Error code – If there is a problem (see v2x_status_enum_type).

Dependencies

An SPS flow must have been successfully initialized with the v2x_radio_tx_sps_sock_create_and_bind().

4.21.4.14 int v2x_radio_tx_event_sock_create_and_bind (const char * *interface,* int *v2x_id,* int *event_portnum,* struct sockaddr_in6 * *event_sock_addr,* int * *sock*)

Opens and binds an event-driven socket (one with no bandwidth reservation). The socket is bound as an AF_INET6 UDP type socket.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_FLOW_OPS permission to successfully invoke this API.

Parameters

in	*interface*	Pointer to the operating system name to use. This interface is an RmNet interface (HLOS).
in	*v2x_id*	Used for transmissions that are ultimately mapped to an L2 destination address.
in	*event_portnum*	Local port number to which the socket is bound. Used for transmissions of this ID.
out	*event_sock_addr*	Pointer to the sockaddr_ll structure buffer to be initialized.
out	*sock*	Pointer to the file descriptor. Loaded when the function is successful.

Detailed description

This function is used only for Tx when no periodicity is available for the application type. If you know your transmit data periodicity, use v2x_radio_tx_sps_sock_create_and_bind() instead.

These event-driven sockets pay attention to QoS parameters in the IP socket.

Returns

0 – On success.

Otherwise:

- EPERM – Socket creation failed; for more details, check errno.h.
- EAFNOSUPPORT – On failure to find the interface.
- EACCES – On failure to get the MAC address of the device.

4.21.4.15 v2x_status_enum_type v2x_radio_start_measurements (v2x_radio_handle_t *handle,* v2x_chan_meas_params_t * *measure_this_way*)

Requests a channel utilization (CBP/CBR) measurement result on a channel.

Associated data types

v2x_radio_handle_t
v2x_chan_meas_params_t

Parameters

in	*handle*	Handle to the port.
in	*measure_this_way*	Indicates how and what to measure, and how often to send results. Some higher-level standards (like J2945/1 and ETSI TS102687 DCC) have specific time windows and items to measure.

Detailed description

This function uses the callbacks passed in during initialization to deliver the measurements. Measurement callbacks continue until the Radio interface is closed.

Returns

V2X_STATUS_SUCCESS – The radio is now ready for data-plane sockets to be created and bound.

V2X_STATUS_FAIL – CBR measurement is not supported yet.

Dependencies

The interface must be pre-initialized with v2x_radio_init() or v2x_radio_init_v2(). The handle from that function must be used as the parameter in this function.

4.21.4.16 v2x_status_enum_type v2x_radio_stop_measurements (v2x_radio_↩ handle_t *handle*)

Discontinues any periodic MAC/Phy channel measurements and the reporting of them via listener calls.

Associated data types

v2x_radio_handle_t

Parameters

in	*handle*	Handle to the radio measurements to be stopped.

Returns

V2X_STATUS_SUCCESS.

Dependencies

The measurements must have been started with v2x_radio_start_measurements().

4.21.4.17 int v2x_radio_sock_close (int * *sock_fd*)

Closes a specified socket file descriptor and deregisters any modem resources associated with it (such as reserved SPS bandwidth contracts).

On platforms with access control enabled, the caller needs to have TELUX_CV2X_FLOW_OPS permission to successfully invoke this API.

Parameters

in	*sock_fd*	Socket file descriptor.

Detailed description

This function works on receive, SPS, or event-driven sockets.

The socket file descriptor must be closed when the client exits. We recommend using a trap to catch controlled shutdowns.

Returns

Integer value of the close(sock) operation.

Dependencies

The socket must have been opened with one of the following methods:

- v2x_radio_rx_sock_create_and_bind()
- v2x_radio_tx_sps_sock_create_and_bind()
- v2x_radio_tx_sps_sock_create_and_bind_v2()
- v2x_radio_tx_sps_only_create()
- v2x_radio_tx_sps_only_create_v2()
- v2x_radio_tx_event_sock_create_and_bind()
- v2x_radio_tx_event_sock_create_and_bind_v2()
- v2x_radio_tx_event_sock_create_and_bind_v3()

4.21.4.18 void v2x_radio_set_log_level (int *new_level,* int *use_syslog*)

Configures the V2X log level and destination for SDK and lower layers.

Parameters

in	*new_level*	Log level to set to one of the standard syslog levels (LOG_ERR, LOG_INFO, and so on).
in	*use_syslog*	Destination: send to stdout (0) or syslog (otherwise).

Returns

None.

4.21.4.19 v2x_event_t cv2x_status_poll (uint64_t * *status_age_useconds*)

Polls for the recent V2X status.

Parameters

out	*status_age_useconds*	Pointer to the age in microseconds of the last event (radio status) that is being reported.

Detailed description

This function does not generate any modem control traffic. For efficiency, it simply returns the most recently cached value that was reported from the modem (often reported at a high rate or frequent rate from the modem).

Returns

Indication of success or failure (see v2x_status_enum_type).

4.21.4.20 int v2x_radio_trigger_l2_update (v2x_radio_handle_t *handle*)

Triggers the modem to change its source L2 address by randomly generating a new address.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_CONFIG permission to successfully invoke this API.

Associated data types

v2x_radio_handle_t

Parameters

in	*handle*	Initialized Radio interface on which this data connection is made.

Detailed description

When the change is complete, clients are notified of the new L2 address via the v2x_radio_calls_t::v2x_radio_l2_addr_changed_listener() callback function.

Returns

0 – On success.

Otherwise:

- EPERM – Socket creation failed; for more details, check errno.h.
- EAFNOSUPPORT – On failure to find the interface.
- EACCES – On failure to get the MAC address of the device.

4.21.4.21 int v2x_radio_update_trusted_ue_list (unsigned int *malicious_list_len,* unsigned int *malicious_list[MAX_MALICIOUS_IDS_LIST_LEN],* unsigned int *trusted_list_len,* trusted_ue_info_t *trusted_list[MAX_TRUSTED_IDS_LIST_LEN]*)

Updates the list of malicious and trusted IDs tracked by the modem.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_CONFIG permission to successfully invoke this API.

Associated data types

trusted_ue_info_t

Parameters

in	*malicious_list_len*	Number of malicious IDs in malicious_list.
in	*malicious_list*	List of malicious IDs.
in	*trusted_list_len*	Number of trusted IDs in trusted_list.
in	*trusted_list*	List of trusted IDs.

Returns

0 – On success.

Otherwise:

- EPERM – Socket creation failed; for more details, check errno.h.
- EAFNOSUPPORT – On failure to find the interface.
- EACCES – On failure to get the MAC address of the device.

4.21.4.22 int v2x_radio_tx_sps_sock_create_and_bind_v2 (v2x_radio_handle_t *handle*, v2x_tx_sps_flow_info_t * *sps_flow_info*, v2x_per_sps_↩ reservation_calls_t * *calls*, int *sps_portnum*, int *event_portnum*, int * *sps_sock*, struct sockaddr_in6 * *sps_sockaddr*, int * *event_sock*, struct sockaddr_in6 * *event_sockaddr*)

Creates and binds a socket with a bandwidth-reserved (SPS) Tx flow with the requested ID, priority, periodicity, and size on a specified source port number. The socket is created as an IPv6 UDP socket.

This v2x_radio_tx_sps_sock_create_and_bind_v2() method differs from v2x_radio_tx_sps_sock_create_and_bind() in that you can use the sps_flow_info parameter to specify transmission resource information about the Tx flow.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_FLOW_OPS permission to successfully invoke this API.

Associated data types

v2x_radio_handle_t
v2x_tx_sps_flow_info_t
v2x_per_sps_reservation_calls_t

Parameters

in	*handle*	Identifies the initialized Radio interface on which this data connection is made.
in	*sps_flow_info*	Pointer to the flow information in the v2x_tx_sps_flow_info_t structure.
in	*calls*	Pointer to reservation callbacks or listeners. This parameter is called when underlying radio MAC parameters change related to the SPS bandwidth contract. For example, the callback after a reservation change, or if the timing offset of the SPS adjusts itself in response to traffic. This parameter passes NULL if no callbacks are required.
in	*sps_portnum*	Requested source port number for the bandwidth reserved SPS transmissions.
in	*event_portnum*	Requested source port number for the bandwidth reserved event transmissions, or -1 for no event port.
out	*sps_sock*	Pointer to the socket that is bound to the requested port for Tx with reserved bandwidth.
out	*sps_sockaddr*	Pointer to the IPv6 UDP socket. The sockaddr_in6 buffer is initialized with the IPv6 source address and source port that are used for the bind() function. The caller can then use the buffer for subsequent sendto() function calls.
out	*event_sock*	Pointer to the socket that is bound to the event-driven transmission port.

out	*event_sockaddr*	Pointer to the IPV6 UDP socket. The sockaddr_in6 buffer is initialized with the IPv6 source address and source port that are used for the bind() function. The caller can then use the buffer for subsequent sendto() function calls.

Detailed description

The radio attempts to reserve the flow with the specified size and rate passed in the request parameters.

This function is used only for Tx. It sets up two UDP sockets on the requested two HLOS port numbers.

For only a single SPS flow, indicate the event port number by using a negative number or NULL for the event_sockaddr. For a single event-driven port, use v2x_radio_tx_event_sock_create_and_bind() or v2x_radio_tx_event_sock_create_and_bind_v2() or v2x_radio_tx_event_sock_create_and_bind_v3() instead.

Because the modem endpoint requires a specific global address, all data sent on these sockets must have a configurable IPv6 destination address for the non-IP traffic.

The Priority parameter of the SPS reservation is used only for the reserved Tx bandwidth (SPS) flow. The non-SPS/event-driven data sent to the event_portnum parameter is prioritized on the air, based on the IPv67 Traffic Class of the packet.

The caller is expected to identify two unused local port numbers to use for binding: one for the event-driven flow and one for the SPS flow.

This call is a blocking call. When it returns, the sockets are ready to use, assuming there is no error.

Returns

0 – On success.

Otherwise:

- EPERM – Socket creation failed; for more details, check errno.h.
- EAFNOSUPPORT – On failure to find the interface.
- EACCES – On failure to get the MAC address of the device.

Dependencies

The interface must be pre-initialized with v2x_radio_init() or v2x_radio_init_v2(). The handle from that function must be used as the parameter in this function.

4.21.4.23 int v2x_radio_tx_sps_only_create_v2 (v2x_radio_handle_t *handle*, v2x_tx_sps_flow_info_t * *sps_flow_info*, v2x_per_sps_reservation_↩ calls_t * *calls*, int *sps_portnum*, int * *sps_sock*, struct sockaddr_in6 * *sps_sockaddr*)

Creates a socket with a bandwidth-reserved (SPS) Tx flow.

Only SPS transmissions are to be implemented for the socket, which is created as an IPv6 UDP socket.

This v2x_radio_tx_sps_only_create_v2() method differs from v2x_radio_tx_sps_only_create() in that you can use the sps_flow_info parameter to specify transmission resource information about the Tx flow.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_FLOW_OPS permission to successfully invoke this API.

Associated data types

v2x_radio_handle_t
v2x_tx_sps_flow_info_t
v2x_per_sps_reservation_calls_t

Parameters

in	*handle*	Identifies the initialized Radio interface on which this data connection is made.
in	*sps_flow_info*	Pointer to the flow information in the v2x_tx_sps_flow_info_t structure.
in	*calls*	Pointer to reservation callbacks or listeners. This parameter is called when underlying radio MAC parameters change related to the SPS bandwidth contract. For example, the callback after a reservation change, or if the timing offset of the SPS adjusts itself in response to traffic. This parameter passes NULL if no callbacks are required.
in	*sps_portnum*	Requested source port number for the bandwidth reserved SPS transmissions.
out	*sps_sock*	Pointer to the socket that is bound to the requested port for Tx with reserved bandwidth.
out	*sps_sockaddr*	Pointer to the IPv6 UDP socket. The sockaddr_in6 buffer is initialized with the IPv6 source address and source port that are used for the bind() function. The caller can then use the buffer for subsequent sendto() function calls.

Detailed description

The radio attempts to reserve the flow with the specified size and rate passed in the request parameters.

This function is used only for Tx. It sets up a UDP socket on the requested HLOS port number. Because the modem endpoint requires a specific global address, all data sent on the socket must have a configurable IPv6 destination address for the non-IP traffic.

The caller is expected to identify an unused local port number to use for binding the SPS flow.

This call is a blocking call. When it returns, the socket is ready to use, assuming there is no error.

Returns

0 – On success.

Otherwise:

- EPERM – Socket creation failed; for more details, check errno.h.
- EINVAL – On failure to find the interface or get bad parameters.

Dependencies

The interface must be pre-initialized with v2x_radio_init() or v2x_radio_init_v2(). The handle from that function must be used as the parameter in this function.

4.21.4.24 v2x_status_enum_type v2x_radio_tx_reservation_change_v2 (int * *sps_sock,* v2x_tx_sps_flow_info_t * *updated_flow_info*)

Adjusts the reservation for transmit bandwidth.

This v2x_radio_tx_reservation_change_v2() method differs from v2x_radio_tx_reservation_change() in that you can use the updated_flow_info parameter to specify transmission resource information about the Tx flow.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_FLOW_OPS permission to successfully invoke this API.

Associated data types

v2x_tx_sps_flow_info_t

Parameters

out	*sps_sock*	Pointer to the socket bound to the requested port.
in	*updated_flow_info*	Pointer to the new reservation information.

Detailed description

This function will not update reservation priority. Can be used as follows:

- When the bandwidth requirement changes in periodicity (for example, due to an application layer DCC algorithm)
- Because the packet size is increasing (for example, due to a growing path history size in a BSM).

When the reservation change is complete, a callback to the structure is passed in a v2x_radio_init() or v2x_radio_init_v2() call.

Returns

V2X_STATUS_SUCCESS.

Error code – On failure (see v2x_status_enum_type).

Dependencies

An SPS flow must have been successfully initialized with v2x_radio_tx_sps_sock_create_and_bind() or v2x_radio_tx_sps_sock_create_and_bind_v2().

4.21.4.25 v2x_status_enum_type v2x_radio_tx_event_flow_info_change (int * *sock*, v2x_tx_flow_info_t * *updated_flow_info*)

Adjusts the flow parameters for an existing Tx event socket.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_CONFIG permission to successfully invoke this API.

Associated data types

v2x_tx_flow_info_t

Parameters

out	*sock*	Pointer to the socket bound to the requested port.
in	*updated_flow_info*	Pointer to the new flow parameters.

Detailed description

When the reservation change is complete, a callback to the structure is passed in a v2x_radio_init() or v2x_radio_init_v2() call.

This call is a blocking call. When it returns, the socket is ready to be use, assuming there is no error.

Returns

V2X_STATUS_SUCCESS.

Error code – On failure (see v2x_status_enum_type).

Dependencies

An event flow must have been successfully initialized with v2x_radio_tx_event_sock_create_and_bind() or v2x_radio_tx_event_sock_create_and_bind_v2() v2x_radio_tx_event_sock_create_and_bind_v3().

4.21.4.26 v2x_status_enum_type start_v2x_mode ( )

Starts V2X mode.

The V2X radio status must be INACTIVE. If the status is ACTIVE or SUSPENDED (see v2x_event_t), call stop_v2x_mode() first.

This call is a blocking call. When it returns, V2X mode has been started, assuming there is no error.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_OPS permission to successfully invoke this API.

Returns

V2X_STATUS_SUCCESS.

Otherwise:

- V2X_STATUS_EALREADY – Failure because V2X mode is already started.
- V2X_STATUS_FAIL – Other failure.

Dependencies

V2X radio status must be V2X_INACTIVE (v2x_event_t).

4.21.4.27 v2x_status_enum_type stop_v2x_mode ( )

Stops V2X mode.

The V2X radio status must be ACTIVE or SUSPENDED (see v2x_event_t). If the status is INACTIVE, call start_v2x_mode() first.

This call is a blocking call. When it returns, V2X mode has been stopped, assuming there is no error.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_OPS permission to successfully invoke this API.

Returns

V2X_STATUS_SUCCESS.

Otherwise:

- V2X_STATUS_EALREADY – Failure because V2X mode is already stopped.
- V2X_STATUS_FAIL – Other failure.

Dependencies

V2X radio status must be V2X_ACTIVE, V2X_TX_SUSPENDED, V2X_RX_SUSPENDED, or V2X_TXRX_SUSPENDED.

4.21.4.28 v2x_radio_handle_t v2x_radio_init_v2 (traffic_ip_type_t *ip_type*, v2x_↩ concurrency_sel_t *mode*, v2x_radio_calls_t * *callbacks_p*, void * *ctx_p*)

Deprecated

This API has been deprecated. Please use v2x_radio_init_v3() instead.

Initializes the Radio interface and sets the callback that will be used when events in the radio change (including when radio initialization is complete).

Associated data types

traffic_ip_type_t
v2x_concurrency_sel_t
v2x_radio_calls_t

Parameters

in	*ip_type*	The Ip or non-Ip interface.
in	*mode*	WAN concurrency mode, although the radio might not support concurrency. Errors can be generated.
in	*callbacks_p*	Pointer to the v2x_radio_calls_t structure that is prepopulated with function pointers used during radio events (such as loss of time synchronization or accuracy) for subscribers. This parameter also points to a callback for this initialization function.
in	*ctx_p*	Voluntary pointer to the first parameter on the callback.

Detailed description

This function call is a blocking, and it is a control plane action.

Use v2x_radio_deinit() when radio operations are complete.

Callback is made when initialization is complete.

Returns

Handle to the specified initialized radio. The handle is used for reconfiguring, opening or changing, and closing reservations.

V2X_RADIO_HANDLE_BAD – If there is an error. No initialization callback is made.

4.21.4.29 int v2x_radio_init_v3 (v2x_concurrency_sel_t *mode*, v2x_radio_calls_t * *callbacks_p*, void * *ctx_p*, v2x_radio_handle_t * *ip_handle_p*, v2x_radio_handle_t * *non_ip_handle_p*)

Initializes Cv2x radio and sets the callback that will be used when events in the radio change (including when radio initialization is complete). The callers can get the handles of Cv2x IP and non-IP interface on success. The handle of interface is used for reconfiguring, opening or changing, and closing reservations.

Associated data types

v2x_concurrency_sel_t
v2x_radio_calls_t

Parameters

in	*mode*	WAN concurrency mode, although the radio might not support concurrency. Errors can be generated.
in	*callbacks_p*	Pointer to the v2x_radio_calls_t structure that is prepopulated with function pointers used during radio events (such as loss of time synchronization or accuracy) for subscribers. This parameter also points to a callback for this initialization function.
in	*ctx_p*	Voluntary pointer to the first parameter on the callback.
out	*ip_handle_p*	Pointer to the handle of IP interface. Pass nullptr if IP interface is not used.
out	*non_ip_handle_p*	Pointer to the handle of non-IP interface. Pass nullptr if non-IP interface is not used.

Detailed description

This function call is a blocking, and it is a control plane action.

Use v2x_radio_deinit() with either IP or non-IP handle when radio operations are complete.

Callback is made when initialization is complete.

Returns

0 – On success.

Otherwise:

- EINVAL – Invalid input parmaters.
- EPERM – Radio initialization failed.

Dependencies

This API might fail if the underlying Cv2x status is currently in an inactive state. Use v2x_register_ext_radio_status_listener to register a listener for CV2X overall Tx/Rx status, then use v2x_get_ext_radio_status to get current V2X overall radio status.

4.21.4.30 int v2x_radio_tx_event_sock_create_and_bind_v3 (traffic_ip_type_t *ip_type*, int *v2x_id*, int *event_portnum*, v2x_tx_flow_info_t * *event_flow_info*, struct sockaddr_in6 * *event_sockaddr*, int * *sock*)

Opens and binds an event-driven socket (one with no bandwidth reservation). The socket is bound as an AF_INET6 UDP type socket.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_FLOW_OPS permission to successfully invoke this API.

This v2x_radio_tx_event_sock_create_and_bind_v3() method differs from v2x_radio_tx_event_sock_create_and_bind_v2() in that you can use the traffic_ip_type_t parameter to specify traffic ip type instead of requiring the interface name.

Associated data types

v2x_tx_flow_info_t traffic_ip_type_t

Parameters

in	*ip_type*	traffice_ip_type.
in	*v2x_id*	Used for transmissions that are ultimately mapped to an L2 destination address.
in	*event_portnum*	Local port number to which the socket is bound. Used for transmissions of this ID.
in	*event_flow_info*	Pointer to the event flow parameters.
out	*event_sockaddr*	Pointer to the sockaddr_ll structure buffer to be initialized.
out	*sock*	Pointer to the file descriptor. Loaded when the function is successful.

Detailed description

This function is used only for Tx when no periodicity is available for the application type. If you know your transmit data periodicity, use v2x_radio_tx_sps_sock_create_and_bind() or v2x_radio_tx_sps_sock_create_and_bind_v2() instead.

These event-driven sockets pay attention to QoS parameters in the IP socket.

Returns

0 – On success.

Otherwise:

- EPERM – Socket creation failed; for more details, check errno.h.
- EAFNOSUPPORT – On failure to find the interface.
- EACCES – On failure to get the MAC address of the device.

4.21.4.31 v2x_status_enum_type get_iface_name (traffic_ip_type_t *ip_type,* char * *iface_name,* size_t *buffer_len*)

Returns interface name set during radio initialization.

Associated data types

traffic_ip_type_t

Parameters

in	*ip_type*	traffic_ip_type_t
out	*iface_name*	pointer to buffer for interface name
in	*buffer_len*	length of the buffer passed for interface name. Must be at least the max buffer size for an interface name (IFNAMSIZE).

Detailed description

This function should only be called after successfully initializing a radio.

Returns

V2X_STATUS_SUCCESS.

V2X_STATUS_FAIL – If there is an error. Interface name will be an empty string.

4.21.4.32 int v2x_radio_tcp_sock_create_and_bind (v2x_radio_handle_t *handle,* const v2x_tx_flow_info_t * *event_info,* const socket_info_t * *sock_info,* int * *sock_fd,* struct sockaddr_in6 * *sockaddr*)

Creates a TCP socket for event Tx and Rx. The socket is bound as an AF_INET6 TCP type socket.

This v2x_radio_tcp_sock_create_and_bind() API creates a new TCP socket and binds the socket to the IPv6 address of local IP interface with specified source port. Additionally, this API also registers a Tx event flow and subscribes Rx with specified service ID to enable TCP control and data packets in both transmitting and receiving directions.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_FLOW_OPS permission to successfully invoke this API.

If the created socket is expected to work as TCP client mode, the caller must establish a connection to the address specifed using function connect(), and then use the socket for send() and recv() on successful connection. The caller must release the created socket and associated resources with v2x_radio_sock_close().

If the created socket is expected to work as TCP server mode, the caller must mark the created socket as a listening socket with function listen(), that is, as a socket that will be used to accept incoming connection requests using accept(). The caller can then use the connected socket returned by accept() for send() and recv(). The caller must close all connected sockets returned by accept() with function close() first, and then release the listening socket and associated resources with v2x_radio_sock_close().

This call is a blocking call. When it returns, the created TCP socket is ready to use, assuming there is no error.

Associated data types

v2x_radio_handle_t

Parameters

in	*handle*	Identifies the initialized Radio interface. The caller must specify IP interface for raido initilization.
in	*event_info*	Pointer to the Tx event flow information.
in	*sock_info*	Pointer to the TCP socket information.
out	*sock_fd*	Pointer to the socket that, on success, returns the TCP socket descriptor. The caller must release this socket with v2x_radio_sock_close().
out	*sockaddr*	Pointer to the address of TCP socket. The sockaddr_in6 buffer is initialized with the IPv6 source address and source port that are used for the bind.

Detailed description

You can execute any sockopts that are appropriate for this type of socket (AF_INET6).

Returns

0 – On success.

Otherwise:

- EINVAL – On failure to find the interface or get bad parameters.
- EPERM – Socket operation failed; for more details, check errno.h.

Dependencies

The interface used for IP communication must be pre-initialized with v2x_radio_init(). The handle from that function must be used as the parameter in this function.

4.21.4.33 v2x_status_enum_type v2x_set_peak_tx_power (int8_t *txPower*)

Set RF peak cv2x transmit power. This affects the power for all existing flows and for any flow created int the future.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_CONFIG permission to successfully invoke this API.

Precondition – v2x mode enabled.

Parameters

in	*txPower*	Desired global Cv2x peak tx power in dbm

Returns

V2X_STATUS_SUCCESS on success. Error status otherwise.

4.21.4.34 v2x_status_enum_type v2x_set_l2_filters (uint32_t *list_len,* src_l2_filter_↩ info ∗ *list_array*)

Set src L2 ID list for filtering. This affects/disables receiving packets from the src L2 IDs in the list.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_CONFIG permission to successfully invoke this API.

Parameters

in	*list_len*	number of rc L2 IDs, max value 50
in	*list_array*	array that stores the src L2 IDs, durations and pppp values for filter

Returns

V2X_STATUS_SUCCESS on success. Error status otherwise.

4.21.4.35 v2x_status_enum_type v2x_remove_l2_filters (uint32_t *list_len,* uint32_t ∗ *l2_id_list*)

Remove specific src L2 ID list for filtering. This affects/enables receiving packets from the src L2 IDs in the list.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_CONFIG permission to successfully invoke this API.

Parameters

in	*list_len*	number of rc L2 IDs, max value 50
in	*l2_id_list*	array that stores the src L2 IDs

Returns

V2X_STATUS_SUCCESS on success. Error status otherwise.

4.21.4.36 v2x_status_enum_type v2x_register_tx_status_report_listener (uint16_t *port,* v2x_tx_status_report_listener *callback*)

Registers a listener for CV2X Tx status report.

Associated data types

v2x_tx_status_report_listener

Parameters

in	*port*	Set this value to the port number of registered Tx Flow if user wants to receive Tx status report associated with its own Tx flow. If user wants to receive Tx status report associated with all Tx flows in system, set this value to 0.
in	*callback*	Callback function of v2x_tx_status_report_listener structure that is called on Tx status reports.

Returns

V2X_STATUS_SUCCESS.

V2X_STATUS_FAIL – If there is an error.

Dependencies

CV2X radio must be pre-initialized with v2x_radio_init_v2() or v2x_radio_init_v3().

4.21.4.37 v2x_status_enum_type v2x_deregister_tx_status_report_listener (uint16_t *port*)

Deregisters a listener for CV2X Tx status report.

Associated data types

v2x_tx_status_report_listener

Parameters

in	*port*	Port number of previously registered v2x_tx_status_report_↩ listener that is to be deregistered. If the listener is registered with port number 0, set this value to 0 to deregister the listener.

Detailed description

User will not receive Tx status reports after the deregistration.

Returns

V2X_STATUS_SUCCESS.

V2X_STATUS_FAIL – If there is an error.

Dependencies

CV2X radio must be pre-initialized with v2x_radio_init_v2() or v2x_radio_init_v3().

4.21.4.38 v2x_status_enum_type v2x_set_global_IPaddr (uint8_t *prefix_len,* uint8_t * *ipv6_addr*)

Set CV2X global IP address for the IP interface.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_CONFIG permission to successfully invoke this API.

Parameters

in	*prefix_len*	CV2X global IP address prefix length in bits, range [64, 128]
in	*ipv6_addr*	CV2X global IP address.

Returns

V2X_STATUS_SUCCESS on success. Error status otherwise.

4.21.4.39 v2x_status_enum_type v2x_set_ip_routing_info (uint8_t * *dest_mac_addr*)

Set CV2X IP interface global IP unicast routing information.

Parameters

in	*dest_mac_addr*	CV2X destination L2 address for unicast routing purpose. expecting a 6 bytes array address, in which the L2 addr stored in the last 3 entries in big endian order.

Returns

V2X_STATUS_SUCCESS on success. Error status otherwise.

4.21.4.40 v2x_status_enum_type v2x_get_ext_radio_status (v2x_radio_status_ex_t * *status*)

Get current V2X overall radio status and per pool status.

Parameters

out	*status*	Pointer to structure v2x_radio_status_ex_t, which contains V2X overall radio status and per pool status on success.

Returns

V2X_STATUS_SUCCESS on success. Error status otherwise.

4.21.4.41 v2x_status_enum_type v2x_register_ext_radio_status_listener (v2x_ext_radio_status_listener *callback*)

Registers a listener for CV2X overall Tx/Rx status and per pool status.

Associated data types

v2x_ext_radio_status_listener

Parameters

in	*callback*	Callback function of v2x_ext_radio_status_listener structure that is called on CV2X Tx/Rx status change.

Returns

V2X_STATUS_SUCCESS.

V2X_STATUS_FAIL – If there is an error.

4.21.4.42 v2x_status_enum_type v2x_get_slss_rx_info (v2x_slss_rx_info_t * *slss_info*)

Get the current V2X SLSS Rx information.

Parameters

out	*slss_info*	Pointer to structure v2x_slss_rx_info_t, which contains V2X SLSS Rx information on success.

Returns

V2X_STATUS_SUCCESS on success. Error status otherwise.

4.21.4.43 v2x_status_enum_type v2x_register_slss_rx_listener (v2x_slss_rx_listener *callback*)

Registers a listener for CV2X SLSS Rx information.

Associated data types

v2x_slss_rx_info_listener

Parameters

in	*callback*	Callback function of v2x_slss_rx_listener structure that is called on SLSS Rx information change.

Returns

V2X_STATUS_SUCCESS.

V2X_STATUS_FAIL – If there is an error.

4.21.4.44 v2x_status_enum_type v2x_deregister_slss_rx_listener (v2x_slss_rx_↩ listener *callback*)

Deregisters a listener for CV2X SLSS Rx information.

Associated data types

v2x_slss_rx_info_listener

Parameters

in	*callback*	Previously registered v2x_slss_rx_listener that is to be deregistered.

Returns

V2X_STATUS_SUCCESS.

V2X_STATUS_FAIL – If there is an error.

4.22 C Vehicle APIs

This section contains C Vehicle APIs related to Cellular-V2X operation. These APIs are provided as reference APIs to work with QTI reference hardware. Customers would need to modify the implementation of these APIs based on the specific H/W and CAN controller being used.

Abstraction of the vehicle system parameters required for CAM/BSM ITS beacons.

4.22.1 Define Documentation

4.22.1.1 #define V2X_VDATA_HANDLE_BAD (-1)

Invalid handle returned when there is an error.

4.22.1.2 #define V2X_J2735_TRACTION_CONTROL_MAX (4)

Guard check value on v2x_transmission_state_enum_type for V2X_TRANSMISSION_MAX. This value is used in a 3-bit bitfield in J2735.

4.22.1.3 #define V2X_TRACTION_CTRL_MAX (4)

Guard check value on v2x_TractionControlStatus_enum_type for V2X_TRACTION_CTRL_MAX. This value is used in a 2-bit bitfield in J2735.

4.22.1.4 #define J2735_ABS_MAX (4)

Guard check value on v2x_AntiLockBrakeStatus_enum_type.

This value cannot be part of the enumeration because the value eventually ends up in 2-bit bitfield over the air.

4.22.1.5 #define V2X_STABILITY_CONTROL_MAX (4)

Guard check value on v2x_StabilityControlStatus_enum_type.

This value is eventually used over the air in a 2-bit bitfield. The enumeration value must never be larger than 4.

4.22.1.6 #define V2X_AUX_BRAKE_MAX (4)

Guard check value on v2x_AuxBrakeStatus_enum_type. This value must never be set this high.

4.22.2 Data Structure Documentation

4.22.2.1 union v2x_control_status_ut

Contains information related to ABS, TCS, stability control state, and other vehicle output controls that might occur and be ongoing. This structure mirrors the J2735 bit frames.

Data fields

Type	Field	Description
struct v2x_control_status_ut	bits	Bit values for control status information.Bit values for control status information.
unsigned short	word	Byte data access to the packed v2x_control_status union structure.

4.22.2.2 struct v2x_control_status_ut.bits

Bit values for control status information.

Data fields

Type	Field	Description
unsigned	unused_padding: 1	Reserved for 16-bit alignment. This field is critical because of 16-bit word access to the packed v2x_control_status_ut union structure.
v2x_AuxBrakeStatus_enum_type	aux_brake_status: 2	Indicates whether the auxiliary braking system is on. **Supported values:** • 0 – Off • 1 – On
v2x_BrakeBoostApplied_enum_type	brake_boost_applied: 2	Indicates whether the brakes are actively being boosted. **Supported values:** • 0 – Not boosted • 1 – Boosted
v2x_StabilityControlStatus_enum_type	stability_control_status: 2	Indicates whether stability control is on and engaged. **Supported values:** • 0 – Off • 1 – On
v2x_AntiLockBrakeStatus_enum_type	antilock_brake_status: 2	Indicates the status of the ABS.
v2x_TractionControlStatus_enum_type	traction_control_status: 2	Indicates whether status of the TCS.

Type	Field	Description
unsigned	rightRear: 1	Indicates whether the right rear brakes are actively being applied. **Supported values:** • 0 – Not applied • 1 – Applied
unsigned	rightFront: 1	Indicates whether the right front brakes are actively being applied. **Supported values:** • 0 – Not applied • 1 – Applied
unsigned	leftRear: 1	Indicates whether the left rear brakes are actively being applied **Supported values:** • 0 – Not applied • 1 – Applied
unsigned	leftFront: 1	Indicates whether the front left brakes are actively being applied. **Supported values:** • 0 – Not applied • 1 – Applied
unsigned	unavailable: 1	No information is available.

4.22.2.3 union vehicleEventFlags_ut

Contains critical events and communication of ongoing events. Also is used for combinations of critical and not critical (wipers) events

This typedef can match the J2735 2016 version or another version you are working with.

Data fields

Type	Field	Description
struct vehicleEventFlags_ut	bits	Bit values for vehicle event flags. A flag indicates the state of the event.Bit values for vehicle event flags.
unsigned short	data	Sixteen-bit word access to the packed vehicleEventFlags union structure.

4.22.2.4 struct vehicleEventFlags_ut.bits

Bit values for vehicle event flags. A flag indicates the state of the event.

Data fields

Type	Field	Description
unsigned	unused: 3	Reserved for 16-bit alignment in the union access.
unsigned	eventAirBag↩ Deployment: 1	Indicates whether the airbag is deployed. **Supported values:** • 0 – Not deployed • 1 – Deployed
unsigned	event↩ Disabled↩ Vehicle: 1	Indicates whether the vehicle is disabled. **Supported values:** • 0 – Not disabled • 1 – Disabled
unsigned	eventFlatTire: 1	Indicates whether the tire is flat. **Supported values:** • 0 – Not flat • 1 – Flat
unsigned	eventWipers↩ Changed: 1	Indicates the status of the windshield wipers. For more information, See the wiper state variables in current_dynamic_vehicle_state_t.
unsigned	eventLights↩ Changed: 1	Indicates the status of one or more lights (such as blinkers and fog).
unsigned	eventHard↩ Braking: 1	Indicates whether hard braking is activated. **Supported values:** • 0 – Not activated • 1 – Activated
unsigned	event↩ Reserved1: 1	Event bit reserved for future use. Do not use.
unsigned	event↩ Hazardous↩ Materials: 1	Indicates whether a hazmat load is present. **Supported values:** • 0 – Not present • 1 – Present
unsigned	eventStability↩ Controlactivated↩ : 1	Indicates whether stability control is on. **Supported values:** • 0 – Off • 1 – On
unsigned	eventTraction↩ ControlLoss: 1	Indicates whether traction control is activated (1) or not (0). **Supported values:** • 0 – Not applied • 1 – Applied

Type	Field	Description
unsigned	eventAB↩ Sactivated: 1	Indicates whether ABS is activated. **Supported values:** • 0 – Not activated • 1 – Activated
unsigned	eventStop↩ LineViolation: 1	Indicates whether the vehicle has detected that a violation of the Stop Line is imminent. **Supported values:** • 0 – Not imminent • 1 – Imminent
unsigned	eventHazard↩ Lights: 1	Indicates whether the hazard lights are on. **Supported values:** • 0 – Off • 1 – On

4.22.2.5 union ExteriorLights_ut

Contains information about the state of the exterior lights.

Data fields

Type	Field	Description
struct Exterior↩ Lights_ut	bits	Bit values for exterior light flags.Bit values for exterior light flags.
unsigned short	data	16-bit short word access to the packed ExteriorLights union structure.

4.22.2.6 struct ExteriorLights_ut.bits

Bit values for exterior light flags.

Data fields

Type	Field	Description
unsigned	parking↩ LightsOn: 1	Indicates whether the parking lights are on. **Supported values:** • 0 – Off • 1 – On
unsigned	fogLightOn: 1	Indicates whether the fog lights are on. **Supported values:** • 0 – Off • 1 – On

Type	Field	Description
unsigned	daytime↩ Running↩ LightsOn: 1	Indicates whether the running lights are on. **Supported values:** • 0 – Off • 1 – On
unsigned	automatic↩ LightControl↩ On: 1	Indicates whether the automatic light control is on. **Supported values:** • 0 – Off • 1 – On
unsigned	hazardSignal↩ On: 1	Indicates whether the hazard lights are on. **Supported values:** • 0 – Off • 1 – On
unsigned	rightTurn↩ SignalOn: 1	Indicates whether the right turn light is on. **Supported values:** • 0 – Off • 1 – On
unsigned	leftTurn↩ SignalOn: 1	Indicates whether the left turn light is on. **Supported values:** • 0 – Off • 1 – On
unsigned	highBeam↩ HeadlightsOn: 1	Indicates whether the high beam headlights are on. **Supported values:** • 0 – Off • 1 – On
unsigned	lowBeam↩ HeadlightsOn: 1	Indicates whether the low beam headlights are on. **Supported values:** • 0 – Off • 1 – On
unsigned	unused: 7	Unused padding bits.

4.22.2.7 struct high_resolution_motion_t

Contains high-resolution motion parameters.

Data fields

Type	Field	Description
double	vehicle_speed	Vehicle speed in meters/second.
double	longitudinal_↩ acceleration	Acceleration in a longitudinal direction, in meters/second^2.
double	yaw_rate	Yaw rate in degrees/second, per SAE J2735.

4.22.2.8 struct current_dynamic_vehicle_state_t

Contains information about the dynamic state of the vehicle.

Data fields

Type	Field	Description
v2x_transmission_state_enum_type	prndl	Specifies the current state of the transmission gear: forward, reverse, and so on.
vehicleEventFlags_ut	events	Flags all critical events and combinations of critical events.
double	throttle_position	Per the J2735 definition, indicates the throttle position from 0% to 100%. However, this value is in double precision between 0 and 1.
double	throttle_confidence	Per the J2735 definition, double precision degrees of confidence.
double	steering_wheel_angle	Per the J2735 definition, double precision degrees of the wheel angle. **Supported values:** -192.0 through 189.0 degrees, with positive being turned to the right
v2x_control_status_ut	brake_status	Indicates whether brakes or emergency brakes (ABS) are activated. **Supported values:** • 0 – Not activated • 1 – Activated
ExteriorLights_ut	exterior_lights	Conglomeration of bits that indicate the status of the exterior lights, such as blinkers.
unsigned char	front_wiper_status	Status of the front windshield wipers.
unsigned char	rear_wiper_status	Status of the rear windshield wipers.

4.22.2.9 struct static_vehicle_parameters_t

Contains static vehicle parameters.

Data fields

Type	Field	Description
double	vehicle_height_cm	Vehicle height in centimeters. This parameter is 0 if the value is not yet available from the vehicle network.
double	vehicle_width_cm	Vehicle width in centimeters. This parameter is 0 if the value is not yet available from the vehicle network.
double	vehicle_length_cm	Vehicle length in centimeters. This parameter is 0 if the value is not yet available from the vehicle network.

Type	Field	Description
double	front_bumper↩ _height_cm	Height of the front bumper, in centimeters. This parameter is 0 if the value is not yet available from the vehicle network.
double	rear_bumper_↩ height_cm	Height of the rear bumper, in centimeters. This parameter is 0 if the value is not yet available from the vehicle network.
double	vehicle_mass↩ _kg	Mass of the vehicle, in kilograms. This parameter is 0 if the value is not yet available from the vehicle network.
double	trailer_↩ weight_kg	Weight of a trailer connected to the vehicle, in kilograms. This parameter is 0 if the value is not yet available from the vehicle network.
char ∗	make	Pointer to the NULL-terminated string with the vehicle manufacturer that this software build supports (such as Ford and GM).
char ∗	model	Pointer to the NULL-terminated string with the vehicle model name that this software build supports (such as Prius, Mustang, Rogue).
unsigned short	begin_model↩ _year	Beginning of the model years that this software build supports.
unsigned short	end_model_↩ year	End of model year that this software build supports. This year might be the same as begin_model_year.

4.22.3 Enumeration Type Documentation

4.22.3.1 enum v2x_transmission_state_enum_type

Valid types for main transmission drive states.

Enumerator

V2X_TRANSMISSION_NEUTRAL
V2X_TRANSMISSION_PARK
V2X_TRANSMISSION_FORWARD_GEARS One of the gears: D, 1, 2, 3, ...
V2X_TRANSMISSION_REVERSE_GEARS
V2X_TRANSMISSION_RESERVED1
V2X_TRANSMISSION_RESERVED2
V2X_TRANSMISSION_RESERVED3
V2X_TRANSMISSION_UNAVAILABLE Status is unknown.
V2X_TRANSMISSION_MAX Sentry variable that must not be exceeded.

4.22.3.2 enum v2x_BrakeBoostApplied_enum_type

Valid types for brake boosting states.

Enumerator

V2X_BRAKEBOOST_UNAVAIL Status is unknown.
V2X_BRAKEBOOST_OFF
V2X_BRAKEBOOST_ON
V2X_BRAKEBOOST_MAX Sentry variable that must not be exceeded.

4.22.3.3 enum v2x_TractionControlStatus_enum_type

Valid types for traction control states.

This enumeration currently matches the J2735 2016 version for the Traction Control System (TCS).

Enumerator

V2X_TRACTION_CTRL_UNAVAIL Status is unknown.
V2X_TRACTION_CTRL_OFF
V2X_TRACTION_CTRL_ON On but currently not engaged.
V2X_TRACTION_CTRL_ENGAGED Actively being engaged.

4.22.3.4 enum v2x_AntiLockBrakeStatus_enum_type

Valid types for antilock-braking states.

This enumeration matches the J2735 2016 version for the Antilock Braking System (ABS) to help BSM packing and unpacking.

Enumerator

V2X_ABS_Unavailable ABS is not equipped, or the status is unknown.
V2X_ABS_Off
V2X_ABS_On On but currently not active.
V2X_ABS_Engaged Actively being engaged on one or more wheels.

4.22.3.5 enum v2x_StabilityControlStatus_enum_type

Valid types for the stability control status. This enumeration should be equivalent to the J2735 BSM version you are working with.

Enumerator

V2X_STABILITY_CONTROL_UNAVAILBLE Stability Control status is unknown.
V2X_STABILITY_CONTROL_OFF Stability Control is not applied.
V2X_STABILITY_CONTROL_ON Stability Control is on, but currently it is not engaged.
V2X_STABILITY_CONTROL_ENGAGED Stability Control is actively being engaged.

4.22.3.6 enum v2x_AuxBrakeStatus_enum_type

Valid types for the auxiliary brake status.

This enumeration should match the J2735 2016 version or any other version you are working with.

Enumerator

V2X_AUX_BRAKE_UNAVAILBLE Vehicle has no auxiliary brake equipment or the status is unknown.
V2X_AUX_BRAKE_OFF
V2X_AUX_BRAKE_ON
V2X_AUX_BRAKE_RESERVED

4.22.4 Function Documentation

4.22.4.1 v2x_api_ver_t v2x_vehicle_api_version (void)

Gets the compiled API version interface (as an integer number).

Returns

v2x_api_ver_t – Filled with the version number, build date, and detailed build information.

4.22.4.2 v2x_status_enum_type v2x_vehicle_get_static_params (static_vehicle_↩ parameters_t * *parameters*)

Returns (via a reference pointer) the static_vehicle_parameters_t structure that enumerates static (unchanging) data items used by ITS stacks.

Associated data types

static_vehicle_parameters_t

Parameters

out	*parameters*	Pointer to the static vehicle parameters, including vehicle dimensions, make, model, and so on.

Detailed description

This call is a nonblocking call. If the values are not yet available from the vehicle, the data element is 0 (NULL).

Because this function is sometimes populated with data from an in-vehicle network, it might be incomplete and only partially populated early in a system start-up. However, all values can be statically compiled in or loaded from an initialization file. In this case, the data is fully provided on the first call.

Returns

V2X_STATUS_SUCCESS – This function is successfully populated with the results.

Error code – If there is a problem (see v2x_status_enum_type).

4.22.4.3 v2x_motion_data_handle_t v2x_high_res_motion_register_listener (v2x_high_res_motion_listener_t *cb*)

Registers for high-resolution motion callbacks from the vehicle data network (CAN bus) when the data changes.

Associated data types

v2x_high_res_motion_listener_t

Parameters

in	*cb*	Callback function to use for this listener.

Returns

Handle number to use with subsequent deregister calls.

-1 – If there is an error in registering a callback.

4.22.4.4 v2x_status_enum_type v2x_high_res_motion_deregister_listener (v2x_motion_data_handle_t *handle*)

Deregisters a previously registered high-resolution motion data callback that was requested via v2x_high_res_motion_register_listener().

Associated data types

v2x_motion_data_handle_t

Parameters

in	*handle*	Handle of the listener callback previously set up.

Returns

V2X_STATUS_SUCCESS.

Dependencies

The callback must have been previously registered with v2x_high_res_motion_register_listener().

4.22.4.5 v2x_vehicle_handle_t v2x_vehicle_register_listener (v2x_vehicle_event_↩ listener_t *cb*, void * *context*)

Registers for a callback for state updates from the vehicle data network (CAN bus). This function requests vehicle data callbacks when data changes or events occur.

Associated data types

v2x_vehicle_event_listener_t

Parameters

in	*cb*	Callback function to use for this listener.
in	*context*	Pointer to the application context for use with the callbacks, which can help the caller code.

Returns

Handle number to use with subsequent deregister calls.

-1 – If there is an error in registering a callback.

4.22.4.6 v2x_status_enum_type v2x_vehicle_deregister_for_callback (v2x_vehicle_↩ handle_t *handle*)

Deregisters a previously registered dynamic event callback that was requested via v2x_vehicle_register_listener().

Associated data types

v2x_vehicle_handle_t

Parameters

in	*handle*	Handle of the listener callback previously set up.

Returns

V2X_STATUS_SUCCESS – If the callback is successfully deregistered.

Dependencies

The callback must have been previously registered.

4.23 C Config APIs

This section contains C Config APIs related to Cellular-V2X operation. These APIs are provided as an abstraction of the CV2X configuration relevant interfaces.

Abstraction of the CV2X configuration relevant interfaces.

4.23.1 Data Structure Documentation

4.23.1.1 struct v2x_config_event_info_t

Information about any update to a V2X config file.

Data fields

Type	Field	Description
v2x_config_↩ soure_t	source	The type of the V2X config file.
v2x_config_↩ event_t	event	Config file event.

4.23.2 Enumeration Type Documentation

4.23.2.1 enum v2x_config_soure_t

V2X configuration source types listed in ascending order of priority. The system always uses the V2X configuration with the highest priority if multiple V2X configuration sources exist.

Enumerator

V2X_CONFIG_SOURCE_UNKNOWN V2X config file source is unknown
V2X_CONFIG_SOURCE_PRECONFIG V2X config file source is preconfig
V2X_CONFIG_SOURCE_SIM_CARD V2X config file source is SIM card
V2X_CONFIG_SOURCE_OMA_DM V2X config file source is OMA-DM

4.23.2.2 enum v2x_config_event_t

Events relevant to CV2X config file.

Enumerator

V2X_CONFIG_EVENT_CHANGED V2X config file is changed
V2X_CONFIG_EVENT_EXPIRED V2X config file is expired

4.23.3 Function Documentation

4.23.3.1 v2x_status_enum_type v2x_register_for_config_change_ind (cv2x_config_event_listener *callback*)

Register listener for any updates to CV2X configuration.

Associated data types

cv2x_config_event_listener

Parameters

in	*callback*	Callback function of cv2x_config_event_listener structure that is used during CV2X config events (such as CV2X configuration expriy or changed).

Detailed description

This function should be called before calling v2x_update_configuration or v2x_retrieve_configuration if the caller has interest in the notification of V2X configuration events.

V2X_CONFIG_EVENT_CHANGED v2x_config_event_t is sent to registered CV2X configuration listeners if the content of the active V2X configuration file is changed by calling v2x_update_configuration or the active V2X configuration file source v2x_config_soure_t is changed from one type to another.

V2X_CONFIG_EVENT_EXPIRED v2x_config_event_t is sent to registered CV2X configuration listeners when the active V2X configuration file is expired.

Returns

V2X_STATUS_SUCCESS.

V2X_STATUS_FAIL – If there is an error.

4.23.3.2 v2x_status_enum_type v2x_update_configuration (const char * *config_file_path*)

Updates the OMA-DM V2X radio configuration file.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_CONFIG permission to successfully invoke this API.

Parameters

in	*config_file_path*	Pointer to the path of the configuration file.

Detailed description

The V2X radio status must be INACTIVE. If the V2X status is ACTIVE or SUSPENDED (see v2x_event_t), call stop_v2x_mode() first.

The functionality of V2X configuration expiration is supported by adding an expiration leaf to the V2X configuration file passed in. When the active configuration expires, the system fallbacks to a lower priority V2X configuration v2x_config_soure_t if existed. If the V2X stauts is active, it changes to suspended when the active V2X configuration expires and then changes to active after the system fallbacks to a lower priority V2X configuration or changes to inactive if no V2X configuration is available.

This call is a blocking call. When it returns the configuration has been updated, assuming no error.

Returns

V2X_STATUS_SUCCESS.

Otherwise:

- V2X_STATUS_EALREADY – Failure because V2X status is not V2X_INACTIVE.
- V2X_STATUS_FAIL – Other failure.

Dependencies

V2X radio status must be V2X_INACTIVE (v2x_event_t).

4.23.3.3 v2x_status_enum_type v2x_retrieve_configuration (const char * *config_file_path*)

Retrieve the V2X radio configuration file.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_CONFIG permission to successfully invoke this API.

Parameters

in	*config_file_path*	Pointer to the path of the configuration file.

This call is a blocking call. When it returns the configuration has been read out, assuming no error.

Returns

V2X_STATUS_SUCCESS.

Otherwise:

- V2X_STATUS_FAIL – Other failure.

4.24 C Packet APIs

This section contains C Packet APIs related to Cellular-V2X packet analysis operation.

Provide utilities and structures for CV2X packet analysis.

4.24.1 Define Documentation

4.24.1.1 #define META_DATA_MASK_SFN 0x01

meata data validity mask of each received packet, used by rx_packet_meta_data_t

4.24.1.2 #define META_DATA_MASK_SUB_CHANNEL_INDEX 0x02

4.24.1.3 #define META_DATA_MASK_SUB_CHANNEL_NUM 0x04

4.24.1.4 #define META_DATA_MASK_PRX_RSSI 0x08

4.24.1.5 #define META_DATA_MASK_DRX_RSSI 0x10

4.24.1.6 #define META_DATA_MASK_L2_DEST 0x20

4.24.1.7 #define META_DATA_MASK_SCI_FORMAT1 0x40

4.24.1.8 #define META_DATA_MASK_DELAY_ESTI 0x80

4.24.2 Data Structure Documentation

4.24.2.1 struct rx_packet_meta_data_t

Contains the detailed meta data report of a packet received.

Data fields

Type	Field	Description
uint32_t	validity	Validity of the meta data
uint16_t	sfn	System Frame Number ∗ 10 + subframe number
uint8_t	sub_channel_↩ index	The subchannel used for transmission
uint8_t	sub_channel_↩ num	Number of subchannels in the Rx pool
int8_t	prx_rssi	RSSI of primary receive signal, in dBm
int8_t	drx_rssi	RSSI of diversity receive signal, in dBm
uint32_t	l2_↩ destination_id	L2 destination ID
uint32_t	sci_format1_↩ info	SCI format1, 3GPP TS 36.213 section 14.1
int32_t	delay_↩ estimation	Packet delay estimation, in Ts (1/(15000 ∗ 2048) seconds)

4.24.3 Function Documentation

4.24.3.1 v2x_status_enum_type v2x_parse_rx_meta_data (const uint8_t * *payload*, uint32_t *length*, rx_packet_meta_data_t * *meta_data*, size_t * *num*, size_t * *meta_data_len*)

Parse the received packet's meta data from the payload

Associated data types

rx_packet_meta_data_t

Parameters

in	*payload*	Pointer to the received message which may contains the meta data reports
in	*length*	Length of the received message in byte
out	*meta_data*	Pointer to the meta data structure array
in,out	*num*	array size of meta_data as input, be assigned to the number of meta data reports parsed out. The caller can use this value to index the array meta_data.
out	*meta_data_len*	length of the meta data in byte parsed out from the payload

Detailed description

This function extracts the received packet's meta data from the payload, there maybe several meta data reports in the received payload.

Returns

V2X_STATUS_SUCCESS.

Otherwise:

- V2X_STATUS_FAIL – Other failure.

4.25 CPP APIs

This section contains C++ APIs related to Cellular-V2X operation. Applications need to have "radio" Linux group permissions to be able to operate successfully with underlying services.

-

4.25.1 Data Structure Documentation

4.25.1.1 class telux::cv2x::ICv2xConfigListener

Listeners for ICv2xConfig must implement this interface.

Public member functions

- virtual void onConfigChanged (const ConfigEventInfo &info)
- virtual ~ICv2xConfigListener ()

4.25.1.1.1 Constructors and Destructors

4.25.1.1.1.1 virtual telux::cv2x::ICv2xConfigListener::~ICv2xConfigListener ( ) [virtual]

Destructor for ICv2xConfigListener

4.25.1.1.2 Member Function Documentation

4.25.1.1.2.1 virtual void telux::cv2x::ICv2xConfigListener::onConfigChanged (const ConfigEventInfo & *info*) [virtual]

Called when CV2X configuration has changed in the below scenarios:

1. The specified configuration source has expired.
2. The active configuration source has changed to the specified configuration source type due to the expiration of the configuration source being used.
3. The specified configuration source has been updated.

Parameters

in	*info*	- Information of CV2X configuration event.

4.25.1.2 class telux::cv2x::ICv2xConfig

Cv2xConfig provide operations to update or request cv2x configuration.

Public member functions

- virtual ~ICv2xConfig ()
- virtual bool isReady ()=0
- virtual std::future< bool > onReady ()=0
- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual telux::common::Status updateConfiguration (const std::string &configFilePath, telux::common::ResponseCallback cb)=0
- virtual telux::common::Status retrieveConfiguration (const std::string &configFilePath, telux::common::ResponseCallback cb)=0
- virtual telux::common::Status registerListener (std::weak_ptr< ICv2xConfigListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< ICv2xConfigListener > listener)=0

4.25.1.2.1 Constructors and Destructors

4.25.1.2.1.1 virtual telux::cv2x::ICv2xConfig::~ICv2xConfig ( ) [virtual]

4.25.1.2.2 Member Function Documentation

4.25.1.2.2.1 virtual bool telux::cv2x::ICv2xConfig::isReady ( ) [pure virtual]

Checks if the Cv2x Config Manager is ready.

Returns

True if Cv2x Config is ready for service, otherwise returns false.

Deprecated

use getServiceStatus instead

4.25.1.2.2.2 virtual std::future<bool> telux::cv2x::ICv2xConfig::onReady ( ) [pure virtual]

Wait for Cv2x Config to be ready.

Returns

A future that caller can wait on to be notified when Cv2x Radio Manager is ready.

Deprecated

the readiness can be notified via the callback passed to Cv2xFactory::getCv2xConfig.

4.25.1.2.2.3 virtual telux::common::ServiceStatus telux::cv2x::ICv2xConfig::getServiceStatus ( ) [pure virtual]

This status indicates whether the Cv2xConfig is in a usable state.

Returns

SERVICE_AVAILABLE - If cv2x config is ready for service. SERVICE_UNAVAILABLE - If cv2x config is temporarily unavailable. SERVICE_FAILED - If cv2x config encountered an irrecoverable failure.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.25.1.2.2.4 virtual telux::common::Status telux::cv2x::ICv2xConfig::updateConfiguration (const std::string & *configFilePath,* telux::common::ResponseCallback *cb*) [pure virtual]

Updates CV2X configuration. Requires CV2X TX/RX radio status be Inactive. If CV2X radio status is Active or Suspended, call ICv2xRadioManager::stopCv2x before trying to update configuration. The functionality of V2X configuration expiration is supported by adding an expiration leaf to the V2X configuration file passed in. When the active configuration expires, the system fallbacks to a lower priority V2X configuration ConfigSourceType if existed. If the V2X stauts is active, it changes to suspended when the active V2X configuration expires and then changes to active after the system fallbacks to a lower priority V2X configuration or changes to inactive if no V2X configuration is available.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_CONFIG permission to successfully invoke this API.

Parameters

in	*configFilePath*	- Path to config file. This is the fully qualified file path including the name of the file.
in	*cb*	- Callback that is invoked when the send is complete. This may be null.

4.25.1.2.2.5 virtual telux::common::Status telux::cv2x::ICv2xConfig::retrieveConfiguration (const std::string & *configFilePath,* telux::common::ResponseCallback *cb*) [pure virtual]

Retrieve active CV2X configuration. The calling application should have write access to the path specified by configFilePath. And if the v2x configuration retrieval request succeed, the file specified by configFilePath will be created and filled with the configuration contents. Otherwise, no file will be created.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_CONFIG permission to successfully invoke this API.

Parameters

in	*configFilePath*	- Path to config file. This is the fully qualified file path including the name of the file.
in	*cb*	- Callback that is invoked when the configuration retrival is complete. This may be null.

4.25.1.2.2.6 virtual telux::common::Status telux::cv2x::ICv2xConfig::registerListener (std::weak_ptr< ICv2xConfigListener > *listener*) [pure virtual]

Registers a listener for this ICv2xConfig.

Parameters

in	*listener*	- Listener that implements ICv2xConfigListener interface.

4.25.1.2.2.7 virtual telux::common::Status telux::cv2x::ICv2xConfig::deregisterListener (std::weak_ptr< ICv2xConfigListener > *listener*) [pure virtual]

Deregisters a listener for this ICv2xConfig.

Parameters

in	*listener*	- Previously registered ICv2xConfigListener that is to be deregistered.

4.25.1.3 class telux::cv2x::Cv2xFactory

Cv2xFactory is the factory that creates the Cv2x Radio.

Public member functions

- virtual std::shared_ptr< ICv2xRadioManager > getCv2xRadioManager (telux::common::InitResponseCb cb=nullptr)
- virtual std::shared_ptr< ICv2xConfig > getCv2xConfig (telux::common::InitResponseCb cb=nullptr)
- virtual std::shared_ptr< ICv2xThrottleManager > getCv2xThrottleManager (telux::common::InitResponseCb cb=nullptr)

Static Public Member Functions

- static Cv2xFactory & getInstance ()

4.25.1.3.1 Member Function Documentation

4.25.1.3.1.1 static Cv2xFactory& telux::cv2x::Cv2xFactory::getInstance ( ) [static]

Get Cv2xFactory instance

Returns

Reference to Cv2xFactory singleton.

4.25.1.3.1.2 virtual std::shared_ptr<ICv2xRadioManager> telux::cv2x::Cv2xFactory::getCv2xRadioManager (telux::common::InitResponseCb *cb = nullptr*) [virtual]

Get Cv2xRadioManager instance.

Parameters

in	*cb*	- Optional callback to get Cv2xRadioManager initialization status

Returns

shared pointer to Cv2x Radio Manager upon success. nullptr otherwise.

4.25.1.3.1.3 virtual std::shared_ptr<ICv2xConfig> telux::cv2x::Cv2xFactory::getCv2xConfig (telux::common::InitResponseCb *cb = nullptr*) [virtual]

Get Cv2xConfig instance.

Parameters

in	*cb*	- Optional callback to get Cv2xConfig initialization status

Returns

shared pointer to Cv2x Config upon success. nullptr otherwise.

4.25.1.3.1.4 virtual std::shared_ptr<ICv2xThrottleManager> telux::cv2x::Cv2xFactory::getCv2xThrottleManager (telux::common::InitResponseCb *cb = nullptr*) [virtual]

Get Cv2xThrottleManager instance.

Returns

shared pointer to Cv2x ThrottleManager upon success. nullptr otherwise.

4.25.1.4 class telux::cv2x::ICv2xRadio

This is class encapsulates a Cv2xRadio interface.

Returned from ICv2xRadioManager::getCv2xRadio

Public member functions

- virtual Cv2xRadioCapabilities getCapabilities () const =0
- virtual bool isReady () const =0
- virtual bool isInitialized () const =0
- virtual std::future< telux::common::Status > onReady ()=0
- virtual telux::common::Status registerListener (std::weak_ptr< ICv2xRadioListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< ICv2xRadioListener > listener)=0
- virtual telux::common::Status createRxSubscription (TrafficIpType ipType, uint16_t port, CreateRxSubscriptionCallback cb, std::shared_ptr< std::vector< uint32_t >> idList=nullptr)=0
- virtual telux::common::Status enableRxMetaDataReport (TrafficIpType ipType, bool enable, std::shared_ptr< std::vector< std::uint32_t >> idList, telux::common::ResponseCallback cb)=0
- virtual telux::common::Status createTxSpsFlow (TrafficIpType ipType, uint32_t serviceId, const SpsFlowInfo &spsInfo, uint16_t spsSrcPort, bool eventSrcPortValid, uint16_t eventSrcPort, CreateTxSpsFlowCallback cb)=0
- virtual telux::common::Status createTxEventFlow (TrafficIpType ipType, uint32_t serviceId, uint16_t eventSrcPort, CreateTxEventFlowCallback cb)=0
- virtual telux::common::Status createTxEventFlow (TrafficIpType ipType, uint32_t serviceId, const EventFlowInfo &flowInfo, uint16_t eventSrcPort, CreateTxEventFlowCallback cb)=0
- virtual telux::common::Status closeRxSubscription (std::shared_ptr< ICv2xRxSubscription > rxSub, CloseRxSubscriptionCallback cb)=0
- virtual telux::common::Status closeTxFlow (std::shared_ptr< ICv2xTxFlow > txFlow, CloseTxFlowCallback cb)=0
- virtual telux::common::Status changeSpsFlowInfo (std::shared_ptr< ICv2xTxFlow > txFlow, const SpsFlowInfo &spsInfo, ChangeSpsFlowInfoCallback cb)=0
- virtual telux::common::Status requestSpsFlowInfo (std::shared_ptr< ICv2xTxFlow > txFlow, RequestSpsFlowInfoCallback cb)=0
- virtual telux::common::Status changeEventFlowInfo (std::shared_ptr< ICv2xTxFlow > txFlow, const EventFlowInfo &flowInfo, ChangeEventFlowInfoCallback cb)=0
- virtual telux::common::Status requestCapabilities (RequestCapabilitiesCallback cb)=0
- virtual telux::common::Status requestDataSessionSettings (RequestDataSessionSettingsCallback cb)=0
- virtual telux::common::Status updateSrcL2Info (UpdateSrcL2InfoCallback cb)=0
- virtual telux::common::Status updateTrustedUEList (const TrustedUEInfoList &infoList, UpdateTrustedUEListCallback cb)=0
- virtual ~ICv2xRadio ()
- virtual std::string getIfaceNameFromIpType (TrafficIpType ipType)=0
- virtual telux::common::Status createCv2xTcpSocket (const EventFlowInfo &eventInfo, const SocketInfo &sockInfo, CreateTcpSocketCallback cb)=0

- virtual telux::common::Status closeCv2xTcpSocket (std::shared_ptr< ICv2xTxRxSocket > sock, CloseTcpSocketCallback cb)=0
- virtual telux::common::Status registerTxStatusReportListener (uint16_t port, std::shared_ptr< ICv2xTxStatusReportListener > listener, telux::common::ResponseCallback cb)=0
- virtual telux::common::Status deregisterTxStatusReportListener (uint16_t port, telux::common::ResponseCallback cb)=0
- virtual telux::common::Status setGlobalIPInfo (const IPv6AddrType &ipv6Addr, common::ResponseCallback cb)=0
- virtual telux::common::Status setGlobalIPUnicastRoutingInfo (const GlobalIPUnicastRoutingInfo &destL2Addr, common::ResponseCallback cb)=0

4.25.1.4.1 Constructors and Destructors

4.25.1.4.1.1 virtual telux::cv2x::ICv2xRadio::~ICv2xRadio ( ) [virtual]

Destructor for ICv2xRadio

4.25.1.4.2 Member Function Documentation

4.25.1.4.2.1 virtual Cv2xRadioCapabilities telux::cv2x::ICv2xRadio::getCapabilities ( ) const [pure virtual]

Get the capabilities of this Cv2xRadio.

Returns

Cv2xRadioCapabilities - Contains capabilities of this Cv2xRadio.

Deprecated

Use requestCapabilities() API

4.25.1.4.2.2 virtual bool telux::cv2x::ICv2xRadio::isReady ( ) const [pure virtual]

Returns true if the radio interface was successfully initialized.

Returns

True if ready. False otherwise.

4.25.1.4.2.3 virtual bool telux::cv2x::ICv2xRadio::isInitialized ( ) const [pure virtual]

Returns true if the radio interface has completed initialization.

Returns

True if initialized. False otherwise.

4.25.1.4.2.4 virtual std::future<telux::common::Status> telux::cv2x::ICv2xRadio::onReady ( ) [pure virtual]

Returns a future that indicated if the radio interface is ready or if radio failed to initialize.

Returns

SUCCESS if Cv2xRadio initialization was successful. Otherwise it returns an Error Code.

4.25.1.4.2.5 virtual telux::common::Status telux::cv2x::ICv2xRadio::registerListener (std::weak_ptr< ICv2xRadioListener > *listener*) [pure virtual]

Registers a listener for this Cv2xRadio.

Parameters

in	*listener*	- Listener that implements Cv2xRadioListener interface.

4.25.1.4.2.6 virtual telux::common::Status telux::cv2x::ICv2xRadio::deregisterListener (std::weak_ptr< ICv2xRadioListener > *listener*) [pure virtual]

Deregisters a listener from this Cv2xRadio.

Parameters

in	*listener*	- Previously registered Cv2xRadioListener that is to be deregistered.

4.25.1.4.2.7 virtual telux::common::Status telux::cv2x::ICv2xRadio::createRxSubscription (IpType *ipType*, uint16_t *port*, CreateRxSubscriptionCallback *cb*, std::shared_ptr< std::vector< uint32_t >> *idList = nullptr*) [pure virtual]

Creates and initializes a new Rx subscription which will be returned in the user-supplied callback.

Parameters

in	*ipType*	- IP traffic type (IP or NON-IP)
in	*port*	- Rx port number
in	*cb*	- Callback function that is invoked when socket creation is complete.

in	*idList*	- Service ID list to subscribe, optional parameter using nullptr by default. Subscribe wildcard if this parameter is set to nullptr.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_FLOW_OPS permission to successfully invoke this API.

Returns

SUCCESS on success. Error status otherwise.

Dependencies The interface must be pre-initialized with init().

4.25.1.4.2.8 virtual telux::common::Status telux::cv2x::ICv2xRadio::enableRxMetaDataReport (TrafficIpType *ipType,* bool *enable,* std::shared_ptr< std::vector< std::uint32_t >> *idList,* telux::common::ResponseCallback *cb*) [pure virtual]

Enable or disable (depends on the parameter "bool enable") the received packets' meta data report for the service IDs provided.

The meta data consist of RF RSSI (received signal strength indicator) status, 32-bit SCI Format 1 (3GPP TS 36.213, section 14.1), packet delay estimation, L2 destination ID, and the resource blocks used for the packet's transmission: subframe, subchannel index.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_INFO permission to successfully invoke this API.

Parameters

in	*ipType*	- IP traffic type (IP or NON-IP)
in	*enable*	- enable the rx meta data if set to true, otherwise disable
in	*idList*	- Service ID list of which the received packets' report are desired
in	*cb*	- Callback that is invoked when meta data is enabled or disabled.

Returns

SUCCESS if no error occurred

Meta data report for IP packets is not supported yet, it will return NOSUPPORTED.

4.25.1.4.2.9 virtual telux::common::Status telux::cv2x::ICv2xRadio::createTxSpsFlow (TrafficIpType *ipType,* uint32_t *serviceId,* const SpsFlowInfo & *spsInfo,* uint16_t *spsSrcPort,* bool *eventSrcPortValid,* uint16_t *eventSrcPort,* CreateTxSpsFlowCallback *cb*) [pure virtual]

Creates a Tx SPS flow with the specified IP type, serviceId, and other parameters specified in reservation. Additionally, an option event flow will be created with the same IP type and serviceId. A Tx socket will be created and initialized for the SPS flow. A Tx socket will be created and initialized for the event flow if the optional event flow is specified.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_FLOW_OPS permission to successfully invoke this API.

Parameters

in	*ipType*	- IP traffic type (IP or NON-IP)
in	*serviceId*	- ID used for transmissions that will be mapped to an L2 destination address. Variable length 4-byte PSID or ITS_AID, or another service ID.
in	*spsInfo*	- SPS reservation parameters.
in	*spsSrcPort*	- Requested source port number for the bandwidth reserved SPS transmissions.
in	*eventSrcPortValid*	- True if an optional event flow is desired. If this field is left false, the event flow will not be created.
in	*eventSrcPort*	- Requested source port number for the optional event flow.
in	*cb*	- Callback function that is invoked when socket creation is complete. This must not be null.

This caller is expected to identify two unused local port numbers

to use for binding: one for the event-driven flow and one for the SPS flow.

Returns

SUCCESS upon success. Error status otherwise.

4.25.1.4.2.10 virtual telux::common::Status telux::cv2x::ICv2xRadio::createTxEventFlow (TrafficIpType *ipType,* uint32_t *serviceId,* uint16_t *eventSrcPort,* CreateTxEventFlowCallback *cb*) [pure virtual]

Creates an event flow. An associated Tx socket will be created and initialized.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_FLOW_OPS permission to successfully invoke this API.

Parameters

in	*ipType*	- IP traffic type (IP or NON-IP)
in	*serviceId*	- ID used for transmissions that will be mapped to an L2 destination address. Variable length 4-byte PSID or ITS_AID, or another service ID.
in	*eventSrcPort*	- Local port number to which the socket is bound. Used for transmissions of this ID.
in	*cb*	- Callback function that is invoked when socket creation is complete. This must not be null.

Detailed description This function is used only for TX when no periodicity is

available for the application type. If your transmit data periodicity is known, use createTxSpsFlow() instead.

These even-driven sockets pay attention to the QoS parameters in

the IP socket.

Returns

SUCCESS upon success. Error status otherwise.

4.25.1.4.2.11 virtual telux::common::Status telux::cv2x::ICv2xRadio::createTxEventFlow (TrafficIpType *ipType,* uint32_t *serviceId,* const EventFlowInfo & *flowInfo,* uint16_t *eventSrcPort,* CreateTxEventFlowCallback *cb*) [pure virtual]

Creates an event flow. An associated Tx socket will be created and initialized.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_FLOW_OPS permission to successfully invoke this API.

Parameters

in	*ipType*	- IP traffic type (IP or NON-IP)
in	*serviceId*	- ID used for transmissions that will be mapped to an L2 destination address. Variable length 4-byte PSID or ITS_AID, or another service ID.
in	*flowInfo*	- Flow configuration parameters
in	*eventSrcPort*	- Local port number to which the socket is bound. Used for transmissions of this ID.
in	*cb*	- Callback function that is invoked when socket creation is complete. This must not be null.

Detailed description This function is used only for TX when no periodicity is

available for the application type. If your transmit data periodicity is known, use createTxSpsFlow() instead.

These even-driven sockets pay attention to the QoS parameters in

the IP socket.

Returns

SUCCESS upon success. Error status otherwise.

4.25.1.4.2.12 virtual telux::common::Status telux::cv2x::ICv2xRadio::closeRxSubscription (std::shared_ptr< ICv2xRxSubscription > *rxSub,* CloseRxSubscriptionCallback *cb*) [pure virtual]

Closes the RxSubscription and frees resources (such as the Rx socket) associated with it.

Parameters

in	*rxSub*	- RxSubscription to close
in	*cb*	- Callback that is invoked when socket close is complete. This may be null.

Returns

SUCCESS if no error occurred.

4.25.1.4.2.13 virtual telux::common::Status telux::cv2x::ICv2xRadio::closeTxFlow (std::shared_ptr< ICv2xTxFlow > *txFlow,* CloseTxFlowCallback *cb*) [pure virtual]

Closes the TxFlow and frees resources associated with it (such as reserved SPS bandwidth contracts and sockets). This function works on both SPS and event flows.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_FLOW_OPS permission to successfully invoke this API.

Parameters

in	*txFlow*	- Tx (SPS or event) flow to close.
in	*cb*	- Callback that is invoked when Tx flow close is complete. This may be null.

Returns

SUCCESS if no error occurred.

4.25.1.4.2.14 virtual telux::common::Status telux::cv2x::ICv2xRadio::changeSpsFlowInfo (std::shared_ptr< ICv2xTxFlow > *txFlow,* const SpsFlowInfo & *spsInfo,* ChangeSpsFlowInfoCallback *cb*) [pure virtual]

Request to change TX SPS Flow reservation parameters.

Parameters

in	*txFlow*	- Tx SPS flow
in	*spsInfo*	- Desired SPS reservation parameters
in	*cb*	- Callback that is invoked upon reservation change. This may be null.

Detailed description

This function does not update reservation priority

Returns

SUCCESS if no error occurred.

4.25.1.4.2.15 virtual telux::common::Status telux::cv2x::ICv2xRadio::requestSpsFlowInfo (std::shared_ptr< ICv2xTxFlow > *txFlow,* RequestSpsFlowInfoCallback *cb*) [pure virtual]

Request SPS flow info.

Parameters

in	*txFlow*	- Tx SPS flow
in	*cb*	- Callback that will be invoked and returns the SPS info. Must not be null.

Returns

SUCCESS if no error occurred.

4.25.1.4.2.16 virtual telux::common::Status telux::cv2x::ICv2xRadio::changeEventFlowInfo (std::shared_ptr< ICv2xTxFlow > *txFlow,* const EventFlowInfo & *flowInfo,* ChangeEventFlowInfoCallback *cb*) [pure virtual]

Request to change TX Event Flow reservation parameters.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_FLOW_OPS permission to successfully invoke this API.

Parameters

in	*txFlow*	- Tx Event flow
in	*flowInfo*	- Desired Event flow parameters
in	*cb*	- Callback that is invoked upon parameter change. This may be null.

Returns

SUCCESS if no error occurred.

4.25.1.4.2.17 virtual telux::common::Status telux::cv2x::ICv2xRadio::requestCapabilities (RequestCapabilitiesCallback *cb*) [pure virtual]

Request modem Cv2x capability information.

Parameters

in	*cb*	- Callback that will be invoked and returns the capability info. Must not be null.

Returns

SUCCESS if no error occurred.

4.25.1.4.2.18 virtual telux::common::Status telux::cv2x::ICv2xRadio::requestDataSessionSettings (RequestDataSessionSettingsCallback *cb*) [pure virtual]

Request data session settings currently in use.

Parameters

in	*cb*	- Callback that will be invoked and returns the data session settings. Must not be null.

Returns

SUCCESS if no error occurred.

4.25.1.4.2.19 virtual telux::common::Status telux::cv2x::ICv2xRadio::updateSrcL2Info (UpdateSrcL2↩ InfoCallback *cb*) [pure virtual]

Requests modem to change L2 info.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_CONFIG permission to successfully invoke this API.

Parameters

in	*cb*	- Callback that will be invoked and returns status. Must not be null.

Returns

SUCCESS if no error occurred.

4.25.1.4.2.20 virtual telux::common::Status telux::cv2x::ICv2xRadio::updateTrustedUEList (const TrustedUEInfoList & *infoList,* UpdateTrustedUEListCallback *cb*) [pure virtual]

Send request to modem to update the list of malicious UE source IDs and trusted UE source IDs with corresponding confidence information.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_CONFIG permission to invoke this API successf

Parameters

in	*infoList*	- Trusted and malicious UE information list
in	*cb*	- Callback that will be invoked and returns status. Must not be null.

Returns

SUCCESS if no error occurred.

4.25.1.4.2.21 virtual std::string telux::cv2x::ICv2xRadio::getIfaceNameFromIpType (TrafficIpType *ipType*) [pure virtual]

Get interface name based on ipType.

Parameters

ipType	- IP traffic type (IP or NON-IP)

Returns

Interface name as a string

4.25.1.4.2.22 virtual telux::common::Status telux::cv2x::ICv2xRadio::createCv2xTcpSocket (const EventFlowInfo & *eventInfo,* const SocketInfo & *sockInfo,* CreateTcpSocketCallback *cb*) [pure virtual]

Creates a CV2X TCP socket with specified event flow information and TCP socket information. The TCP socket will be created and bound to the IPv6 address of local IP interface with specifed source port. Additionally, this API also registers a Tx event flow and subscribes Rx with specified service ID. If the created socket is expected to work as TCP client mode, the caller must connect the created socket to a destination using connect() and then use the socket for send() and recv() on successful connection. If the created socket is expected to work as TCP server mode, the caller must mark this socket as a listening socket using listen() and accept connections received from this listening socket using accept(), and then use the accepted sockets returned from accept() for send() or recv().

On platforms with access control enabled, the caller needs to have TELUX_CV2X_FLOW_OPS permission to successfully invoke this API.

Parameters

in	*eventInfo*	- Information for the Event flow.
in	*sockInfo*	- Information for the TCP socket.
in	*cb*	- Callback function that is invoked when socket creation is complete. This must not be null.

The caller is expected to identify an unused local port number as the source

port number in structure SocketInfo to use for binding.

The caller must release the created socket and associated resources with

closeCv2xTcpSocket. Additionally, if the created socket is marked as a listening socket, the caller must close all the accepted sockets returned by accept() using close() first, and then release the listening socket and associated resources by calling closeCv2xTcpSocket.

Returns

SUCCESS upon success. Error status otherwise.

4.25.1.4.2.23 virtual telux::common::Status telux::cv2x::ICv2xRadio::closeCv2xTcpSocket (std::shared_ptr< ICv2xTxRxSocket > *sock*, CloseTcpSocketCallback *cb*) [pure virtual]

Closes the CV2X TCP socket and frees resources associated with it (such as registered event Tx flow and subscribed Rx service ID and created TCP socket).

On platforms with access control enabled, the caller needs to have TELUX_CV2X_FLOW_OPS permission to successfully invoke this API.

Parameters

in	*sock*	- CV2X TCP socket to close.
in	*cb*	- Callback that is invoked when CV2X TCP socket close is complete. This may be null.

Returns

SUCCESS if no error occurred.

4.25.1.4.2.24 virtual telux::common::Status telux::cv2x::ICv2xRadio::registerTxStatusReportListener (uint16_t *port*, std::shared_ptr< ICv2xTxStatusReportListener > *listener*, telux::common::ResponseCallback *cb*) [pure virtual]

Registers a listener for Tx status report.

Parameters

in	*port*	- Set this value to the port number of registered Tx Flow if user wants to receive Tx status report associated with its own Tx flow. If user wants to receive Tx status report associated with all Tx flows in system, set this value to 0.
in	*listener*	- Listener that implements ICv2xTxStatusReportListener interface.
in	*cb*	- Callback that is invoked when the registration of CV2X Tx status report is complete.

4.25.1.4.2.25 virtual telux::common::Status telux::cv2x::ICv2xRadio::deregisterTxStatusReportListener (uint16_t *port,* telux::common::ResponseCallback *cb*) [pure virtual]

Deregisters a listener for Tx status report.

Parameters

in	*port*	- Port number of previously registered ICv2xTxStatusReport↩ Listener that is to be deregistered. If the listener is registered with port number 0, set this value to 0 to deregister the listener.
in	*cb*	- Callback that is invoked when the deregistration of CV2X Tx status report is complete.

4.25.1.4.2.26 virtual telux::common::Status telux::cv2x::ICv2xRadio::setGlobalIPInfo (const IPv6AddrType & *ipv6Addr,* common::ResponseCallback *cb*) [pure virtual]

Set CV2X global IP address for the IP interface.

Use case and Precondition: OBU: Registers a TX/RX *NON IP* flow for receiving the signed WSA/WRA for IP session initiation; Once receives the IP prefix in the WDS/WRA from RSU, call this method.

RSU: Specifies its own global prefix via this method, and creates/composes WSA/WRA advertising the IP configs.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_CONFIG permission to successfully invoke this API.

Parameters

in	*ipv6Addr*	- CV2X global IP address.
in	*cb*	- Callback that is invoked when set the global IP address complete. This may be null.

Returns

SUCCESS if no error occurred.

4.25.1.4.2.27 virtual telux::common::Status telux::cv2x::ICv2xRadio::setGlobalIPUnicastRoutingInfo (const GlobalIPUnicastRoutingInfo & *destL2Addr,* common::ResponseCallback *cb*) [pure virtual]

Set CV2X IP interface global IP unicast routing information.

Use case and Precondition: OBU: Registers a TX/RX *NON IP* flow for receiving the signed WSA/WRA for IP session initiation; Once receives the IP prefix in the WSA/WRA from RSU, call the setGlobalIPInfo method to update the ip interface with global IP; Now call this method to set the routing information with dest L2 addr negotiated in WSA/WRA.

RSU: Specifies its own global prefix via setGlobalIPInfo, and creates/composes WSA/WRA advertising the IP configs; Now set routing information of its own via this method.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_CONFIG permission to successfully invoke this API.

Parameters

in	*destL2Addr*	- CV2X destination L2 address for unicast routing purpose.
in	*cb*	- Callback that is invoked when set global IP unicast routing information complete. This may be null.

Returns

SUCCESS if no error occurred.

4.25.1.5 class telux::cv2x::ICv2xRadioListener

Listeners for Cv2xRadio must implement this interface.

Public member functions

- virtual void onStatusChanged (Cv2xStatus status)
- virtual void onStatusChanged (Cv2xStatusEx status)
- virtual void onL2AddrChanged (uint32_t newL2Address)
- virtual void onSpsOffsetChanged (int spsId, MacDetails details)
- virtual void onSpsSchedulingChanged (const SpsSchedulingInfo &schedulingInfo)
- virtual void onCapabilitiesChanged (const Cv2xRadioCapabilities &capabilities)
- virtual ~ICv2xRadioListener ()

4.25.1.5.1 Constructors and Destructors

4.25.1.5.1.1 virtual telux::cv2x::ICv2xRadioListener::~ICv2xRadioListener ( ) [virtual]

Destructor for ICv2xRadioListener

4.25.1.5.2 Member Function Documentation

4.25.1.5.2.1 virtual void telux::cv2x::ICv2xRadioListener::onStatusChanged (Cv2xStatus *status*) [virtual]

Called when the status of the CV2X radio has changed.

Parameters

in	*status*	- CV2X radio status.

Deprecated

use onStatusChanged in Cv2xListener

4.25.1.5.2.2 virtual void telux::cv2x::ICv2xRadioListener::onStatusChanged (Cv2xStatusEx *status*) [virtual]

Called when the status of the CV2X radio has changed.

Parameters

in	*status*	- CV2X radio status.

Deprecated

use onStatusChanged in Cv2xListener

4.25.1.5.2.3 virtual void telux::cv2x::ICv2xRadioListener::onL2AddrChanged (uint32_t *newL2Address*) [virtual]

Called when the L2 Address has changed.

Parameters

in	*newL2Address*	- The new L2 address.

4.25.1.5.2.4 virtual void telux::cv2x::ICv2xRadioListener::onSpsOffsetChanged (int *spsId,* MacDetails *details*) [virtual]

Called when SPS offset has changed.

Parameters

in	*spsId*	- SPS Id of the SPS flow
in	*details*	- new SPS MAC PHY details.

Deprecated

use onSpsSchedulingChanged

4.25.1.5.2.5 virtual void telux::cv2x::ICv2xRadioListener::onSpsSchedulingChanged (const SpsSchedulingInfo & *schedulingInfo*) [virtual]

Called when SPS scheduling has changed.

Parameters

in	*schedulingInfo*	- SPS scheduling information .

4.25.1.5.2.6 virtual void telux::cv2x::ICv2xRadioListener::onCapabilitiesChanged (const Cv2xRadioCapabilities & *capabilities*) [virtual]

Called when Cv2x radio capabilities have changed.

Parameters

in	*capabilities*	- Capabilities of the CV2X radio .

4.25.1.6 class telux::cv2x::ICv2xRadioManager

Cv2xRadioManager manages instances of Cv2xRadio.

Public member functions

- virtual bool isReady ()=0
- virtual std::future< bool > onReady ()=0
- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual std::shared_ptr< ICv2xRadio > getCv2xRadio (TrafficCategory category)=0
- virtual telux::common::Status startCv2x (StartCv2xCallback cb)=0
- virtual telux::common::Status stopCv2x (StopCv2xCallback cb)=0
- virtual telux::common::Status requestCv2xStatus (RequestCv2xStatusCallback cb)=0
- virtual telux::common::Status requestCv2xStatus (RequestCv2xStatusCallbackEx cb)=0
- virtual telux::common::Status registerListener (std::weak_ptr< ICv2xListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< ICv2xListener > listener)=0
- virtual telux::common::Status updateConfiguration (const std::string &configFilePath, UpdateConfigurationCallback cb)=0
- virtual telux::common::Status setPeakTxPower (int8_t txPower, common::ResponseCallback cb)=0
- virtual telux::common::Status setL2Filters (const std::vector< L2FilterInfo > &filterList, common::ResponseCallback cb)=0

- virtual telux::common::Status removeL2Filters (const std::vector< uint32_t > &l2IdList, common::ResponseCallback cb)=0
- virtual telux::common::Status getSlssRxInfo (GetSlssRxInfoCallback cb)=0
- virtual ~ICv2xRadioManager ()

4.25.1.6.1 Constructors and Destructors

4.25.1.6.1.1 virtual telux::cv2x::ICv2xRadioManager::~ICv2xRadioManager ( ) [virtual]

4.25.1.6.2 Member Function Documentation

4.25.1.6.2.1 virtual bool telux::cv2x::ICv2xRadioManager::isReady ( ) [pure virtual]

Checks if the Cv2x Radio Manager is ready.

Returns

True if Cv2x Radio Manager is ready for service, otherwise returns false.

Deprecated

use getServiceStatus instead

4.25.1.6.2.2 virtual std::future<bool> telux::cv2x::ICv2xRadioManager::onReady ( ) [pure virtual]

Wait for Cv2x Radio Manager to be ready.

Returns

A future that caller can wait on to be notified when Cv2x Radio Manager is ready.

Deprecated

the readiness can be notified via the callback passed to Cv2xFactory::getCv2xRadioManager.

4.25.1.6.2.3 virtual telux::common::ServiceStatus telux::cv2x::ICv2xRadioManager::getServiceStatus ( ) [pure virtual]

This status indicates whether the Cv2xRadioManager is in a usable state.

Returns

SERVICE_AVAILABLE - If cv2x radio manager is ready for service. SERVICE_UNAVAILABLE - If cv2x radio manager is temporarily unavailable. SERVICE_FAILED - If cv2x radio manager encountered an irrecoverable failure.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.25.1.6.2.4 virtual std::shared_ptr<ICv2xRadio> telux::cv2x::ICv2xRadioManager::getCv2xRadio (TrafficCategory *category*) [pure virtual]

Get Cv2xRadio instance

Parameters

in	*category*	- Specifies the category of the client application. This field is currently unused.

Returns

Reference to Cv2xRadio interface that corresponds to the Cv2x Traffic Category specified.

4.25.1.6.2.5 virtual telux::common::Status telux::cv2x::ICv2xRadioManager::startCv2x (StartCv2x↩ Callback *cb*) [pure virtual]

Put modem into CV2X mode.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_OPS permission to successfully invoke this API.

Parameters

in	*cb*	- Callback that is invoked when Cv2x mode is started

Returns

SUCCESS on success. Error status otherwise.

4.25.1.6.2.6 virtual telux::common::Status telux::cv2x::ICv2xRadioManager::stopCv2x (StopCv2x↩ Callback *cb*) [pure virtual]

Take modem outo of CV2X mode

On platforms with access control enabled, the caller needs to have TELUX_CV2X_OPS permission to successfully invoke this API.

Parameters

in	*cb*	- Callback that is invoked when Cv2x mode is stopped

Returns

SUCCESS on success. Error status otherwise.

4.25.1.6.2.7 virtual telux::common::Status telux::cv2x::ICv2xRadioManager::requestCv2xStatus (RequestCv2xStatusCallback *cb*) [pure virtual]

request CV2X status from modem

Parameters

in	*cb*	- Callback that is invoked when Cv2x status is retrieved

Returns

SUCCESS on success. Error status otherwise.

Deprecated

use requestCv2xStatus(RequestCv2xCalbackEx)

4.25.1.6.2.8 virtual telux::common::Status telux::cv2x::ICv2xRadioManager::requestCv2xStatus (RequestCv2xStatusCallbackEx *cb*) [pure virtual]

request CV2X status from modem

Parameters

in	*cb*	- Callback that is invoked when Cv2x status is retrieved

Returns

SUCCESS on success. Error status otherwise.

4.25.1.6.2.9 virtual telux::common::Status telux::cv2x::ICv2xRadioManager::registerListener (std::weak_ptr< ICv2xListener > *listener*) [pure virtual]

Registers a listener for this manager.

Parameters

in	*listener*	- Listener that implements Cv2xListener interface.

4.25.1.6.2.10 virtual telux::common::Status telux::cv2x::ICv2xRadioManager::deregisterListener (std::weak_ptr< ICv2xListener > *listener*) [pure virtual]

Deregisters a Cv2xListener for this manager.

Parameters

in	*listener*	- Previously registered CvListener that is to be deregistered.

4.25.1.6.2.11 virtual telux::common::Status telux::cv2x::ICv2xRadioManager::updateConfiguration (const std::string & *configFilePath,* UpdateConfigurationCallback *cb*) [pure virtual]

Updates CV2X configuration. Requires CV2X TX/RX radio status be Inactive. If CV2X radio status is Active or Suspended, call stopCv2x before updateConfiguration.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_CONFIG permission to successfully invoke this API.

Parameters

in	*configFilePath*	- Path to config file.
in	*cb*	- Callback that is invoked when the send is complete. This may be null.

Deprecated

Use ICv2xConfig instead

4.25.1.6.2.12 virtual telux::common::Status telux::cv2x::ICv2xRadioManager::setPeakTxPower (int8_t *txPower,* common::ResponseCallback *cb*) [pure virtual]

Set RF peak cv2x transmit power. This affects the power for all existing flows and for any flow created int the future

On platforms with access control enabled, the caller needs to have TELUX_CV2X_CONFIG permission to successfully invoke this API.

Parameters

in	*txPower*	- Desired global Cv2x peak tx power in dbm
in	*cb*	- Callback that is invoked when Cv2x peak tx power is set

Returns

SUCCESS on success. Error status otherwise.

4.25.1.6.2.13 virtual telux::common::Status telux::cv2x::ICv2xRadioManager::setL2Filters (const std::vector< L2FilterInfo > & *filterList,* common::ResponseCallback *cb*) [pure virtual]

Request to install remote UE src L2 filters. This affects receiving of the UEs' packets in specified period with specified PPPP

On platforms with access control enabled, the caller needs to have TELUX_CV2X_CONFIG permission to successfully invoke this API.

Parameters

in	*filterList*	- remote UE src L2 Id, filter duration and PPPP list, max size 50
in	*cb*	- Callback that is invoked when the request is sent

Returns

SUCCESS on success. Error status otherwise.

4.25.1.6.2.14 virtual telux::common::Status telux::cv2x::ICv2xRadioManager::removeL2Filters (const std::vector< uint32_t > & *l2IdList,* common::ResponseCallback *cb*) [pure virtual]

Remove the previously installed filters matching src L2 address list. Hence forth this would allow reception of packets from specified UE's

On platforms with access control enabled, the caller needs to have TELUX_CV2X_CONFIG permission to successfully invoke this API.

Parameters

in	*l2IdList*	- remote UE src L2 Id list, max size 50
in	*cb*	- Callback that is invoked when the request is sent

Returns

SUCCESS on success. Error status otherwise.

4.25.1.6.2.15 virtual telux::common::Status telux::cv2x::ICv2xRadioManager::getSlssRxInfo (GetSlssRxInfoCallback *cb*) [pure virtual]

Get CV2X SLSS Rx information from modem.

On platforms with access control enabled, the caller needs to have TELUX_CV2X_INFO permission to successfully invoke this API.

Parameters

in	*cb*	- Callback that is invoked when Cv2x SLSS Rx information is retrieved.

Returns

SUCCESS on success. Error status otherwise.

4.25.1.7 struct telux::cv2x::SyncRefUeInfo

Encapsulates parameters of an SLSS sync reference UE. Used in SlssRxInfo.

Data fields

Type	Field	Description
uint16_t	slssId	The SLSS ID of the sync reference UE that is defined in 3GPP TS 36.331 chapter 6.3.8.
bool	inCoverage	Indicates whether or not the UE is in coverage of GNSS that is defined in 3GPP TS 36.331 chapter 6.5.2.

Type	Field	Description
SlssSyncPattern	pattern	Indicates the SLSS sync pattern of the UE that is defined in 3GPP TS 36.331 chapter 6.3.8.
uint8_t	rsrp	SLSS RSRP value of the UE in dBm is ((float)rsrp - 256)/2.
bool	selected	Indicates whether or not the sync reference UE has been selected as the timing source.

4.25.1.8 struct telux::cv2x::SlssRxInfo

Encapsulates parameters of CV2X SLSS Rx Information.

Used in telux::cv2x::ICv2xListener::onSlssRxInfoChanged.

Data fields

Type	Field	Description
vector< SyncRefUeInfo >	ueInfo	Vector of detected SLSS sync reference UEs.

4.25.1.9 struct telux::cv2x::SocketInfo

Encapsulates parameters of a CV2X socket.

Used in ICv2xRadio::createCv2xTcpSocket.

Data fields

Type	Field	Description
uint32_t	serviceId	V2X service ID bound to the socket.
uint16_t	localPort	Local port number of the socket used for binding.

4.25.1.10 struct telux::cv2x::Cv2xStatus

Encapsulates status of CV2X radio.

Used in ICv2xRadioManager::requestCv2xStatus and ICv2xRadioListener.

Data fields

Type	Field	Description
Cv2xStatusType	rxStatus	RX status
Cv2xStatusType	txStatus	TX status
Cv2xCauseType	rxCause	RX cause of failure
Cv2xCauseType	txCause	TX cause of failure
uint8_t	cbrValue	Channel Busy Ratio
bool	cbrValueValid	CBR value is valid

4.25.1.11 struct telux::cv2x::Cv2xPoolStatus

Encapsulates status for single pool.

Used in Cv2xStatusEx.

Data fields

Type	Field	Description
uint8_t	poolId	pool ID
Cv2xStatus	status	status

4.25.1.12 struct telux::cv2x::Cv2xStatusEx

Encapsulates status of CV2X radio and per pool status.

Used in ICv2xRadioManager::requestCv2xStatus and Cv2xRadioListener.

Data fields

Type	Field	Description
Cv2xStatus	status	Overall Cv2x status
vector< Cv2xPool↩ Status >	poolStatus	Multi pool status vector
bool	time↩ Uncertainty↩ Valid	Time uncertainty value is valid
float	time↩ Uncertainty	Time uncertainty value in milleseconds

4.25.1.13 struct telux::cv2x::TxPoolIdInfo

Contains minimum and maximum EARFCNs for a given Tx pool ID. Multiple Tx Pools allow the same radio and overall frequency range to be shared for multiple types of traffic like V2V and V2X. Each pool ID and frequency range corresponds to a certain type of traffic. Both edge guard bands are not included in the EARFCN range reported. The calculation for the full bandwidth includes both edge guard bands is: bandwidth(MHz) = (maxFreq-minFreq)/9.

Used in Cv2xRadioCapabilities

Data fields

Type	Field	Description
uint8_t	poolId	TX pool ID.
uint16_t	minFreq	Minimum EARFCN of this pool.
uint16_t	maxFreq	Maximum EARFCN of this pool.

4.25.1.14 struct telux::cv2x::EventFlowInfo

Contains event flow configuration parameters.

Used in ICv2xRadio::createTxEventFlow

Data fields

Type	Field	Description
bool	autoRetrans↩ EnabledValid	Set to true if autoRetransEnabled field is specified. If false, the system will use the default setting.
bool	autoRetrans↩ Enabled	Used to enable automatic-retransmissions.
bool	peakTxPower↩ Valid	Set to true if peakTxPower is used. If false, the system will use the default setting.
int32_t	peakTxPower	Max Tx power setting in dBm.
bool	mcsIndexValid	Set to true if mcsIndex is used. If false, the system will use its default setting.
uint8_t	mcsIndex	Modulation and Coding Scheme Index to use.
bool	txPoolIdValid	Set to true if txPoolId is used. If false, the system will use its default setting.
uint8_t	txPoolId	Transmission Pool ID.
bool	isUnicast	Set to true if isUnicast flow. If false, Non-Unicast flow will be created. Note: Unicast flows ignore subscribed Service Ids

4.25.1.15 struct telux::cv2x::SpsFlowInfo

Used to request the QoS bandwidth contract, implemented in PC5 3GPP V2X radio as a *Semi Persistent Flow* (SPS).

The underlying radio providing the interface might support periodicities of various granularity in 100ms integer multiples (e.g. 200ms, 300ms).

Used in ICv2xRadio::createTxSpsFlow and ICv2xRadio::changeSpsFlowInfo

Data fields

Type	Field	Description
Priority	priority	Specifies one of the 3GPP levels of Priority for the traffic that is pre-reserved on the SPS flow. Default is PRIORITY_2. Use getCapabilities() to discover the supported priority levels. **Deprecated** : periodicity, Use new periodicityMs instead
Periodicity	periodicity	

Type	Field	Description
uint64_t	periodicityMs	This is the new interface to specify periodicity in milliseconds for SpsFlowInfo. Enum Periodicity is deprecated and will be removed in future release.Bandwidth-reserved periodicity interval in interval in milliseconds. There are limits on which intervals the underlying radio supports. Use getCapabilities() to discover minPeriodicityMultiplierMs and maximumPeriodicityMs.
uint32_t	nbytesReserved	Number of bytes of TX bandwidth that are sent every periodicity interval.
bool	autoRetrans↩ EnabledValid	Set to true if autoRetransEnabled field is specified. If false, the system will use the default setting.
bool	autoRetrans↩ Enabled	Used to enable automatic-retransmissions.
bool	peakTxPower↩ Valid	Set to true if peakTxPower is used. If false, the system will use the default setting.
int32_t	peakTxPower	Max Tx power setting in dBm.
bool	mcsIndexValid	Set to true if mcsIndex is used. If false, the system will use its default setting.
uint8_t	mcsIndex	Modulation and Coding Scheme Index to use.
bool	txPoolIdValid	Set to true if txPoolId is used. If false, the system will use its default setting.
uint8_t	txPoolId	Transmission Pool ID.

4.25.1.16 struct telux::cv2x::Cv2xRadioCapabilities

Contains capabilities of the Cv2xRadio.

Used in ICv2xRadio::requestCapabilities and ICv2xRadioListener::onCapabilitiesChanged

Data fields

Type	Field	Description
uint32_t	linkIpMtuBytes	Maximum data payload length (in bytes) of a packet supported by the IP Radio interface.
uint32_t	linkNonIp↩ MtuBytes	Maximum data payload length (in bytes) of a packet supported by the non-IP Radio interface.
Radio↩ Concurrency↩ Mode	max↩ Supported↩ Concurrency	Indicates whether this interface supports concurrent WWAN with V2X (PC5).
uint16_t	nonIpTx↩ Payload↩ OffsetBytes	Byte offset in a non-IP Tx packet before the actual payload begins.
uint16_t	nonIpRx↩ Payload↩ OffsetBytes	Byte offset in a non-IP Rx packet before the actual payload begins. **Deprecated** : periodicitiesSupported, Use new periodicities instead

Type	Field	Description
bitset< 8 >	periodicities↩ Supported	
vector< uint64_t >	periodicities	Specifies the periodicities supported
uint8_t	maxNum↩ Auto↩ Retransmissions	Least frequent bandwidth periodicity that is supported. Above this value, use event-driven periodic messages of a period larger than this value.
uint8_t	layer2Mac↩ AddressSize	Size of the L2 MAC address. Different Radio Access Technologies have different-sized L2 MAC addresses: 802.11 has 6 bytes, whereas 3GPP PC5 has only 3 bytes. Because a randomized MAC address comes from an HSM with good pseudo random entropy, higher layers must know how many bytes of the MAC address to generate.
bitset< 8 >	priorities↩ Supported	Bit set of different priority levels supported by this Cv2xRadio. Refer to Priority
uint16_t	maxNumSps↩ Flows	Maximum number of supported SPS reservations.
uint16_t	maxNumNon↩ SpsFlows	Maximum number of supported event flows (non-SPS ports).
int32_t	maxTxPower	Maximum supported transmission power.
int32_t	minTxPower	Minimum supported transmission power.
vector< Tx↩ PoolIdInfo >	txPoolIds↩ Supported	Vector of supported transmission pool IDs.
uint8_t	isUnicast↩ Supported	Non zero value if UDP event unicast is supported.

4.25.1.17 struct telux::cv2x::MacDetails

Contains MAC information that is reported from the actual MAC SPS in the radio. The offsets can periodically change on any given transmission report.

Data fields

Type	Field	Description
uint32_t	periodicityIn↩ UseNs	Actual transmission interval period (in nanoseconds) scheduled relative to 1PP 0:00.00 time
uint16_t	currently↩ Reserved↩ PeriodicBytes	Actual number of bytes currently reserved at the MAC layer. This number can be slightly larger than original request.
uint32_t	txReservation↩ OffsetNs	Actual offset, from a 1PPS pulse and TX flow periodicity, that the MAC selected and is using for the transmit reservation. If the data goes to the radio with enough time, it can be transmitted on the medium in the next immediately scheduled slot.

4.25.1.18 struct telux::cv2x::SpsSchedulingInfo

Contains SPS packet scheduling information that is reported from the radio.

Used in ICv2xRadioListener::onSpsSchedulingChanged

Data fields

Type	Field	Description
uint8_t	spsId	SPS ID
uint64_t	utcTime	Absolute UTC start time of next selected grant in nanoseconds.
uint32_t	periodicity	Periodicity of the grant in milliseconds.

4.25.1.19 struct telux::cv2x::TrustedUEInfo

Contains time confidence, position confidence, and propogation delay for a trusted UE.

Used in TrustedUEInfo

Data fields

Type	Field	Description
uint32_t	sourceL2Id	Trusted Source L2 ID
float	time↩ Uncertainty	Time uncertainty value in milliseconds.
uint16_t	time↩ Confidence↩ Level	**Deprecated** Use timeUncertainty Time confidence level. Range from 0 to 127 with 0 being invalid/unavailable and 127 being the most confident.
uint16_t	position↩ Confidence↩ Level	Position confidence level. Range from 0 to 127 with 0 being invalid/unavailable and 127 being the most confident.
uint32_t	propagation↩ Delay	Propagation delay in microseconds.

4.25.1.20 struct telux::cv2x::TrustedUEInfoList

Contains list of malicious UE source L2 IDs. Contains list of trusted UE source L2 IDs and associated confidence values.

Used in ICv2xRadio::updateTrustedUEList

Data fields

Type	Field	Description
bool	maliciousIds↩ Valid	Malicious remote UE sources are valid.

Type	Field	Description
vector< uint32_t >	maliciousIds	Malicious remote UE source L2 IDs.
bool	trustedUEs↩ Valid	Trusted remote UE sources are valid.
vector< TrustedUE↩ Info >	trustedUEs	Trusted remote UE sources.

4.25.1.21 struct telux::cv2x::IPv6Address

Contains IPv6 address.

Used in DataSessionSettings

Data fields

Type	Field	Description
uint8_t	addr[16]	

4.25.1.22 struct telux::cv2x::DataSessionSettings

Contains packet data session settings.

Used in ICv2xRadio::requestDataSessionSettings

Data fields

Type	Field	Description
bool	mtuValid	Set to true if mtu is valid.
uint32_t	mtu	MTU size.
bool	ipv6AddrValid	Set to true if ipv6 address is valid.
IPv6Address	ipv6Addr	IPv6 address.

4.25.1.23 struct telux::cv2x::ConfigEventInfo

Information about any update to a CV2X config file.

Used in ICv2xConfigListener::onConfigChanged

Data fields

Type	Field	Description
ConfigSource↩ Type	source	The type of the V2X config file.
ConfigEvent	event	V2X config event.

4.25.1.24 struct telux::cv2x::L2FilterInfo

Contains remote UE source L2 ID that modem will drop on Rx.

Used in ICv2xRadioManager::setL2Filters

Data fields

Type	Field	Description
uint32_t	srcL2Id	< remote UE L2 MAC addr to filter. Duration, in millisec (resolution 100 msec). 0 means delete the filter.
uint32_t	durationMs	/∗ Proximity service per packet priority (PPPP), packets with priority above this value will be dropped. Range 0-7, 0 mean all priority pkts from that UE would be dropped.
uint8_t	pppp	

4.25.1.25 struct telux::cv2x::RFTxInfo

Tx status per Tx chain and Tx power per Tx antenna for a specific transport block.

Used in TxStatusReport

Data fields

Type	Field	Description
RFTxStatus	status	Fault detection status for a specific Tx chain.
int32_t	power	The target Tx power after MPR/AMPR reduction for a specific Tx antenna in dBm∗10 format. Invalid value is -700, it means the corresponding antenna is not being used for the transmission of this transport block.

4.25.1.26 struct telux::cv2x::TxStatusReport

Information on Tx status of a V2X transport block that is reported from low layer.

1. A V2X packet might trigger multiple reports because of the segmentaion and re-Tx in low layer.
2. If a transport block is dropped in low layer, no report will be triggered for that transport block.
3. The power in the array of rfInfo is the target Tx power value in dBm∗10 after MPR/AMPR reduction for a specific Tx antenna. The status in the array of rfInfo is the fault detection status for a specific Tx chain.
   - In CDD mode, two antennas are being used for a specific transport block, both rfInfo[0].power and rfInfo[1].power are valid (not -700), rfInfo[i].status is reflecting the status of Tx chain/Tx antenna i.
   - In TXD mode, data transmission swtiches between two antennas/chains and only one antenna/chain is being used for a specific transport block, the Tx antenna being used has valid power (not -700) in the array of rfInfo, rfInfo[i].status is reflecting the status of Tx chain i or the status of the Tx antenna i whose power is valid (not -700) in the array of rfInfo. Used in ICv2xTxStatusReportListener::onTxStatusReport

Data fields

Type	Field	Description
RFTxInfo	rfInfo[MAX_↩ ANTENNAS↩ _SUPPORT↩ ED]	Tx status per Tx chain and Tx power per Tx antenna.
uint8_t	numRb	Number of resource blocks used for the transport block.
uint8_t	startRb	Start resource block index used for the transport block.
uint8_t	mcs	Modulation and coding scheme used for the transport block that is defined in 3GPP TS 36.213.
uint8_t	segNum	Total number of segments of a V2X packet.
SegmentType	segType	Segment type of the transport block.
TxType	txType	Indication of new Tx or re-Tx of the transport block.
uint16_t	otaTiming	OTA timing in format of system frame number∗10 + subframe number.
uint16_t	port	Port number that can be used to link the report to a specific Tx flow which has the same source port number.

4.25.1.27 struct telux::cv2x::IPv6AddrType

Encapsulates ipv6 prefix length in bits and ipv6 prefix.

Used in ICv2xRadio::setGlobalIPInfo.

Data fields

Type	Field	Description
uint8_t	prefixLen	< ipv6 address prefix length in bits, range [64, 128]
uint8_t	ipv6Addr[C↩ V2X_IPV6_↩ ADDR_ARR↩ AY_LEN]	

4.25.1.28 struct telux::cv2x::GlobalIPUnicastRoutingInfo

Encapsulates destination L2 address.

Used in ICv2xRadio::setGlobalIPUnicastRoutingInfo.

Data fields

Type	Field	Description
uint8_t	destMac↩ Addr[CV2X_↩ MAC_ADD↩ R_LEN]	< Array that stores CV2X L2 MAC address at the last 3 bytes in big endian order.

4.25.1.29 struct telux::cv2x::RxPacketMetaDataReport

Contains the detailed meta data report of a packet received.

The meta data report comes from the same data interface as the packet itself, it consist of RF RSSI (received signal strength indicator) status, 32-bit SCI Format 1 (3GPP TS 36.213, section 14.1), packet delay estimation, L2 destination ID, and the resource blocks used for the packet's transmission: subframe, subchannel index.

The meta data is always received after the successful receipt of the corresponding packet. In order to associate the meta data report with the specific packet, the sfn and subChannelIndex should be present in the packet's payload and the meta data report, so the meta data report can be matched up to the packet.

There is no guarantee that all items listed above will be presented, metaDataMask need to be used for the validity. Use telux::cv2x::Cv2xRxMetaDataHelper::getRxMetaDataInfo to extract the meta data.

Data fields

Type	Field	Description
RxMetaData↩ Validity	metaDataMask	
uint16_t	sfn	
uint8_t	subChannel↩ Index	
uint8_t	subChannel↩ Num	
int8_t	prxRssi	
int8_t	drxRssi	
uint32_t	l2DestinationId	
uint32_t	sciFormat1Info	
int32_t	delay↩ Estimation	

4.25.1.30 class telux::cv2x::Cv2xRxMetaDataHelper

Static Public Member Functions

- static telux::common::Status getRxMetaDataInfo (const uint8_t ∗payload, uint32_t payloadLength, size_t &metaDataLen, std::shared_ptr< std::vector< RxPacketMetaDataReport >> metaDatas)

4.25.1.30.1 Member Function Documentation

4.25.1.30.1.1 static telux::common::Status telux::cv2x::Cv2xRxMetaDataHelper::getRxMetaData↩ Info (const uint8_t ∗ *payload,* uint32_t *payloadLength,* size_t & *metaDataLen,* std::shared_ptr< std::vector< RxPacketMetaDataReport >> *metaDatas*) [static]

4.25.1.31 class telux::cv2x::ICv2xRxSubscription

This class encapsulates a Cv2xRadio Rx Subscription. It contains the Rx socket associated with the subscription from which client applications can read data. This class is referenced in ICv2xRadio::createRxSubscription and ICv2xRadio::closeRxSubscription.

Public member functions

- virtual uint32_t getSubscriptionId () const =0
- virtual TrafficIpType getIpType () const =0
- virtual int getSock () const =0
- virtual struct sockaddr_in6 getSockAddr () const =0
- virtual uint16_t getPortNum () const =0
- virtual std::shared_ptr< std::vector< uint32_t > > getServiceIDList () const =0
- virtual void setServiceIDList (const std::shared_ptr< std::vector< uint32_t >> idList)=0
- virtual ~ICv2xRxSubscription ()

4.25.1.31.1 Constructors and Destructors

4.25.1.31.1.1 virtual telux::cv2x::ICv2xRxSubscription::~ICv2xRxSubscription ( ) [virtual]

4.25.1.31.2 Member Function Documentation

4.25.1.31.2.1 virtual uint32_t telux::cv2x::ICv2xRxSubscription::getSubscriptionId ( ) const [pure virtual]

Accessor for Rx subscription ID

Returns

subscription ID

4.25.1.31.2.2 virtual TrafficIpType telux::cv2x::ICv2xRxSubscription::getIpType ( ) const [pure virtual]

Accessor for IP traffic type

Returns

The Rx subscriptions's IP traffic type (IP or NON-IP)

4.25.1.31.2.3 virtual int telux::cv2x::ICv2xRxSubscription::getSock ( ) const [pure virtual]

Accessor for the socket file descriptor

Returns

The Rx subscriptions's socket fd.

4.25.1.31.2.4 virtual struct sockaddr_in6 telux::cv2x::ICv2xRxSubscription::getSockAddr ( ) const [pure virtual]

Accessor for the socket address description

Returns

The Rx subscriptions's socket address

4.25.1.31.2.5 virtual uint16_t telux::cv2x::ICv2xRxSubscription::getPortNum ( ) const [pure virtual]

Accessor for the subscriptions's port number

Returns

The Rx subscriptions's port num

4.25.1.31.2.6 virtual std::shared_ptr<std::vector<uint32_t> > telux::cv2x::ICv2xRxSubscription::getServiceIDList ( ) const [pure virtual]

Get subscriptions's service ID list

Returns

The Rx subscriptions's service ID list

4.25.1.31.2.7 virtual void telux::cv2x::ICv2xRxSubscription::setServiceIDList (const std::shared_ptr< std::vector< uint32_t >> *idList*) [pure virtual]

Set subscriptions's service ID list

Parameters

in	*idList*	- the subscriptions's service ID list

4.25.1.32 class telux::cv2x::ICv2xThrottleManagerListener

Listener class for getting filter rate update notification.

Public member functions

- virtual void onFilterRateAdjustment (int rate)
- virtual void onServiceStatusChange (telux::common::ServiceStatus status)
- virtual ~ICv2xThrottleManagerListener ()

4.25.1.32.1 Constructors and Destructors

4.25.1.32.1.1 virtual telux::cv2x::ICv2xThrottleManagerListener::~ICv2xThrottleManagerListener ( ) [virtual]

Destructor of ICv2xThrottleManagerListener

4.25.1.32.2 Member Function Documentation

4.25.1.32.2.1 virtual void telux::cv2x::ICv2xThrottleManagerListener::onFilterRateAdjustment (int *rate*) [virtual]

This API is invoked to advise the client to adjust the incoming message filtering rate by `rate` messages/second. If the `rate` is positive, it indicates the client to filter `rate` more messages/second If the `rate` is negative, it indicates the client to filter `rate` less messages/second

Parameters

in	*rate*	the reported filter rate adjustment value.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.25.1.32.2.2 virtual void telux::cv2x::ICv2xThrottleManagerListener::onServiceStatusChange (telux::common::ServiceStatus *status*) [virtual]

This API is invoked when the service status changes for example when a subsytem restart (SSR) occurs

Parameters

in	*status*	- telux::common::ServiceStatus

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.25.1.33 class telux::cv2x::ICv2xThrottleManager

ThrottleManager provides throttle manager client interface.

ThrottleManager provides APIs that allows applications to specify the incoming verification load on the system. This is used to make decisions on how to optimally use the resources available in the system. The API also provides feedback to clients on the suggested filtering that needs to be done when the incoming message verification rate exceeds the instantaneous system capacity.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual telux::common::Status registerListener (std::weak_ptr< ICv2xThrottleManagerListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< ICv2xThrottleManagerListener > listener)=0
- virtual telux::common::Status setVerificationLoad (int load, setVerificationLoadCallback cb)=0
- virtual ~ICv2xThrottleManager ()

4.25.1.33.1 Constructors and Destructors

4.25.1.33.1.1 virtual telux::cv2x::ICv2xThrottleManager::~ICv2xThrottleManager ( ) [virtual]

4.25.1.33.2 Member Function Documentation

4.25.1.33.2.1 virtual telux::common::ServiceStatus telux::cv2x::ICv2xThrottleManager::getServiceStatus ( ) [pure virtual]

This status indicates whether the object is in a usable state.

Returns

SERVICE_AVAILABLE - If location manager is ready for service. SERVICE_UNAVAILABLE - If location manager is temporarily unavailable. SERVICE_FAILED - If location manager encountered an irrecoverable failure.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.25.1.33.2.2 virtual telux::common::Status telux::cv2x::ICv2xThrottleManager::registerListener (std::weak_ptr< ICv2xThrottleManagerListener > *listener*) [pure virtual]

Registers a listener to receive the updated filer rate adjustment data.

Parameters

in	*listener*	- Listener that implement ICv2xThrottleManagerListener interface.

4.25.1.33.2.3 virtual telux::common::Status telux::cv2x::ICv2xThrottleManager::deregisterListener (std::weak_ptr< ICv2xThrottleManagerListener > *listener*) [pure virtual]

Deregister a ICv2xThrottleManagerListener.

Parameters

in	*listener*	- Previously registered Cv2xThrottleManagerListener that is deregistered.

4.25.1.33.2.4 virtual telux::common::Status telux::cv2x::ICv2xThrottleManager::setVerificationLoad (int *load,* setVerificationLoadCallback *cb*) [pure virtual]

Set current measured/average verification load.

Parameters

in	*load*	- current measured verification load(verification/second).
in	*cb*	- callack for indicating the result of set verfication load.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.25.1.34 class telux::cv2x::ICv2xTxFlow

This is class encapsulates a Cv2xRadio Tx flows. It contains the Tx socket associated with the flow through which client applications can send data. This class is referenced in ICv2xRadio::createTxSpsFlow, ICv2xRadio::createTxEventFlow, and ICv2xRadio::closeTxFlow

Public member functions

- virtual uint32_t getFlowId () const =0
- virtual TrafficIpType getIpType () const =0
- virtual uint32_t getServiceId () const =0
- virtual int getSock () const =0
- virtual struct sockaddr_in6 getSockAddr () const =0
- virtual uint16_t getPortNum () const =0
- virtual ~ICv2xTxFlow ()

4.25.1.34.1 Constructors and Destructors

4.25.1.34.1.1 virtual telux::cv2x::ICv2xTxFlow::~ICv2xTxFlow ( ) [virtual]

4.25.1.34.2 Member Function Documentation

4.25.1.34.2.1 virtual uint32_t telux::cv2x::ICv2xTxFlow::getFlowId ( ) const [pure virtual]

Accessor for flow ID. The flow ID should be unique within a process but will not be unique between processes.

Returns

flow ID

4.25.1.34.2.2 virtual TrafficIpType telux::cv2x::ICv2xTxFlow::getIpType ( ) const [pure virtual]

Accessor for IP traffic type

Returns

The flow's IP traffic type (IP or NON-IP)

4.25.1.34.2.3 virtual uint32_t telux::cv2x::ICv2xTxFlow::getServiceId ( ) const [pure virtual]

Accessor for service ID

Returns

The flow's Service ID.

4.25.1.34.2.4 virtual int telux::cv2x::ICv2xTxFlow::getSock ( ) const [pure virtual]

Accessor for the socket file descriptor

Returns

The flow's socket fd.

4.25.1.34.2.5 virtual struct sockaddr_in6 telux::cv2x::ICv2xTxFlow::getSockAddr ( ) const [pure virtual]

Accessor for the socket address description

Returns

The flow's socket address

4.25.1.34.2.6 virtual uint16_t telux::cv2x::ICv2xTxFlow::getPortNum ( ) const [pure virtual]

Accessor for the flow's source port number

Returns

The flow's source port num

4.25.1.35 class telux::cv2x::ICv2xTxRxSocket

This is class encapsulates a Cv2xRadio socket for both Tx and Rx. It contains the socket through which client applications can send and receive data. This class is referenced in ICv2xRadio::createCv2xTcpSocket and ICv2xRadio::closeCv2xTcpSocket.

Public member functions

- virtual uint32_t getId () const =0
- virtual uint32_t getServiceId () const =0
- virtual int getSocket () const =0
- virtual struct sockaddr_in6 getSocketAddr () const =0
- virtual uint16_t getPortNum () const =0
- virtual ~ICv2xTxRxSocket ()

4.25.1.35.1 Constructors and Destructors

4.25.1.35.1.1 virtual telux::cv2x::ICv2xTxRxSocket::~ICv2xTxRxSocket ( ) [virtual]

4.25.1.35.2 Member Function Documentation

4.25.1.35.2.1 virtual uint32_t telux::cv2x::ICv2xTxRxSocket::getId ( ) const [pure virtual]

Accessor for Cv2xRadio socket ID. The socket ID should be unique within a process but will not be unique between processes.

Returns

Cv2xRadio socket ID

4.25.1.35.2.2 virtual uint32_t telux::cv2x::ICv2xTxRxSocket::getServiceId ( ) const [pure virtual]

Accessor for service ID

Returns

The Service ID bound to the socket.

4.25.1.35.2.3 virtual int telux::cv2x::ICv2xTxRxSocket::getSocket ( ) const [pure virtual]

Accessor for the socket file descriptor

Returns

The socket fd.

4.25.1.35.2.4 virtual struct sockaddr_in6 telux::cv2x::ICv2xTxRxSocket::getSocketAddr ( ) const [pure virtual]

Accessor for the socket address description

Returns

The socket address

4.25.1.35.2.5 virtual uint16_t telux::cv2x::ICv2xTxRxSocket::getPortNum ( ) const [pure virtual]

Accessor for the local port number bound to the socket

Returns

The local port number

4.25.1.36 class telux::cv2x::ICv2xTxStatusReportListener

Listeners for CV2X Tx status report must implement this interface.

Public member functions

- virtual void onTxStatusReport (const TxStatusReport &info)
- virtual ~ICv2xTxStatusReportListener ()

4.25.1.36.1 Constructors and Destructors

4.25.1.36.1.1 virtual telux::cv2x::ICv2xTxStatusReportListener::~ICv2xTxStatusReportListener ( ) [virtual]

Destructor for ICv2xTxStatusReportListener

4.25.1.36.2 Member Function Documentation

4.25.1.36.2.1 virtual void telux::cv2x::ICv2xTxStatusReportListener::onTxStatusReport (const TxStatusReport & *info*) [virtual]

Called when a CV2X transport block is transmitted in low layer if a listener for Tx status report has been registered by calling ICv2xRadio::registerTxStatusReportListener.

Parameters

in	*info*	- Tx status of the transport block.

4.25.1.37 class telux::cv2x::Cv2xUtil

Cv2x utility class

Static Public Member Functions

- static uint8_t priorityToTrafficClass (Priority priority)
- static Priority TrafficClassToPriority (uint8_t trafficClass)

4.25.1.37.1 Member Function Documentation

4.25.1.37.1.1 static uint8_t telux::cv2x::Cv2xUtil::priorityToTrafficClass (Priority *priority*) [static]

This function is called to convert cv2x flow priority to traffic class. The Traffic Class indicates class or priority of IPv6 packet. If congestion occurs then packets with least priority will be discarded(See RFC2460 section-7). The result of this method is to fill the IPv6 header traffic class field, it is usually called just before sending IPv6 packet.

Parameters

in	*priority*	- cv2x flow priority

Returns

uint8_t to indicate the result of traffic class

4.25.1.37.1.2 static Priority telux::cv2x::Cv2xUtil::TrafficClassToPriority (uint8_t *trafficClass*) [static]

This function is called to convert IPv6 packet traffic class to cv2x flow priority. The Traffic Class indicates class or priority of IPv6 packet. If congestion occurs then packets with least priority will be discarded(See RFC2460 section-7). It is to get the corresponding cv2x flow priority of the received packets, which usually being called when a new IPv6 packet received.

Parameters

in	*trafficClass*	- class or priority of IPv6 packet(See RFC2460 section-7)

Returns

cv2x flow priority

4.25.2 Enumeration Type Documentation

4.25.2.1 enum telux::cv2x::TrafficCategory [strong]

Defines CV2X Traffic Types.

Enumerator

SAFETY_TYPE Safety message traffic category
NON_SAFETY_TYPE Non-safety message traffic category

4.25.2.2 enum telux::cv2x::Cv2xStatusType [strong]

Defines possible values for CV2X radio RX/TX status.

1. If Rx is in inactive state, Tx should also be in inactive state.
2. If Rx is in active state, Tx should be in active(normal case) or suspended state(sensing or tunnel mode).
3. If Rx is in suspended state, Tx should be in suspended state. Used in Cv2xStatus

Enumerator

INACTIVE RX/TX is inactive
ACTIVE RX/TX is active
SUSPENDED RX/TX is suspended
UNKNOWN RX/TX status unknown

4.25.2.3 enum telux::cv2x::Cv2xCauseType [strong]

Defines possible values for cause of CV2X radio failure. The cause code is only associated with cv2x suspend/inactive status, if cv2x is active, the cause code has no meaning. Used in Cv2xStatus

Enumerator

TIMING CV2X is suspended due to the outage of timing reference.
CONFIG CV2X is inactive due to v2x.xml is missing, invalid, or expired.
UE_MODE CV2X is inactive due to CV2X mode is not started.
GEOPOLYGON CV2X is inactive due to UE enters a geo-polygon that does not support cv2x.
THERMAL CV2X is suspended when the device's temperature is high.
THERMAL_ECALL CV2X is suspended when the device's temperature is high and emergency call is ongoing.
GEOPOLYGON_SWITCH CV2X is suspended when UE switches to a new geopolygon that also supports CV2X and UE is already in CV2X active status, CV2X status will change to active after the update is done.
SENSING CV2X Tx is suspended when GNSS signal recovers or CV2X mode just starts. UE needs sensing for 1 second before Tx can begin, Tx status will change to active after sensing is done.

LPM CV2X is inactive when UE enters Low Power Mode.
DISABLED CV2X is inactive due to CV2X is disabled in the EFS.
NO_GNSS CV2X is inactive due to GNSS signal is not available when starting CV2X.
INVALID_LICENSE CV2X is inactive due to invalid license.
UNKNOWN Invalid cause type only used internally.

4.25.2.4 enum telux::cv2x::SlssSyncPattern [strong]

Defines possible values for SLSS sync pattern. Used in SyncRefUeInfo

Enumerator

OFFSET_IND_1 UE transmits SLSS in subframes indicated by the syncOffsetIndicator1 specified in V2X configuration.
OFFSET_IND_2 UE transmits SLSS in subframes indicated by the syncOffsetIndicator2 specified in V2X configuration.
OFFSET_IND_3 UE transmits SLSS in subframes indicated by the syncOffsetIndicator3 specified in V2X configuration.
ODD_RESERVED UE transmits SLSS in odd-numbered reserved subframes.
EVEN_RESERVED UE transmits SLSS in even-numbered reserved subframes.
UNKNOWN Invalid cause type only used internally.

4.25.2.5 enum telux::cv2x::TrafficIpType [strong]

Defines CV2X traffic type in terms of IP or NON-IP.

Enumerator

TRAFFIC_IP IP message traffic
TRAFFIC_NON_IP NON-IP message traffic

4.25.2.6 enum telux::cv2x::RadioConcurrencyMode [strong]

Defines CV2X modes of concurrency with cellular WWAN.

Used in Cv2xRadioCapabilities

Enumerator

WWAN_NONCONCURRENT No simultaneous WWAN + CV2X on this interface
WWAN_CONCURRENT Interface supports requests for concurrent WWAN + CV2X connections.

4.25.2.7 enum telux::cv2x::Cv2xEvent [strong]

Defines CV2X status change events. The state can change in response to the loss of timing precision or a geofencing change.

Used in ICv2xRadioListener::onStatusChanged

Enumerator

CV2X_INACTIVE

CV2X_ACTIVE
TX_SUSPENDED
TXRX_SUSPENDED

4.25.2.8 enum telux::cv2x::Priority [strong]

Range of supported priority levels, where a lower number means a higher priority. For example, 8 is the current 3GPP standard.

Used in Cv2xRadioCapabilities and SpsFlowInfo

Enumerator

MOST_URGENT
PRIORITY_1
PRIORITY_2
PRIORITY_3
PRIORITY_4
PRIORITY_5
PRIORITY_6
PRIORITY_BACKGROUND
PRIORITY_UNKNOWN

4.25.2.9 enum telux::cv2x::Periodicity [strong]

Range of supported periodicities in milliseconds.

Used in Cv2xRadioCapabilities and SpsFlowInfo

Deprecated

: enum class not going to be supported in future releases. Clients should stop using this. Once a class has been marked as Deprecated, the class could be removed in future releases.

Enumerator

PERIODICITY_10MS
PERIODICITY_20MS
PERIODICITY_50MS
PERIODICITY_100MS
PERIODICITY_UNKNOWN

4.25.2.10 enum telux::cv2x::ConfigSourceType [strong]

V2X configuration source types listed in ascending order of priority. The system always uses the V2X configuration with the highest priority if multiple V2X configuration sources exist.

Used in ConfigEventInfo

Enumerator

UNKNOWN V2X config file source is unknown
PRECONFIG V2X config file source is preconfig
SIM_CARD V2X config file source is SIM card
OMA_DM V2X config file source is OMA-DM

4.25.2.11 enum telux::cv2x::ConfigEvent [strong]

Defines possible values for the events relevant to CV2X config file.

Used in ConfigEventInfo

Enumerator

CHANGED V2X config file is changed
EXPIRED V2X config file is expired

4.25.2.12 enum telux::cv2x::RFTxStatus [strong]

Fault detection for Tx chain that including PA and front end.

Used in RFTxInfo

Enumerator

INACTIVE The Tx chain is not working.
OPERATIONAL The Tx chain is operational.
FAULT Fault detected on the Tx chain.

4.25.2.13 enum telux::cv2x::SegmentType [strong]

Defines possible values for the segment type of a transport block.

Used in TxStatusReport

Enumerator

FIRST V2X packet is segmented, it's the first transport block.
LAST V2X packet is segmented, it's the last transport block.
MIDDLE V2X packet is segmented, it's a transport block between first and last.
ONLY_ONE V2X packet is not segmented, it's the only one transport block.

4.25.2.14 enum telux::cv2x::TxType [strong]

Defines new Tx or re-Tx type relevant to a transport block.

Used in TxStatusReport

Enumerator

NEW_TX New Tx of the V2X transport block.
RE_TX Re-Tx of the V2X transport block.
SLSS_TX Tx of SLSS.

4.25.2.15 enum telux::cv2x::RxMetaDataValidityType

Specify set of RX Meta data that contribute to received packet's meta data report. Used in RxPacketMetaDataReport

Enumerator

RX_SUBFRAME_NUMBER Bit mask to specify whether sfn is valid in RxPacketMetaDataReport Bit mask to specify whether subChannelIndex is valid in RxPacketMetaDataReport

RX_SUBCHANNEL_INDEX Bit mask to specify whether subChannelNum is valid in RxPacketMetaDataReport

RX_SUBCHANNEL_NUMBER Bit mask to specify whether rssi0 is valid in RxPacketMetaDataReport

RX_PRX_RSSI Bit mask to specify whether rssi1 is valid in RxPacketMetaDataReport

RX_DRX_RSSI Bit mask to specify whether l2DestinationId is valid in RxPacketMetaDataReport

RX_L2_DEST_ID Bit mask to specify whether sciFormat1Info is valid in RxPacketMetaDataReport

RX_SCI_FORMAT1 Bit mask to specify whether delayEstimation is valid in RxPacketMetaDataReport

RX_DELAY_ESTIMATION

4.25.3 Variable Documentation

4.25.3.1 constexpr uint8_t telux::cv2x::MAX_ANTENNAS_SUPPORTED = 2u

Defines Maximum number of antennas that is supported.

Used in TxStatusReport

4.26 Audio

- Audio Manager
- Audio Streams
- Transcoder

This section contains APIs related to audio.

4.26.1 Data Structure Documentation

4.26.1.1 class telux::audio::AudioFactory

Allows the creation of an IAudioManager instance.

Public member functions

- virtual std::shared_ptr< IAudioManager > getAudioManager (telux::common::InitResponseCb callback=nullptr)=0

Static Public Member Functions

- static AudioFactory & getInstance ()

4.26.1.1.1 Member Function Documentation

4.26.1.1.1.1 static AudioFactory& telux::audio::AudioFactory::getInstance ( ) [static]

Gets the AudioFactory instance.

4.26.1.1.1.2 virtual std::shared_ptr<IAudioManager> telux::audio::AudioFactory::getAudioManager (telux::common::InitResponseCb *callback* = `nullptr`) [pure virtual]

Gets the IAudioManager instance.

Parameters

in	*callback*	Optional, callback to know the status of the AudioManager initialization

Returns

IAudioManager instance

4.27 Audio Manager

This section contains APIs related to Audio Manager operation.

4.27.1 Data Structure Documentation

4.27.1.1 struct telux::audio::FormatParams

Represents the base class for compressed audio formats.

4.27.1.2 struct telux::audio::AmrwbpParams

Specifies the details of the adaptive multirate wide band format frame.

Data Fields

- uint32_t bitWidth
- AmrwbpFrameFormat frameFormat

4.27.1.2.1 Field Documentation

4.27.1.2.1.1 uint32_t telux::audio::AmrwbpParams::bitWidth

Bit width of the stream (16 or 24)

4.27.1.2.1.2 AmrwbpFrameFormat telux::audio::AmrwbpParams::frameFormat

Refer to AmrwbpFrameFormat

4.27.1.3 struct telux::audio::StreamConfig

Defines the parameters when creating an audio stream.

Data fields

Type	Field	Description
StreamType	type	Refer to StreamType
int	modemSubId	**Deprecated** represents modem subscription ID (default set to 1). Use the StreamConfig::slotId field instead of this
SlotId	slotId	SlotId – specifies the slot ID where the UICC card is inserted. Used in conjuction with StreamType::VOICE_CALL only

Type	Field	Description
uint32_t	sampleRate	Sample rate in Hz. Typical values: • 8k • 16k • 32k • 48k For Bluetooth use-cases, supported values are 8k and 16k. Not used for for voice call, compressed playback and tone generation.
ChannelType↩ Mask	channelType↩ Mask	Refer to ChannelTypeMask
AudioFormat	format	Refer to AudioFormat
vector< DeviceType >	deviceTypes	Defines the list of audio devices DeviceType to use for this stream. For StreamType::PLAY and StreamType::TONE_GENERATOR, a single sink device should be specified. For StreamType::CAPTURE, a single source device should be specified. For StreamType::VOICE_CALL and StreamType::LOOPBACK, both sink and source should be specified with sink as the first device and source as the second.
vector< Direc- tion >	voicePaths	For an in-call audio usecase, this represents the voice path direction Direction
FormatParams *	formatParams	Refer to FormatParams
EcnrMode	ecnrMode	Refer to EcnrMode

4.27.1.4 struct telux::audio::FormatInfo

Specifies the parameters when setting up streams for transcoding.

Data fields

Type	Field	Description
uint32_t	sampleRate	Sample rate in Hz, typical values 8k/16k/32k/48k Sample rate is a dummy paramter for voice stream and compressed playback
ChannelType↩ Mask	mask	Refer to ChannelTypeMask
AudioFormat	format	Refer to AudioFormat
FormatParams *	params	Refer to FormatParams

4.27.1.5 class telux::audio::IAudioListener

Listener for the audio service availability. Refer to telux::common::IServiceStatusListener for details.

Public member functions

- virtual ~IAudioListener ()

4.27.1.5.1 Constructors and Destructors

4.27.1.5.1.1 virtual telux::audio::IAudioListener::~IAudioListener ( ) [virtual]

Destructor of IAudioListener.

4.27.1.6 class telux::audio::IAudioManager

Provides the APIs to discover the supported audio devices, create streams, and subscribe for audio service status updates.

Public member functions

- virtual bool isSubsystemReady ()=0
- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual std::future< bool > onSubsystemReady ()=0
- virtual telux::common::Status getDevices (GetDevicesResponseCb callback=nullptr)=0
- virtual telux::common::Status getStreamTypes (GetStreamTypesResponseCb callback=nullptr)=0
- virtual telux::common::Status createStream (StreamConfig streamConfig, CreateStreamResponseCb callback=nullptr)=0
- virtual telux::common::Status createTranscoder (FormatInfo input, FormatInfo output, CreateTranscoderResponseCb callback)=0
- virtual telux::common::Status deleteStream (std::shared_ptr< IAudioStream > stream, DeleteStreamResponseCb callback=nullptr)=0
- virtual telux::common::Status registerListener (std::weak_ptr< IAudioListener > listener)=0
- virtual telux::common::Status deRegisterListener (std::weak_ptr< IAudioListener > listener)=0
- virtual telux::common::Status getCalibrationInitStatus (GetCalInitStatusResponseCb callback)=0
- virtual ~IAudioManager ()

4.27.1.6.1 Constructors and Destructors

4.27.1.6.1.1 virtual telux::audio::IAudioManager::~IAudioManager ( ) [virtual]

Destructor of the IAudioManager.

4.27.1.6.2 Member Function Documentation

4.27.1.6.2.1 virtual bool telux::audio::IAudioManager::isSubsystemReady ( ) [pure virtual]

Checks if the audio service is ready for use.

Returns

True if the audio service is ready for use, otherwise, False

Deprecated

Use getServiceStatus()

4.27.1.6.2.2 virtual telux::common::ServiceStatus telux::audio::IAudioManager::getServiceStatus ( ) [pure virtual]

Gets the audio service status.

Returns

telux::common::ServiceStatus::SERVICE_AVAILABLE if the audio service is ready for use, telux::common::ServiceStatus::SERVICE_UNAVAILABLE if the audio service is temporarily unavailable (possibly undergoing initialization), telux::common::ServiceStatus::SERVICE_FAILED if the audio service needs re-initialization

4.27.1.6.2.3 virtual std::future<bool> telux::audio::IAudioManager::onSubsystemReady ( ) [pure virtual]

Suggests when the audio service is ready.

Returns

Future to block on until the service status is updated to read

Deprecated

Use telux::common::InitResponseCb in AudioFactory::getAudioManager()

4.27.1.6.2.4 virtual telux::common::Status telux::audio::IAudioManager::getDevices (GetDevicesResponseCb *callback* = *nullptr*) [pure virtual]

Gets the list of the supported audio devices.

Parameters

in	*callback*	Mandatory, callback that will receive the list

Returns

Status telux::common::Status::SUCCESS if the request is initiated successfully, otherwise, an appropriate error code

4.27.1.6.2.5 virtual telux::common::Status telux::audio::IAudioManager::getStreamTypes (GetStreamTypesResponseCb *callback* = *nullptr*) [pure virtual]

Gets the list of the supported stream types.

Parameters

in	*callback*	Mandatory, callback that will receive the list

Returns

Status telux::common::Status::SUCCESS if the request is initiated successfully, otherwise, an appropriate error code

4.27.1.6.2.6 virtual telux::common::Status telux::audio::IAudioManager::createStream (StreamConfig *streamConfig,* CreateStreamResponseCb *callback* = `nullptr`) `[pure virtual]`

Creates an audio stream with the parameters provided.

On platforms with access control enabled, the caller must have TELUX_AUDIO_VOICE, TELUX_AUDIO_PLAY, TELUX_AUDIO_CAPTURE, or TELUX_AUDIO_FACTORY_TEST permission to invoke this method successfully.

Parameters

in	*streamConfig*	Parameters of the stream
in	*callback*	Mandatory, invoked to pass the stream created

Returns

Status telux::common::Status::SUCCESS if the request is initiated successfully, otherwise, an appropriate error code

4.27.1.6.2.7 virtual telux::common::Status telux::audio::IAudioManager::createTranscoder (FormatInfo *input,* FormatInfo *output,* CreateTranscoderResponseCb *callback*) `[pure virtual]`

Set up the transcoder with the given parameters.

Transcoder instance is obtained in CreateTranscoderResponseCb. It can be used only for a single transcoding operation.

On platforms with access control enabled, the caller must have TELUX_AUDIO_TRANSCODE permission to invoke this method successfully.

Parameters

in	*input*	Details of the input to transcode
in	*output*	Details of the transcoded output required
in	*callback*	Invoked to pass the transcoder instance

Returns

Status telux::common::Status::SUCCESS if the request is initiated successfully, otherwise, an appropriate error code

4.27.1.6.2.8 virtual telux::common::Status telux::audio::IAudioManager::deleteStream (std::shared_ptr< IAudioStream > *stream*, DeleteStreamResponseCb *callback* = `nullptr`) [pure virtual]

Deletes the stream created with createStream(). It closes the stream and releases all resources allocated for this stream.

On platforms with access control enabled, the caller must have TELUX_AUDIO_VOICE, TELUX_AUDIO_PLAY, TELUX_AUDIO_CAPTURE, or TELUX_AUDIO_FACTORY_TEST permission to invoke this method successfully.

Parameters

in	*stream*	Stream to delete
in	*callback*	Optional, invoked to pass the result of the stream deletion

Returns

Status telux::common::Status::SUCCESS if the request is initiated successfully, otherwise, an appropriate error code

4.27.1.6.2.9 virtual telux::common::Status telux::audio::IAudioManager::registerListener (std::weak_ptr< IAudioListener > *listener*) [pure virtual]

Registers the given listener to get notified when the audio service status changes. The method IAudioListener::onServiceStatusChange() is invoked to notify of the new status.

Parameters

in	*listener*	Invoked to pass the new service status

Returns

telux::common::Status::SUCCESS if the listener is registered, otherwise, an appropriate error code

4.27.1.6.2.10 virtual telux::common::Status telux::audio::IAudioManager::deRegisterListener (std::weak_ptr< IAudioListener > *listener*) [pure virtual]

Unregisters the given listener registered previously with registerListener().

Parameters

in	*listener*	Listener to unregister

Returns

telux::common::Status::SUCCESS if the listener is unregistered, otherwise, an appropriate error code

4.27.1.6.2.11 virtual telux::common::Status telux::audio::IAudioManager::getCalibrationInitStatus (GetCalInitStatusResponseCb *callback*) [pure virtual]

Gets the current initialization status of the audio calibration database (ACDB). This status is obtained in the GetCalInitStatusResponseCb callback.

Parameters

in	*callback*	Invoked to pass the initialization status

Returns

Status telux::common::Status::SUCCESS if the request is initiated successfully, otherwise, an appropriate error code

4.27.1.7 class telux::audio::IAudioDevice

Represents an audio device. Used in conjunction with GetDevicesResponseCb.

Public member functions

- virtual DeviceType getType ()=0
- virtual DeviceDirection getDirection ()=0
- virtual ~IAudioDevice ()

4.27.1.7.1 Constructors and Destructors

4.27.1.7.1.1 virtual telux::audio::IAudioDevice::~IAudioDevice ( ) [virtual]

Destructor of the IAudioDevice.

4.27.1.7.2 Member Function Documentation

4.27.1.7.2.1 virtual DeviceType telux::audio::IAudioDevice::getType ( ) [pure virtual]

Gets the type of the audio device.

Returns

Type of the audio device

4.27.1.7.2.2 virtual DeviceDirection telux::audio::IAudioDevice::getDirection ( ) [pure virtual]

Gets the direction of the audio device.

Returns

Direction of the audio device

4.27.2 Enumeration Type Documentation

4.27.2.1 enum telux::audio::DeviceType

Represents an audio device. Each device is mapped to its corresponding platform specific audio device type. Below table provides default mapping of devices on a QTI's reference platform.

Applicable for SA515M, SA415M, SA410M, SA2150P based software products:

Telsdk device type	Direction	Mapped HAL device
DEVICE_TYPE_NONE	N/A	AUDIO_DEVICE_NONE
DEVICE_TYPE_SPEA↩ KER	RX	AUDIO_DEVICE_OU↩ T_SPEAKER
DEVICE_TYPE_SPEA↩ KER_2	RX	AUDIO_DEVICE_OU↩ T_EARPIECE
DEVICE_TYPE_SPEA↩ KER_3	RX	AUDIO_DEVICE_OU↩ T_WIRED_HEADSET
DEVICE_TYPE_BT_S↩ CO_SPEAKER	RX	AUDIO_DEVICE_NONE
DEVICE_TYPE_PRO↩ XY_SPEAKER	RX	AUDIO_DEVICE_OU↩ T_PROXY
DEVICE_TYPE_MIC	TX	AUDIO_DEVICE_IN_↩ BACK_MIC
DEVICE_TYPE_MIC_2	TX	AUDIO_DEVICE_IN_↩ BUILTIN_MIC
DEVICE_TYPE_MIC_3	TX	AUDIO_DEVICE_IN_↩ WIRED_HEADSET
DEVICE_TYPE_BT_S↩ CO_MIC	TX	AUDIO_DEVICE_NONE
DEVICE_TYPE_PRO↩ XY_MIC	TX	AUDIO_DEVICE_IN_↩ PROXY

Applicable for SA525M based software products:

Telsdk device type	Direction	Mapped PAL device
DEVICE_TYPE_NONE	N/A	PAL_DEVICE_NONE
DEVICE_TYPE_SPEA↩ KER	RX	PAL_DEVICE_OUT_S↩ PEAKER
DEVICE_TYPE_SPEA↩ KER_2	RX	PAL_DEVICE_OUT_H↩ ANDSET

Telsdk device type	Direction	Mapped HAL device
DEVICE_TYPE_SPEA↩ KER_3	RX	PAL_DEVICE_OUT_↩ WIRED_HEADSET
DEVICE_TYPE_BT_S↩ CO_SPEAKER	RX	PAL_DEVICE_OUT_B↩ LUETOOTH_SCO
DEVICE_TYPE_PRO↩ XY_SPEAKER	RX	PAL_DEVICE_OUT_P↩ ROXY
DEVICE_TYPE_MIC	TX	PAL_DEVICE_IN_SPE↩ AKER_MIC
DEVICE_TYPE_MIC_2	TX	PAL_DEVICE_IN_HA↩ NDSET_MIC
DEVICE_TYPE_MIC_3	TX	PAL_DEVICE_IN_WI↩ RED_HEADSET
DEVICE_TYPE_BT_S↩ CO_MIC	TX	PAL_DEVICE_IN_BL↩ UETOOTH_SCO_HEA↩ DSET
DEVICE_TYPE_PRO↩ XY_MIC	TX	PAL_DEVICE_IN_PR↩ OXY

Enumerator

DEVICE_TYPE_NONE Default device (invalid)
DEVICE_TYPE_SPEAKER Sink device as per above mapping
DEVICE_TYPE_SPEAKER_2 Sink device as per above mapping
DEVICE_TYPE_SPEAKER_3 Sink device as per above mapping
DEVICE_TYPE_BT_SCO_SPEAKER Bluetooth sink device for voice call
DEVICE_TYPE_PROXY_SPEAKER Virtual sink device as per above mapping
DEVICE_TYPE_MIC Source device as per above mapping
DEVICE_TYPE_MIC_2 Source device as per above mapping
DEVICE_TYPE_MIC_3 Source device as per above mapping
DEVICE_TYPE_BT_SCO_MIC Bluetooth source device for voice call
DEVICE_TYPE_PROXY_MIC Virtual mic connected over ethernet

4.27.2.2 enum telux::audio::DeviceDirection `[strong]`

Defines the direction of an audio device.

Enumerator

NONE Default direction (invalid)
RX Audio will go out of the device, for example through a speaker (sink)
TX Audio will come into the device, for example through a mic (source)

4.27.2.3 enum telux::audio::StreamType `[strong]`

Defines the type of the audio stream and the type's purpose.

Enumerator

NONE Default type (invalid)

VOICE_CALL Used for audio over a cellular network
PLAY Used for playing audio, for example playing music and notifications
CAPTURE Used for capturing audio, for example recording sound using a mic
LOOPBACK Used for generating audio from a DeviceDirection::RX device, which is intended to be captured back by a DeviceDirection::TX device
TONE_GENERATOR Used for single tone and DTMF tone generation

4.27.2.4 enum telux::audio::StreamDirection [strong]

Defines the direction of an audio stream.

Enumerator

NONE Default direction (invalid)
RX Specifies that the audio data will flow towards a sink device
TX Specifies that the audio data originates from a source device

4.27.2.5 enum telux::audio::AmrwbpFrameFormat [strong]

Defines the properties of the audio data for compressed playback and transcoding.

Enumerator

UNKNOWN Default format (invalid)
TRANSPORT_INTERFACE_FORMAT Unsupported
FILE_STORAGE_FORMAT Specifies that the audio content from AMR∗ format file has been parsed and only actual audio content is sent for playback

4.27.2.6 enum telux::audio::EcnrMode [strong]

On a voice call stream, enables or disables echo cancellation and noise reduction (ECNR). Used with an audio device capable of supporting ECNR.

Enumerator

DISABLE Disables ECNR
ENABLE Enables ECNR

4.27.2.7 enum telux::audio::CalibrationInitStatus [strong]

Represents the state of the platform calibration for audio.

Enumerator

UNKNOWN Default state
INIT_SUCCESS Platform calibrated successfully
INIT_FAILED Platform calibration failed

4.28 Audio Streams

This section contains APIs related to Audio Stream operation.

4.28.1 Data Structure Documentation

4.28.1.1 struct telux::audio::ChannelVolume

Defines the volume levels for a given audio channel.

Data fields

Type	Field	Description
ChannelType	channelType	ChannelType to which the volume level is associated.
float	vol	Volume level – minimum 0.0 and maximum 1.0

4.28.1.2 struct telux::audio::StreamVolume

Defines the volume levels for the audio device.

Data fields

Type	Field	Description
vector< ChannelVolume >	volume	List of the volume levels per channel, specified by ChannelVolume
Stream↩ Direction	dir	StreamDirection associated with the device

4.28.1.3 struct telux::audio::StreamMute

Specifies the mute state of the audio device.

Data fields

Type	Field	Description
bool	enable	True if the device is muted, False if the device is unmuted
Stream↩ Direction	dir	StreamDirection associated with the device

4.28.1.4 struct telux::audio::DtmfTone

Defines the characteristics of the DTMF tone.

Data fields

Type	Field	Description
DtmfLowFreq	lowFreq	Lower frequency associated with the DTMF tone
DtmfHighFreq	highFreq	Higher frequency associated with the DTMF tone

Type	Field	Description
Stream↩ Direction	direction	StreamDirection associated with the stream

4.28.1.5 class telux::audio::IVoiceListener

Listener for a DTMF tone detected event on a StreamType::VOICE_CALL stream.

Public member functions

- virtual void onDtmfToneDetection (DtmfTone dtmfTone)
- virtual ~IVoiceListener ()

4.28.1.5.1 Constructors and Destructors

4.28.1.5.1.1 virtual telux::audio::IVoiceListener::~IVoiceListener ( ) [virtual]

Destructor of the IVoiceListener.

4.28.1.5.2 Member Function Documentation

4.28.1.5.2.1 virtual void telux::audio::IVoiceListener::onDtmfToneDetection (DtmfTone *dtmfTone*) [virtual]

Called when a DTMF tone is detected on a StreamType::VOICE_CALL stream. Used in conjuction with IAudioVoiceStream::registerListener().

Parameters

in	*dtmfTone*	Contains details of the tone detected

4.28.1.6 class telux::audio::IPlayListener

Listener for events on a playback stream.

Public member functions

- virtual void onReadyForWrite ()
- virtual void onPlayStopped ()
- virtual ~IPlayListener ()

4.28.1.6.1 Constructors and Destructors

4.28.1.6.1.1 virtual telux::audio::IPlayListener::~IPlayListener ( ) [virtual]

Destructor of IPlayListener.

4.28.1.6.2 Member Function Documentation

4.28.1.6.2.1 virtual void telux::audio::IPlayListener::onReadyForWrite ( ) [virtual]

Called when the audio pipeline is ready to accept the next buffer to play during compressed playback.

4.28.1.6.2.2 virtual void telux::audio::IPlayListener::onPlayStopped ( ) [virtual]

Called when the compressed playback has stopped.

4.28.1.7 class telux::audio::IAudioBuffer

Represents the buffer containing the audio data for playback when used with the StreamType::PLAY stream. Represents the audio data received when used with the StreamType::CAPTURE stream.

Public member functions

- virtual size_t getMinSize ()=0
- virtual size_t getMaxSize ()=0
- virtual uint8_t * getRawBuffer ()=0
- virtual uint32_t getDataSize ()=0
- virtual void setDataSize (uint32_t size)=0
- virtual telux::common::Status reset ()=0
- virtual ~IAudioBuffer ()

4.28.1.7.1 Constructors and Destructors

4.28.1.7.1.1 virtual telux::audio::IAudioBuffer::~IAudioBuffer ( ) [virtual]

Destructor of the IAudioBuffer.

4.28.1.7.2 Member Function Documentation

4.28.1.7.2.1 virtual size_t telux::audio::IAudioBuffer::getMinSize ( ) [pure virtual]

For the StreamType::PLAY stream, specifies the minimum number of bytes that must be sent for playback. For the StreamType::CAPTURE stream, specifies the minimum number of bytes that can be read.

Returns

Minimum size (in bytes)

4.28.1.7.2.2 virtual size_t telux::audio::IAudioBuffer::getMaxSize ( ) [pure virtual]

For the StreamType::PLAY stream, specifies the maximum number of bytes that can be sent for playback. For the StreamType::CAPTURE stream, specifies the maximum number of bytes that can be read.

Returns

Maximum size (in bytes)

4.28.1.7.2.3 virtual uint8_t* telux::audio::IAudioBuffer::getRawBuffer ( ) [pure virtual]

Gives the managed raw buffer. It is freed when IAudioBuffer is destructed. For the StreamType::PLAY stream, the actual audio samples should be copied into this raw buffer for playback. For the StreamType::CAPTURE stream, the actual audio contents are obtained from this buffer.

Returns

Managed raw buffer

4.28.1.7.2.4 virtual uint32_t telux::audio::IAudioBuffer::getDataSize ( ) [pure virtual]

For the StreamType::CAPTURE stream, specifies how many bytes were read. Not used for the StreamType::PLAY stream.

Returns

Size of the valid data bytes in the raw buffer

4.28.1.7.2.5 virtual void telux::audio::IAudioBuffer::setDataSize (uint32_t *size*) [pure virtual]

For the StreamType::PLAY stream, specifies how many bytes should be played. Not used for the StreamType::CAPTURE stream.

Returns

Size of the valid data bytes in the raw buffer

4.28.1.7.2.6 virtual telux::common::Status telux::audio::IAudioBuffer::reset ( ) [pure virtual]

Clears the contents of the managed raw buffer.

Returns

telux::common::Status::SUCCESS if the buffer is cleared successfully, otherwise, an appropriate error code

4.28.1.8 class telux::audio::IStreamBuffer

Implements the IAudioBuffer interface to give contexual meaning to its methods based on the StreamType type associated with the stream, with which this buffer will be used.

Public member functions

- virtual ~IStreamBuffer ()

4.28.1.8.1 Constructors and Destructors

4.28.1.8.1.1 virtual telux::audio::IStreamBuffer::~IStreamBuffer ( ) [virtual]

Destructor of the IStreamBuffer.

4.28.1.9 class telux::audio::IAudioStream

Base class for all audio stream types. Contains the common properties and methods.

Public member functions

- virtual StreamType getType ()=0
- virtual telux::common::Status setDevice (std::vector< DeviceType > devices, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status getDevice (GetStreamDeviceResponseCb callback=nullptr)=0
- virtual telux::common::Status setVolume (StreamVolume volume, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status getVolume (StreamDirection dir, GetStreamVolumeResponseCb callback=nullptr)=0
- virtual telux::common::Status setMute (StreamMute mute, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status getMute (StreamDirection dir, GetStreamMuteResponseCb callback=nullptr)=0
- virtual ~IAudioStream ()

4.28.1.9.1 Constructors and Destructors

4.28.1.9.1.1 virtual telux::audio::IAudioStream::~IAudioStream ( ) [virtual]

Destructor of the IAudioStream.

4.28.1.9.2 Member Function Documentation

4.28.1.9.2.1 virtual StreamType telux::audio::IAudioStream::getType ( ) [pure virtual]

Gets the StreamType associated with the stream.

Returns

Type of the stream

4.28.1.9.2.2 virtual telux::common::Status telux::audio::IAudioStream::setDevice (std::vector< DeviceType > *devices,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

Associates the given audio device with the stream.

Applicable for StreamType::VOICE_CALL, StreamType::PLAY, and StreamType::CAPTURE only.

For StreamType::VOICE_CALL, the stream must be started using IAudioVoiceStream::startAudio() to make the device effective.

Parameters

in	*devices*	List of the audio devices to use with the stream
in	*callback*	Invoked to confirm if the device is associated

Returns

Status telux::common::Status::SUCCESS if the request is initiated successfully, otherwise, an appropriate error code

4.28.1.9.2.3 virtual telux::common::Status telux::audio::IAudioStream::getDevice (GetStreamDeviceResponseCb *callback = nullptr*) [pure virtual]

Gets the list of the audio devices associated with the stream.

Applicable for StreamType::VOICE_CALL, StreamType::PLAY, and StreamType::CAPTURE only.

Parameters

in	*callback*	Invoked to pass the associated device

Returns

Status telux::common::Status::SUCCESS if the request is initiated successfully, otherwise, an appropriate error code

4.28.1.9.2.4 virtual telux::common::Status telux::audio::IAudioStream::setVolume (StreamVolume *volume,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

Sets the volume level of the audio device.

For StreamType::VOICE_CALL, direction must be StreamDirection::RX.

Applicable for StreamType::VOICE_CALL, StreamType::PLAY, and StreamType::CAPTURE only.

Direction of the stream is ignored.

Parameters

in	*volume*	Specifies the volume level and the stream's direction
in	*callback*	Invoked to confirm if the volume level is set

Returns

Status telux::common::Status::SUCCESS if the request is initiated successfully, otherwise, an appropriate error code

4.28.1.9.2.5 virtual telux::common::Status telux::audio::IAudioStream::getVolume (StreamDirection *dir,* GetStreamVolumeResponseCb *callback = nullptr*) [pure virtual]

Gets the current volume level of the audio device.

For StreamType::VOICE_CALL, direction must be StreamDirection::RX.

Applicable for StreamType::VOICE_CALL, StreamType::PLAY, and StreamType::CAPTURE only.

Direction of the stream is ignored.

Parameters

in	*dir*	Direction of the stream associated with the device
in	*callback*	Invoked to pass the volume read

Returns

Status telux::common::Status::SUCCESS if the request is initiated successfully, otherwise, an appropriate error code

4.28.1.9.2.6 virtual telux::common::Status telux::audio::IAudioStream::setMute (StreamMute *mute,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

Mute or unmute the stream as specified by the StreamMute provided.

Applicable for StreamType::VOICE_CALL, StreamType::PLAY, and StreamType::CAPTURE only.

For StreamType::VOICE_CALL, the stream must be started using IAudioVoiceStream::startAudio() before setting the mute state.

Direction of the stream is ignored.

Parameters

in	*mute*	Defines the stream is to be muted or unmuted
in	*callback*	Invoked to confirm if the stream is muted/unmuted

Returns

Status telux::common::Status::SUCCESS if the request is initiated successfully, otherwise, an appropriate error code

4.28.1.9.2.7 virtual telux::common::Status telux::audio::IAudioStream::getMute (StreamDirection *dir*, GetStreamMuteResponseCb *callback* = `nullptr`) [pure virtual]

Gets the current mute state of the audio stream.

Applicable for StreamType::VOICE_CALL, StreamType::PLAY, and StreamType::CAPTURE only.

For StreamType::VOICE_CALL, the stream must be started using IAudioVoiceStream::startAudio() before reading the mute state.

Direction of the stream is ignored.

Parameters

in	*dir*	Direction of the stream
in	*callback*	Invoked to pass the mute state

Returns

Status telux::common::Status::SUCCESS if the request is initiated successfully, otherwise, an appropriate error code

4.28.1.10 class telux::audio::IAudioVoiceStream

Represents the stream created with the StreamType::VOICE_CALL type. Provides methods to establish a voice call on a cellular network, and play and detect DTMF tones.

Public member functions

- virtual telux::common::Status startAudio (telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status stopAudio (telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status playDtmfTone (DtmfTone dtmfTone, uint16_t duration, uint16_t gain, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status stopDtmfTone (StreamDirection direction, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status registerListener (std::weak_ptr< IVoiceListener > listener, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status deRegisterListener (std::weak_ptr< IVoiceListener > listener)=0
- virtual ~IAudioVoiceStream ()

4.28.1.10.1 Constructors and Destructors

4.28.1.10.1.1 virtual telux::audio::IAudioVoiceStream::~IAudioVoiceStream ( ) [virtual]

Destructor of the IAudioVoiceStream.

4.28.1.10.2 Member Function Documentation

4.28.1.10.2.1 virtual telux::common::Status telux::audio::IAudioVoiceStream::startAudio (telux::common::ResponseCallback *callback* = *nullptr*) [pure virtual]

Starts a voice call stream.

Parameters

in	*callback*	Optional, invoked to confirm if the stream has started

Returns

Status telux::common::Status::SUCCESS if the request is initiated successfully, otherwise, an appropriate error code

4.28.1.10.2.2 virtual telux::common::Status telux::audio::IAudioVoiceStream::stopAudio (telux::common::ResponseCallback *callback* = *nullptr*) [pure virtual]

Stops a voice call stream.

Parameters

in	*callback*	Optional, invoked to confirm if the stream has stopped

Returns

Status telux::common::Status::SUCCESS if the request is initiated successfully, otherwise, an appropriate error code

4.28.1.10.2.3 virtual telux::common::Status telux::audio::IAudioVoiceStream::playDtmfTone (DtmfTone *dtmfTone*, uint16_t *duration*, uint16_t *gain*, telux::common::ResponseCallback *callback* = *nullptr*) [pure virtual]

Generates a DTMF tone on a local device (on RX path) associated with the active voice call stream.

Parameters

in	*dtmfTone*	Specifies the tone's properties
in	*duration*	Duration (in milliseconds) for which the tone is played. Set it to INFINITE_TONE_DURATION to play indefinitely
in	*gain*	Volume level of the tone, valid value range is 0 to 4000
in	*callback*	Optional, invoked to confirm if the tone play has started

Returns

Status telux::common::Status::SUCCESS if the request is initiated successfully, otherwise, an appropriate error code

4.28.1.10.2.4 virtual telux::common::Status telux::audio::IAudioVoiceStream::stopDtmfTone (StreamDirection *direction,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

If IAudioVoiceStream::playDtmfTone() was called with the duration set to INFINITE_DTMF_DURATION, then this method stops playing the DTMF tone.

Parameters

in	*direction*	Direction of the stream
in	*callback*	Optional, invoked to confirm if the tone play has stopped

Returns

Status telux::common::Status::SUCCESS if the request is initiated successfully, otherwise, an appropriate error code

4.28.1.10.2.5 virtual telux::common::Status telux::audio::IAudioVoiceStream::registerListener (std::weak_ptr< IVoiceListener > *listener,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

Registers the given listener to get notified whenever a DTMF tone is detected on a voice call stream. Used in conjunction with IVoiceListener::onDtmfToneDetection().

Parameters

in	*listener*	Receives the DTMF tone detected event
in	*callback*	Invoked to confirm if the registration is successful

Returns

telux::common::Status::SUCCESS if the listener is registered, otherwise, an appropriate error code

4.28.1.10.2.6 virtual telux::common::Status telux::audio::IAudioVoiceStream::deRegisterListener (std::weak_ptr< IVoiceListener > *listener*) [pure virtual]

Unregisters the given listener registered with IAudioVoiceStream::registerListener().

Parameters

in	*listener*	Listener to unregister

Returns

telux::common::Status::SUCCESS if the listener is unregistered, otherwise, an appropriate error code

4.28.1.11 class telux::audio::IAudioPlayStream

Represents the stream created with the StreamType::PLAY type. Provides the methods to play the audio.

Public member functions

- virtual std::shared_ptr< IStreamBuffer > getStreamBuffer ()=0
- virtual telux::common::Status write (std::shared_ptr< IStreamBuffer > buffer, WriteResponseCb callback=nullptr)=0
- virtual telux::common::Status stopAudio (StopType stopType, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status registerListener (std::weak_ptr< IPlayListener > listener)=0
- virtual telux::common::Status deRegisterListener (std::weak_ptr< IPlayListener > listener)=0
- virtual ~IAudioPlayStream ()

4.28.1.11.1 Constructors and Destructors

4.28.1.11.1.1 virtual telux::audio::IAudioPlayStream::~IAudioPlayStream ( ) [virtual]

Destructor of the IAudioPlayStream.

4.28.1.11.2 Member Function Documentation

4.28.1.11.2.1 virtual std::shared_ptr<IStreamBuffer> telux::audio::IAudioPlayStream::getStreamBuffer ( ) [pure virtual]

Gets an audio buffer containing the audio samples to play.

Returns

IStreamBuffer instance or nullptr if memory allocation fails

4.28.1.11.2.2 virtual telux::common::Status telux::audio::IAudioPlayStream::write (std::shared_ptr< IStreamBuffer > *buffer,* WriteResponseCb *callback = nullptr*) [pure virtual]

Sends the audio data for playback. First write starts the playback operation.

For uncompressed playback (for example, AudioFormat::PCM_16BIT_SIGNED), the next buffer can be sent the moment telux::common::ErrorCode::SUCCESS is received by WriteResponseCb.

For compressed playback (for example, AudioFormat::AMR∗), the next buffer should be sent only after both; (a) telux::common::ErrorCode::SUCCESS is received by WriteResponseCb (indicating that the current buffer has been pushed in the pipeline for playback) and (b) IPlayListener::onReadyForWrite() has been invoked (indicating that the pipeline can accommodate the next buffer).

Parameters

in	*buffer*	Contains the audio data to play
in	*callback*	Optional, invoked to confirm if the data is played successfully

Returns

Status telux::common::Status::SUCCESS if the request is initiated successfully, otherwise, an appropriate error code

4.28.1.11.2.3 virtual telux::common::Status telux::audio::IAudioPlayStream::stopAudio (StopType *stopType*, telux::common::ResponseCallback *callback* = `nullptr`) `[pure virtual]`

Finishes the ongoing compressed playback in a way specified by the StopType provided.

Parameters

in	*callback*	Invoked to confirm if the playback has finished
in	*stopType*	Defines how to finish playback

Returns

Status telux::common::Status::SUCCESS if the request is initiated successfully, otherwise, an appropriate error code

4.28.1.11.2.4 virtual telux::common::Status telux::audio::IAudioPlayStream::registerListener (std::weak_ptr< IPlayListener > *listener*) `[pure virtual]`

Registers the given listener to receive events; (a) pipeline is ready to accept the next buffer for compressed playback (b) compressed playback has stopped. Events are received by the listener implementing the IPlayListener interface.

Parameters

in	*listener*	Receives the playstream events

Returns

telux::common::Status::SUCCESS if the listener is registered, otherwise, an appropriate error code

4.28.1.11.2.5 virtual telux::common::Status telux::audio::IAudioPlayStream::deRegisterListener (std::weak_ptr< IPlayListener > *listener*) `[pure virtual]`

Unregisters the given listener registered with IAudioPlayStream::registerListener().

Parameters

in	*listener*	Listener to unregister

Returns

telux::common::Status::SUCCESS if the listener is unregistered, otherwise, an appropriate error code

4.28.1.12 class telux::audio::IAudioCaptureStream

Represents the stream created with the StreamType::CAPTURE type. Provides the methods to read the captured audio.

Public member functions

- virtual std::shared_ptr< IStreamBuffer > getStreamBuffer ()=0
- virtual telux::common::Status read (std::shared_ptr< IStreamBuffer > buffer, uint32_t bytesToRead, ReadResponseCb callback=nullptr)=0
- virtual ~IAudioCaptureStream ()

4.28.1.12.1 Constructors and Destructors

4.28.1.12.1.1 virtual telux::audio::IAudioCaptureStream::~IAudioCaptureStream ( ) [virtual]

Destructor of the IAudioCaptureStream.

4.28.1.12.2 Member Function Documentation

4.28.1.12.2.1 virtual std::shared_ptr<IStreamBuffer> telux::audio::IAudioCaptureStream::getStreamBuffer ( ) [pure virtual]

Gets an audio buffer that will contain the audio data read.

Returns

IStreamBuffer instance or nullptr if memory allocation fails

4.28.1.12.2.2 virtual telux::common::Status telux::audio::IAudioCaptureStream::read (std::shared_ptr< IStreamBuffer > *buffer*, uint32_t *bytesToRead*, ReadResponseCb *callback* = *nullptr*) [pure virtual]

Read the audio data from the source device associated with this stream. Data captured will be received by the ReadResponseCb callback.

First read call starts the capture operation.

Parameters

in	*buffer*	Buffer in which data should be read
in	*bytesToRead*	Length of the data (in bytes) to read
in	*callback*	Receives the captured data

Returns

Status telux::common::Status::SUCCESS if the request is initiated successfully, otherwise, an appropriate error code

4.28.1.13 class telux::audio::IAudioLoopbackStream

Represents the stream created with the StreamType::LOOPBACK type. Provides the methods to start and stop the audio loopback operation.

Public member functions

- virtual telux::common::Status startLoopback (telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status stopLoopback (telux::common::ResponseCallback callback=nullptr)=0
- virtual ~IAudioLoopbackStream ()

4.28.1.13.1 Constructors and Destructors

4.28.1.13.1.1 virtual telux::audio::IAudioLoopbackStream::~IAudioLoopbackStream ( ) [virtual]

Destructor of the IAudioLoopbackStream.

4.28.1.13.2 Member Function Documentation

4.28.1.13.2.1 virtual telux::common::Status telux::audio::IAudioLoopbackStream::startLoopback (telux::common::ResponseCallback *callback* = `nullptr`) [pure virtual]

Starts looping back the audio between the source and sink devices associated with this stream.

Parameters

in	*callback*	Invoked to confirm if the loopback has started

Returns

Status telux::common::Status::SUCCESS if the request is initiated successfully, otherwise, an appropriate error code

4.28.1.13.2.2 virtual telux::common::Status telux::audio::IAudioLoopbackStream::stopLoopback (telux::common::ResponseCallback *callback* = `nullptr`) [pure virtual]

Starts looping back the audio between the source and sink devices associated with this stream.

Parameters

in	*callback*	Optional, invoked to confirm if the loopback has stopped

Returns

Status telux::common::Status::SUCCESS if the request is initiated successfully, otherwise, an appropriate error code

4.28.1.14 class telux::audio::IAudioToneGeneratorStream

Represents the stream created with the StreamType::TONE_GENERATOR type. Provides the methods to play an audio tone.

Public member functions

- virtual telux::common::Status playTone (std::vector< uint16_t > freq, uint16_t duration, uint16_t gain, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status stopTone (telux::common::ResponseCallback callback=nullptr)=0
- virtual ~IAudioToneGeneratorStream ()

4.28.1.14.1 Constructors and Destructors

4.28.1.14.1.1 virtual telux::audio::IAudioToneGeneratorStream::~IAudioToneGeneratorStream ( ) [virtual]

Destructor of the IAudioToneGeneratorStream.

4.28.1.14.2 Member Function Documentation

4.28.1.14.2.1 virtual telux::common::Status telux::audio::IAudioToneGeneratorStream::playTone (std::vector< uint16_t > *freq*, uint16_t *duration*, uint16_t *gain*, telux::common::ResponseCallback *callback* = `nullptr`) [pure virtual]

Plays an audio tone with the given parameters.

Parameters

in	*freq*	Frequency of the tone. For single tone, freq[0] should be provided. For dual tone, both freq[0] and freq[1] should be provided.
in	*duration*	Duration (in milliseconds) for which the tone is played. Set it to INFINITE_TONE_DURATION to play indefinitely
in	*gain*	Defines the volume level of the tone, valid value range is 0 to 4000
in	*callback*	Optional, invoked to confirm if the tone play started

Returns

Status telux::common::Status::SUCCESS if the request is initiated successfully, otherwise, an appropriate error code

4.28.1.14.2.2 virtual telux::common::Status telux::audio::IAudioToneGeneratorStream::stopTone (telux::common::ResponseCallback *callback* = `nullptr`) [pure virtual]

If the IAudioToneGeneratorStream::playTone() was called with the INFINITE_TONE_DURATION duration, then this method stops playing the tone.

Parameters

in	*callback*	Optional, invoked to confirm if the tone play has stopped

Returns

Status telux::common::Status::SUCCESS if the request is initiated successfully, otherwise, an appropriate error code

4.28.2 Enumeration Type Documentation

4.28.2.1 enum telux::audio::Direction [strong]

Used for an in-call audio usecase. Represents the direction of the audio data flow.

Enumerator

RX Defines that playback should occur on a voice downlink path (cellular network to a device)
TX Defines that playback should occur on voice uplink path (device to a cellular network)

4.28.2.2 enum telux::audio::ChannelType

Adds positional perspective to the audio data in a given audio frame. For example, in a 2-speaker audio system, ChannelType::LEFT may represent audio played on speaker-1 while ChannelType::RIGHT represents audio played on speaker-2.

Enumerator

LEFT Specifies the left channel
RIGHT Specifies the right channel

4.28.2.3 enum telux::audio::AudioFormat [strong]

Specifies how audio data is represented (for example, endianness and number of bits) for storage or exchanging among various audio software and hardware layers.

Enumerator

UNKNOWN Default format (invalid)
PCM_16BIT_SIGNED PCM signed 16 bits
AMRNB Adaptive multirate narrow band format
AMRWB Adaptive multirate wide band format
AMRWB_PLUS Extended adaptive multirate wide band format

4.28.2.4 enum telux::audio::DtmfLowFreq [strong]

When generating a DTMF tone, defines the value of the low frequency component.

Enumerator

FREQ_697 697 Hz
FREQ_770 770 Hz
FREQ_852 852 Hz
FREQ_941 941 Hz

4.28.2.5 enum telux::audio::DtmfHighFreq [strong]

When generating a DTMF tone, defines the value of the high frequency component.

Enumerator

FREQ_1209 1209 Hz
FREQ_1336 1336 Hz
FREQ_1477 1477 Hz
FREQ_1633 1633 Hz

4.28.2.6 enum telux::audio::StopType [strong]

Defines the behavior for how a compressed audio format playback should be finished.

Enumerator

FORCE_STOP Stop playing immediately and discard all pending audio samples
STOP_AFTER_PLAY Stop playing after all samples in the pipeline have been played

4.29 Transcoder

This section contains APIs related to Audio Transcoder operation.

4.29.1 Data Structure Documentation

4.29.1.1 class telux::audio::ITranscodeListener

Listener for events during transcoding.

Public member functions

- virtual void onReadyForWrite ()
- virtual ~ITranscodeListener ()

4.29.1.1.1 Constructors and Destructors

4.29.1.1.1.1 virtual telux::audio::ITranscodeListener::~ITranscodeListener ( ) [virtual]

Destructor of ITranscodeListener.

4.29.1.1.2 Member Function Documentation

4.29.1.1.2.1 virtual void telux::audio::ITranscodeListener::onReadyForWrite ( ) [virtual]

Called when the audio pipeline is ready to accept the next buffer containing data to transcode.

4.29.1.2 class telux::audio::ITranscoder

Provides the methods for transcoding the compressed audio data.

Public member functions

- virtual std::shared_ptr< IAudioBuffer > getWriteBuffer ()=0
- virtual std::shared_ptr< IAudioBuffer > getReadBuffer ()=0
- virtual telux::common::Status write (std::shared_ptr< IAudioBuffer > buffer, uint32_t isLastBuffer, TranscoderWriteResponseCb callback=nullptr)=0
- virtual telux::common::Status tearDown (telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status read (std::shared_ptr< IAudioBuffer > buffer, uint32_t bytesToRead, TranscoderReadResponseCb callback=nullptr)=0
- virtual telux::common::Status registerListener (std::weak_ptr< ITranscodeListener > listener)=0
- virtual telux::common::Status deRegisterListener (std::weak_ptr< ITranscodeListener > listener)=0
- virtual ~ITranscoder ()

4.29.1.2.1 Constructors and Destructors

4.29.1.2.1.1 virtual telux::audio::ITranscoder::~ITranscoder ( ) [virtual]

Destructor of the ITranscoder.

4.29.1.2.2 Member Function Documentation

4.29.1.2.2.1 virtual std::shared_ptr<IAudioBuffer> telux::audio::ITranscoder::getWriteBuffer ( ) [pure virtual]

Gets a buffer for sending the data for transcoding.

Returns

IAudioBuffer instance representing the buffer or nullptr if allocation failed

4.29.1.2.2.2 virtual std::shared_ptr<IAudioBuffer> telux::audio::ITranscoder::getReadBuffer ( ) [pure virtual]

Gets a buffer that will contain the transcoded data.

Returns

IAudioBuffer instance representing the buffer or nullptr if allocation failed

4.29.1.2.2.3 virtual telux::common::Status telux::audio::ITranscoder::write (std::shared_ptr< IAudioBuffer > *buffer,* uint32_t *isLastBuffer,* TranscoderWriteResponseCb *callback = nullptr*) [pure virtual]

Sends the compressed data for transcoding. First write starts the transcoding operation.

Internally, a pipeline is maintained for the data to transcode. The application should send the next data for transcoding only when the pipeline can accomodate more data. This readiness is indicated by calling the ITranscodeListener::onReadyForWrite() method.

Parameters

in	*buffer*	Contains the data to transcode
in	*isLastBuffer*	Marks that this is the last chunk of the data to transcode
in	*callback*	Optional, invoked to pass the status of pushing the data in the pipeline

Returns

telux::common::Status::SUCCESS if the data is sent, otherwise, an appropriate error code

4.29.1.2.2.4 virtual telux::common::Status telux::audio::ITranscoder::tearDown (telux::common::↩ ResponseCallback *callback = nullptr*) [pure virtual]

Destroys the ITranscoder instance created with IAudioManager::createTranscoder(). This must be called after the transcoding is finished.

Parameters

in	*callback*	Optional, invoked to pass the result of the destruction

Returns

telux::common::Status::SUCCESS if the teardown was initiated, otherwise, an appropriate error code

4.29.1.2.2.5 virtual telux::common::Status telux::audio::ITranscoder::read (std::shared_ptr< IAudioBuffer > *buffer,* uint32_t *bytesToRead,* TranscoderReadResponseCb *callback = nullptr*) [pure virtual]

Initiates a read request to fetch the transcoded data. Transcoded data will be by the TranscoderReadResponseCb callback.

Parameters

in	*buffer*	Buffer that will contain the transcoded data
in	*bytesToRead*	Length of the data to fetch
in	*callback*	Optional, invoked to pass the transcoded data

Returns

telux::common::Status::SUCCESS if the request is sent, otherwise, an appropriate error code

4.29.1.2.2.6 virtual telux::common::Status telux::audio::ITranscoder::registerListener (std::weak_ptr< ITranscodeListener > *listener*) [pure virtual]

Registers the given listener to know 'when the pipeline is ready to accept the next buffer' for transcoding. Event is received by the ITranscodeListener::onReadyForWrite() method.

Parameters

in	*listener*	Receives the events during transcoding

Returns

telux::common::Status::SUCCESS if the listener is registered, otherwise, an appropriate error code

4.29.1.2.2.7 virtual telux::common::Status telux::audio::ITranscoder::deRegisterListener (std::weak_ptr< ITranscodeListener > *listener*) [pure virtual]

Unregisters the given listener registered with ITranscoder::registerListener().

Parameters

`in`	*listener*	Listener to unregister

Returns

telux::common::Status::SUCCESS if the listener is unregistered, otherwise, an appropriate error code

4.30 Thermal

- Thermal Management
- Thermal Shutdown Management

This section contains APIs related to thermal.

4.31 Thermal Management

This section contains APIs related to Thermal Management such as read list of thermal zones, cooling devices and binding info.

4.31.1 Data Structure Documentation

4.31.1.1 class telux::therm::ThermalFactory

ThermalFactory allows creation of thermal manager.

Public member functions

- virtual std::shared_ptr< IThermalManager > getThermalManager (telux::common::InitResponseCb callback=nullptr, telux::common::ProcType operType=telux::common::ProcType::LOCAL_PROC)=0
- virtual std::shared_ptr< IThermalShutdownManager > getThermalShutdownManager (telux::common::InitResponseCb callback=nullptr)=0

Static Public Member Functions

- static ThermalFactory & getInstance ()

4.31.1.1.1 Member Function Documentation

4.31.1.1.1.1 static ThermalFactory& telux::therm::ThermalFactory::getInstance ( ) [static]

Get Thermal Factory instance.

4.31.1.1.1.2 virtual std::shared_ptr<IThermalManager> telux::therm::ThermalFactory::getThermal↩ Manager (telux::common::InitResponseCb *callback* = `nullptr`, telux::common::ProcType *operType* = telux::common::ProcType::LOCAL_PROC) [pure virtual]

Get thermal manager instance associated with a telux::common::ProcType to get list of thermal zones (sensors) and cooling devices supported by the device

On platforms with Access control enabled, Caller needs to have TELUX_THERM_DATA_READ permission to invoke this API successfully.

Parameters

in	*callback*	Optional callback pointer to get the response of the manager initialization.
in	*oprType*	Operation type telux::common::ProcType. Local operation type fetches the thermal zones information where the application is running. Remote operation type fetches the thermal zones information of modem if the application is running on external application processor(EAP) and vice versa.

Returns

Pointer of IThermalManager object.

4.31.1.1.1.3 virtual std::shared_ptr<IThermalShutdownManager> telux::therm::ThermalFactory::getThermalShutdownManager (telux::common::InitResponseCb *callback = nullptr*) [pure virtual]

Get thermal shutdown manager instance to control automatic thermal shutdown and get relevant notifications

On platforms with Access control enabled, Caller needs to have TELUX_THERM_SHUTDOWN_CTRL permission to invoke this API successfully.

Parameters

in	*callback*	Optional callback pointer to get the response of the manager initialization.

Returns

Pointer of IThermalShutdownManager object.

4.31.1.2 class telux::therm::IThermalListener

Listener class for getting notifications when thermal service status changes. The client needs to implement these methods as briefly as possible and avoid blocking calls in it. The methods in this class can be invoked from multiple different threads. Client needs to make sure that the implementation is thread-safe.

Public member functions

- virtual ~IThermalListener ()
- virtual void onCoolingDeviceLevelChange (std::shared_ptr< ICoolingDevice > coolingDevice)
- virtual void onTripEvent (std::shared_ptr< ITripPoint > tripPoint, TripEvent tripEvent)

4.31.1.2.1 Constructors and Destructors

4.31.1.2.1.1 virtual telux::therm::IThermalListener::~IThermalListener ( ) [virtual]

Destructor of IThermalListener

4.31.1.2.2 Member Function Documentation

4.31.1.2.2.1 virtual void telux::therm::IThermalListener::onCoolingDeviceLevelChange (std::shared_ptr< ICoolingDevice > *coolingDevice*) [virtual]

This function is called at the time of cooling device level update. On platforms with Access control enabled, the client needs to have TELUX_THERM_DATA_READ permission to receive this event.

Parameters

in	*coolingDevice*	- vector of cooling device for which the level has been updated.

4.31.1.2.2.2 virtual void telux::therm::IThermalListener::onTripEvent (std::shared_ptr< ITripPoint > *tripPoint,* TripEvent *tripEvent*) [virtual]

This function is called at the time of trip event occurs. On platforms with Access control enabled, the client needs to have TELUX_THERM_DATA_READ permission to receive this event.

Parameters

in	*tripInfo*	- Vector of the trip point for which trip event has been occured.
in	*tripEvent*	- Indicates trip event. • NONE • CROSSED_UNDER • CROSSED_OVER

4.31.1.3 struct telux::therm::BoundCoolingDevice

Defines the trip points to which cooling device is bound.

Data fields

Type	Field	Description
int	cooling↩ DeviceId	Cooling device Id associated with trip points
vector< shared_ptr< ITripPoint > >	bindingInfo	List of trippoints bound to the cooling device

4.31.1.4 class telux::therm::IThermalManager

IThermalManager provides interface to get thermal zone and cooling device information.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual telux::common::Status registerListener (std::weak_ptr< IThermalListener > listener, ThermalNotificationMask mask=0xFFFF)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< IThermalListener > listener, ThermalNotificationMask mask=0xFFFF)=0
- virtual std::vector< std::shared_ptr< IThermalZone > > getThermalZones ()=0
- virtual std::vector< std::shared_ptr< ICoolingDevice > > getCoolingDevices ()=0
- virtual std::shared_ptr< IThermalZone > getThermalZone (int thermalZoneId)=0

- virtual std::shared_ptr< ICoolingDevice > getCoolingDevice (int coolingDeviceId)=0
- virtual ~IThermalManager ()

4.31.1.4.1 Constructors and Destructors

4.31.1.4.1.1 virtual telux::therm::IThermalManager::~IThermalManager ( ) [virtual]

Destructor of IThermalManager

4.31.1.4.2 Member Function Documentation

4.31.1.4.2.1 virtual telux::common::ServiceStatus telux::therm::IThermalManager::getServiceStatus ( ) [pure virtual]

This status indicates whether the object is in a usable state.

Returns

telux::common::ServiceStatus

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.31.1.4.2.2 virtual telux::common::Status telux::therm::IThermalManager::registerListener (std::weak_ptr< IThermalListener > *listener,* ThermalNotificationMask *mask = 0xFFFF*) [pure virtual]

Registers the listener for Thermal Manager indications.

Parameters

in	*listener*	- pointer to implemented listener.
in	*mask*	- Bit mask representing a set of notifications that needs to be registered - ThermalNotificationType Notifications under IThermalListener that are not listed in ThermalNotificationType would always be registered by default when this API is invoked. In the absence of this optional parameter, all the notifications will be registered. Bits that are not set in the mask are ignored and do not have any effect on registration or deregistration. To deregister, the API deregisterListener should be used. For Example: API invoked with mask: 0x0001 enables onTripEvent notification, next invocation with mask: 0x0002 enables onCoolingDeviceLevelUpdate notification and previous registration for onTripEvent remains intact.

Returns

status of the registration request.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.31.1.4.2.3 virtual telux::common::Status telux::therm::IThermalManager::deregisterListener (std::weak_ptr< IThermalListener > *listener,* ThermalNotificationMask *mask = 0xFFFF*) [pure virtual]

Deregisters the previously registered listener.

Parameters

in	*listener*	- pointer to registered listener that needs to be removed.
in	*mask*	- Bit mask that denotes a set of notifications that needs to be de-registered - ThermalNotificationType Notifications under IThermalListener that are not listed in ThermalNotificationType would not be de-registered by default. If the client does not specifies mask or sets all the bits, this API de-registers all the notifications. Bits that are not set in the mask are ignored and do not have any effect on registration or deregistration,To register, the API registerListener should be used. For Example: API invoked with mask: 0x0001 disables onTripEvent notification, next invocation with mask: 0x0002 disables onCoolingDeviceLevelUpdate notification. mask: 0x0000 is invalid options and API invoked with mask 0x0000 will be ignored.

Returns

status of the deregistration request.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.31.1.4.2.4 virtual std::vector<std::shared_ptr<IThermalZone> > telux::therm::IThermalManager::getThermalZones ( ) [pure virtual]

Retrieves the list of thermal zone info like type, temperature and trip points.

Returns

List of thermal zones.

4.31.1.4.2.5 virtual std::vector<std::shared_ptr<ICoolingDevice> > telux::therm::IThermalManager::getCoolingDevices ( ) [pure virtual]

Retrieves the list of thermal cooling device info like type, maximum throttle state and currently requested throttle state.

Returns

List of cooling devices.

4.31.1.4.2.6 virtual std::shared_ptr<IThermalZone> telux::therm::IThermalManager::getThermalZone (int *thermalZoneId*) [pure virtual]

Retrieves the thermal zone details like temperature, type and trip point info for the given thermal zone identifier.

Parameters

in	*thermalZoneId*	Thermal zone identifier

Returns

Pointer to thermal zone.

4.31.1.4.2.7 virtual std::shared_ptr<ICoolingDevice> telux::therm::IThermalManager::getCoolingDevice (int *coolingDeviceId*) [pure virtual]

Retrieves the cooling device details like type of the device, maximum cooling level and current cooling level for the given cooling device identifier.

Parameters

in	*coolingDeviceId*	Cooling device identifier

Returns

Pointer to cooling device.

4.31.1.5 class telux::therm::ITripPoint

ITripPoint provides interface to get trip point type, trip point temperature and hysteresis value for that trip point.

Public member functions

- virtual TripType getType () const =0
- virtual int getThresholdTemp () const =0
- virtual int getHysteresis () const =0

- virtual int getTripId () const =0
- virtual int getTZoneId () const =0
- virtual bool operator== (const ITripPoint &rHs) const =0
- virtual ~ITripPoint ()

4.31.1.5.1 Constructors and Destructors

4.31.1.5.1.1 virtual telux::therm::ITripPoint::~ITripPoint ( ) [virtual]

Destructor of ITripPoint

4.31.1.5.2 Member Function Documentation

4.31.1.5.2.1 virtual TripType telux::therm::ITripPoint::getType ( ) const [pure virtual]

Retrieves trip point type.

Returns

Type of trip point if available else return UNKNOWN.

- TripType

4.31.1.5.2.2 virtual int telux::therm::ITripPoint::getThresholdTemp ( ) const [pure virtual]

Retrieves the temperature above which certain trip point will be fired.

- Units: MilliDegree Celsius

Returns

Threshold temperature

4.31.1.5.2.3 virtual int telux::therm::ITripPoint::getHysteresis ( ) const [pure virtual]

Retrieves hysteresis value that is the difference between current temperature of the device and the temperature above which certain trip point will be fired. Units: MilliDegree Celsius

Returns

Hysteresis value

4.31.1.5.2.4 virtual int telux::therm::ITripPoint::getTripId ( ) const [pure virtual]

Retrieves the identifier for trip point.

Returns

Identifier for trip point

4.31.1.5.2.5 virtual int telux::therm::ITripPoint::getTZoneId ( ) const [pure virtual]

Retrieves associated tzone id for a trip point.

Returns

Identifier for thermal zone

4.31.1.5.2.6 virtual bool telux::therm::ITripPoint::operator== (const ITripPoint & *rHs*) const [pure virtual]

Operator for compare two trip points

Returns

result of two trip points whether equal or not equal.

4.31.1.6 class telux::therm::IThermalZone

IThermalZone provides interface to get type of the sensor, the current temperature reading, trip points and the cooling devices binded etc.

Public member functions

- virtual int getId () const =0
- virtual std::string getDescription () const =0
- virtual int getCurrentTemp () const =0
- virtual int getPassiveTemp () const =0
- virtual std::vector< std::shared_ptr< ITripPoint > > getTripPoints () const =0
- virtual std::vector< BoundCoolingDevice > getBoundCoolingDevices () const =0
- virtual ~IThermalZone ()

4.31.1.6.1 Constructors and Destructors

4.31.1.6.1.1 virtual telux::therm::IThermalZone::~IThermalZone ( ) [virtual]

Destructor of IThermalZone

4.31.1.6.2 Member Function Documentation

4.31.1.6.2.1 virtual int telux::therm::IThermalZone::getId ( ) const [pure virtual]

Retrieves the identifier for thermal zone.

Returns

Identifier for thermal zone

4.31.1.6.2.2 virtual std::string telux::therm::IThermalZone::getDescription ( ) const [pure virtual]

Retrieves the type of sensor.

Returns

Sensor type

4.31.1.6.2.3 virtual int telux::therm::IThermalZone::getCurrentTemp ( ) const [pure virtual]

Retrieves the current temperature of the device. Units: MilliDegree Celsius

Returns

Current temperature

4.31.1.6.2.4 virtual int telux::therm::IThermalZone::getPassiveTemp ( ) const [pure virtual]

Retrieves the temperature of passive trip point for the zone. Default value is 0. Valid values: 0 (disabled) or greater than 1000 (enabled), Units: MilliDegree Celsius

Returns

Temperature of passive trip point

4.31.1.6.2.5 virtual std::vector<std::shared_ptr<ITripPoint> > telux::therm::IThermalZone::getTrip↩ Points ( ) const [pure virtual]

Retrieves trip point information like trip type, trip temperature and hysteresis.

Returns

Trip point info list

4.31.1.6.2.6 virtual std::vector<BoundCoolingDevice> telux::therm::IThermalZone::getBoundCoolingDevices ( ) const [pure virtual]

Retrieves the list of cooling device and the associated trip points bound to cooling device in given thermal zone.

Returns

List of bound cooling device for the given thermal zone.

4.31.1.7 class telux::therm::ICoolingDevice

ICoolingDevice provides interface to get type of the cooling device, the maximum throttle state and the currently requested throttle state of the cooling device.

Public member functions

- virtual int getId () const =0
- virtual std::string getDescription () const =0
- virtual int getMaxCoolingLevel () const =0
- virtual int getCurrentCoolingLevel () const =0
- virtual ~ICoolingDevice ()

4.31.1.7.1 Constructors and Destructors

4.31.1.7.1.1 virtual telux::therm::ICoolingDevice::~ICoolingDevice ( ) [virtual]

Destructor of ICoolingDevice

4.31.1.7.2 Member Function Documentation

4.31.1.7.2.1 virtual int telux::therm::ICoolingDevice::getId ( ) const [pure virtual]

Retrieves the identifier of the thermal cooling device.

Returns

Cooling device identifier

4.31.1.7.2.2 virtual std::string telux::therm::ICoolingDevice::getDescription ( ) const [pure virtual]

Retrieves the type of the cooling device.

Returns

Cooling device type

4.31.1.7.2.3 virtual int telux::therm::ICoolingDevice::getMaxCoolingLevel ( ) const [pure virtual]

Retrieves the maximum cooling level of the cooling device.

Returns

Maximum cooling level of the thermal cooling device

4.31.1.7.2.4 virtual int telux::therm::ICoolingDevice::getCurrentCoolingLevel ( ) const [pure virtual]

Retrieves the current cooling level of the cooling device. This value can be between 0 and max cooling level. Max cooling level is different for different cooling devices like fan, processor etc.

Returns

Current cooling level of the thermal cooling device

4.31.2 Enumeration Type Documentation

4.31.2.1 enum telux::therm::AutoShutdownMode [strong]

Defines the status of automatic thermal shutdown

Enumerator

UNKNOWN Automatic thermal shutdown status is unknown
ENABLE Automatic thermal shutdown is enabled
DISABLE Automatic thermal shutdown is disabled

4.31.2.2 enum telux::therm::TripType [strong]

Defines the type of trip points, it can be one of the values for ACPI (Advanced Configuration and Power Interface) thermal zone

Enumerator

UNKNOWN Trip type is unknown
CRITICAL Trip point at which system shuts down
HOT Trip point to notify emergency
PASSIVE Trip point at which kernel lowers the CPU's frequency and throttle the processor down

ACTIVE Trip point at which processor fan turns on

CONFIGURABLE_HIGH Triggering threshold at which mitigation starts. This type is added to support legacy targets

CONFIGURABLE_LOW Clearing threshold at which mitigation stops. This type is added to support legacy targets

4.31.2.3 enum telux::therm::TripEvent `[strong]`

Defines the event of trip.

Enumerator

NONE Trip event is none

CROSSED_UNDER This event will be triggered when the temperature decreases and crosses below the configured trip minus hysteresis temp. This event will not be triggered again, if the temperature remains below the trip temperature. For Example: Below scenario considered as CROSSED_UNDER. Prev temp: 27000 milli degree Celsius, Trip temp: 25000 milli degree Celsius, Hyst: 5000 milli degree Celsius, Curr Temp: 19000 milli degree Celsius, Below scenario will not generate CROSSED_UNDER event again. Prev temp: 19000 milli degree Celsius, Trip temp: 25000 milli degree Celsius, Hyst: 5000 milli degree Celsius, Curr Temp: 18000 milli degree Celsius / 22000 milli degree Celsius

CROSSED_OVER This event will be triggered when the temperature increases and crosses over the configured trip temperature. This event will not be triggered again, if the temperature remains over the trip temperature. For Example: Below scenario considered as CROSSED_OVER. Prev temp: 24000 milli degree Celsius, Trip temp: 25000 milli degree Celsius, Curr Temp: 26000 milli degree Celsius, Below scenario will not generate CROSSED_OVER event again. Prev temp: 26000 milli degree Celsius, Trip temp: 25000 milli degree Celsius, Curr Temp: 27000 milli degree Celsius

4.31.2.4 enum telux::therm::ThermalNotificationType

Defines some of the notifications supported by IThermalListener which can be dynamically disabled/enabled.

Enumerator

TNT_TRIP_UPDATE
TNT_CDEV_LEVEL_UPDATE
TNT_MAX_TYPE

4.31.3 Variable Documentation

4.31.3.1 const uint32_t telux::therm::DEFAULT_TIMEOUT = 30

Default time out (in seconds) for thermal auto-shutdown service to re-enable thermal auto-shutdown.

4.32 Thermal Shutdown Management

This section contains APIs related to Thermal Shutdown Management such as set/get thermal auto-shutdown mode, receive notifications on every auto-shutdown update.

4.32.1 Data Structure Documentation

4.32.1.1 class telux::therm::IThermalShutdownListener

Listener class for getting notifications when automatic thermal shutdown mode is enabled/ disabled or will be enabled imminently. The client needs to implement these methods as briefly as possible and avoid blocking calls in it. The methods in this class can be invoked from multiple different threads. Client needs to make sure that the implementation is thread-safe.

Public member functions

- virtual void onShutdownEnabled ()
- virtual void onShutdownDisabled ()
- virtual void onImminentShutdownEnablement (uint32_t imminentDuration)
- virtual ~IThermalShutdownListener ()

4.32.1.1.1 Constructors and Destructors

4.32.1.1.1.1 virtual telux::therm::IThermalShutdownListener::~IThermalShutdownListener ( ) [virtual]

Destructor of IThermalShutdownListener

4.32.1.1.2 Member Function Documentation

4.32.1.1.2.1 virtual void telux::therm::IThermalShutdownListener::onShutdownEnabled ( ) [virtual]

This function is called when the automatic shutdown mode changes to ENABLE

4.32.1.1.2.2 virtual void telux::therm::IThermalShutdownListener::onShutdownDisabled ( ) [virtual]

This function is called when the automatic shutdown mode changes to DISABLE

4.32.1.1.2.3 virtual void telux::therm::IThermalShutdownListener::onImminentShutdownEnablement (uint32_t *imminentDuration*) [virtual]

This function is called when the automatic shutdown mode is about to change to ENABLE. Clients that want to keep the shutdown mode disabled, needs to set it accordingly with in the imminentDuration time. If disabled successfully within imminentDuration time, the system timer for auto-enablement will be reset.

Parameters

in	*imminentDuration*	Time elapsed(in seconds) for the shutdown mode to be enabled

4.32.1.2 class telux::therm::IThermalShutdownManager

IThermalShutdownManager class provides interface to enable/disable automatic thermal shutdown. Additionally it facilitates to register for notifications when the automatic shutdown mode changes.

Public member functions

- virtual bool isReady ()=0
- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual std::future< bool > onReady ()=0
- virtual telux::common::Status registerListener (std::weak_ptr< IThermalShutdownListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< IThermalShutdownListener > listener)=0
- virtual telux::common::Status setAutoShutdownMode (AutoShutdownMode mode, telux::common::ResponseCallback callback=nullptr, uint32_t timeout=DEFAULT_TIMEOUT)=0
- virtual telux::common::Status getAutoShutdownMode (GetAutoShutdownModeResponseCb callback)=0
- virtual ~IThermalShutdownManager ()

4.32.1.2.1 Constructors and Destructors

4.32.1.2.1.1 virtual telux::therm::IThermalShutdownManager::~IThermalShutdownManager ( ) [virtual]

Destructor of IThermalShutdownManager

4.32.1.2.2 Member Function Documentation

4.32.1.2.2.1 virtual bool telux::therm::IThermalShutdownManager::isReady ( ) [pure virtual]

Checks the status of thermal shutdown management service and if the other APIs are ready for use and returns the result.

Returns

True if the services are ready otherwise false.

Deprecated

use getServiceStatus()

4.32.1.2.2.2 virtual telux::common::ServiceStatus telux::therm::IThermalShutdownManager::getServiceStatus ( ) [pure virtual]

This status indicates whether the object is in a usable state.

Returns

telux::common::ServiceStatus

4.32.1.2.2.3 virtual std::future<bool> telux::therm::IThermalShutdownManager::onReady ( ) [pure virtual]

Wait for thermal shutdown management service to be ready.

Returns

A future that caller can wait on to be notified when thermal shutdown management service is ready.

Deprecated

The callback mechanism introduced in the ThermalFactory::getThermalShutdownManager with initialization callback along with getServiceStatus API will provide the similar mechanism as onReady and isReady. This API will soon be removed from further releases.

4.32.1.2.2.4 virtual telux::common::Status telux::therm::IThermalShutdownManager::registerListener (std::weak_ptr< IThermalShutdownListener > *listener*) [pure virtual]

Register a listener for updates on automatic shutdown mode changes

Parameters

in	*listener*	Pointer of IThermalShutdownListener object that processes the notification

Returns

Status of registerListener i.e success or suitable status code.

4.32.1.2.2.5 virtual telux::common::Status telux::therm::IThermalShutdownManager::deregisterListener (std::weak_ptr< IThermalShutdownListener > *listener*) [pure virtual]

Remove a previously registered listener.

Parameters

in	*listener*	Previously registered IThermalShutdownListener that needs to be removed

Returns

Status of deregisterListener, success or suitable status code

4.32.1.2.2.6 virtual telux::common::Status telux::therm::IThermalShutdownManager::setAuto↩ ShutdownMode (AutoShutdownMode *mode,* telux::common::ResponseCallback *callback = nullptr,* uint32_t *timeout* = DEFAULT_TIMEOUT) [pure virtual]

Set automatic thermal shutdown mode. When set to DISABLE mode successfully, it remains in DISABLE mode briefly and automatically changes to ENABLE mode after notifying the clients.

Parameters

in	*mode*	desired AutoShutdownMode to be set
in	*callback*	Optional callback to get the response of the command
in	*timeout*	Optional timeout(in seconds) for which auto-shutdown remains disabled.

Returns

Status of setAutoShutdownMode i.e. success or suitable status code.

4.32.1.2.2.7 virtual telux::common::Status telux::therm::IThermalShutdownManager::get↩ AutoShutdownMode (GetAutoShutdownModeResponseCb *callback*) [pure virtual]

Get automatic thermal shutdown mode.

Parameters

in	*callback*	GetAutoShutdownModeResponseCb to get response of the request

Returns

Status of getAutoShutdownMode i.e. success or suitable status code.

4.33 Power

- TCU Activity Manager

This section contains APIs related to power.

4.34 TCU Activity Manager

This section contains APIs related to TCU activity state management.

4.34.1 Data Structure Documentation

4.34.1.1 class telux::power::PowerFactory

PowerFactory allows creation of TCU-activity manager instance.

Public member functions

- virtual std::shared_ptr< ITcuActivityManager > getTcuActivityManager (ClientInstanceConfig config, telux::common::InitResponseCb callback=nullptr)=0
- virtual std::shared_ptr< ITcuActivityManager > getTcuActivityManager (ClientType clientType=ClientType::SLAVE, common::ProcType procType=common::ProcType::LOCAL_PROC, telux::common::InitResponseCb callback=nullptr)=0

Static Public Member Functions

- static PowerFactory & getInstance ()

4.34.1.1.1 Member Function Documentation

4.34.1.1.1.1 static PowerFactory& telux::power::PowerFactory::getInstance ( ) [static]

API to get the factory instance for TCU-activity management

4.34.1.1.1.2 virtual std::shared_ptr<ITcuActivityManager> telux::power::PowerFactory::getTcuActivityManager (ClientInstanceConfig *config,* telux::common::InitResponseCb *callback = nullptr*) [pure virtual]

Gets the TCU-activity manager instance.

Parameters

in	*config*	TCU-activity manager configuration
in	*callback*	Optional callback pointer to get the response of the manager initialization.

Returns

Pointer to ITcuActivityManager object.

Note

This API is recommended for both hypervisor and non-hypervisor based systems.
Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.34.1.1.1.3 virtual std::shared_ptr<ITcuActivityManager> telux::power::PowerFactory::getTcu↩ ActivityManager (ClientType *clientType* = ClientType::SLAVE, common::ProcType *procType* = common::ProcType::LOCAL_PROC, telux::common::InitResponseCb *callback* = *nullptr*) [pure virtual]

Gets the TCU-activity manager instance.

Parameters

in	*clientType*	Type of the client that is going to access ITcuActivityManager APIs ClientType
in	*procType*	Required processor type on which the operations will be performed telux::common::ProcType telux::common::ProcType::REMOTE_PROC is not supported
in	*callback*	Optional callback pointer to get the response of the manager initialization.

Returns

Pointer of ITcuActivityManager object.

Note

This API cannot be used on virtual machines or on systems with hypervisor. The alternative API PowerFactory::getTcuActivityManager(ClientInstanceConfig config,telux::common::InitResponseCb callback) should be used.

Deprecated

Use PowerFactory::getTcuActivityManager(ClientInstanceConfig config, telux::common::InitResponseCb callback) API instead

Type	Field	Description

4.34.1.2 struct telux::power::ClientInstanceConfig

TCU-activity Manager configuration

Data fields

Type	Field	Description
ClientType	clientType	Type of the client that is going to access ITcuActivityManager APIs ClientType. There will be a single ClientType::MASTER across all available machines.
string	clientName	Identifies the client that is retrieving an instance of the TcuActivityManager. This is a mandatory field and it needs to be unique across all TcuActivityManager clients across all machines in the system. To make it unique, one example could be machineName_ProcessName_ProcessId. This field will be used to provide a list of client names to the master via ITcuActivityListener::onSlaveAckStatusUpdate in case any slave client does not acknowledge or provide a nack via ITcuActivityManager::sendActivityStateAck for state transition triggered by the master via ITcuActivityManager::setActivityState
string	machineName	This field is unnecessary for clients of type ClientType::MASTER. For clients of type ClientType::SLAVE this field specifies whether the slave is interested in the power state transition of all machines or only the machine where the slave is running. For interest in all machines, this field should be assigned ALL_MACHINES and for local machine assign LOCAL_MACHINE. For slaves, if this field is not provided, then the local machine name will be used as the default.

4.34.1.3 class telux::power::ITcuActivityListener

Listener class for getting notifications related to TCU-activity state and also the updates related to TCU-activity service status. The client needs to implement these methods as briefly as possible and avoid blocking calls in it. The methods in this class can be invoked from multiple different threads. Client needs to make sure that the implementation is thread-safe.

Public member functions

- virtual void onTcuActivityStateUpdate (TcuActivityState state, std::string machineName)
- virtual void onSlaveAckStatusUpdate (const telux::common::Status status, const std::string machineName, const std::vector< ClientInfo > unresponsiveClients, const std::vector< ClientInfo > nackResponseClients)
- virtual void onMachineUpdate (const std::string machineName, const MachineEvent machineEvent)
- virtual void onSlaveAckStatusUpdate (telux::common::Status status)
- virtual void onTcuActivityStateUpdate (TcuActivityState state)
- virtual ~ITcuActivityListener ()

4.34.1.3.1 Constructors and Destructors

4.34.1.3.1.1 virtual telux::power::ITcuActivityListener::~ITcuActivityListener ( ) [virtual]

Destructor of ITcuActivityListener

4.34.1.3.2 Member Function Documentation

4.34.1.3.2.1 virtual void telux::power::ITcuActivityListener::onTcuActivityStateUpdate (TcuActivityState *state*, std::string *machineName*) [virtual]

This function is called when the TCU activity state of the machine(that the client is registered for) is going to change. When the master triggers state change of a machine using ITcuActivityManager::setActivityState, the slave clients interested in that machine will receive this notification. This notification will not be received by the Master. State change of ALL_MACHINES via ITcuActivityManager::setActivityState could lead to an individual machine's state change, resulting in a notification to clients of all machines. Slave clients who got this indication must acknowledge it with ITcuActivityManager::sendActivityStateAck.

Parameters

in	*state*	TCU-activity state that the machine is about to enter
in	*machineName*	Machine name that is undergoing the state change. Assigned ALL_MACHINES for a global state change and LOCAL_MACHINE for a local state change.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backward compatibility.

4.34.1.3.2.2 virtual void telux::power::ITcuActivityListener::onSlaveAckStatusUpdate (const telux::common::Status *status*, const std::string *machineName*, const std::vector< ClientInfo > *unresponsiveClients*, const std::vector< ClientInfo > *nackResponseClients*) [virtual]

Informs the master with the consolidated acknowledgement from all slave clients for the state change previously triggered by the master client.

This API will be invoked only for the MASTER client.

On platforms with access control enabled, the client needs to have TELUX_POWER_CONTROL_STATE permission for this listener API to be invoked.

Parameters

in	*status*	This is the status of acknowledgements corresponding to a particular request. If any slave doesn't acknowledge within the configured timeout, then Status::EXPIRED is reported. If any slave sends a negative acknowledgement, then Status::NOTREADY is reported. If both types of acknowledgement errors exist, then the status code corresponding to most number of clients is reported.
in	*machineName*	Machine name that is undergoing the state change. Assigned ALL_MACHINES for a global state change and LOCAL_MACHINE for a local state change.
in	*unresponsiveClients*	List of client and respective machine names that have not responded via sendActivityStateAck for state transitions of suspend or shutdown triggered by the master via ITcuActivityManager::setActivityState.
in	*nackResponseClients*	List of client and respective machine name who responded with TcuActivityStateChangeResponse::NACK for state transitions of suspend or shutdown triggered by the master via ITcuActivityManager::setActivityState.

Note

This API is recommended for systems with and without hypervisor.

4.34.1.3.2.3 virtual void telux::power::ITcuActivityListener::onMachineUpdate (const std::string *machineName,* const MachineEvent *machineEvent*) [virtual]

This API will be invoked if any machine availability changes with respect to power management.

User can use ITcuActivityManager::getAllMachineNames() to get all updated available machines. It will be useful for the master client if they are interested in setting the TCUActivityState of a specific machine ITcuActivityManager::setActivityState().

This API is meant for clients that have instantiated the ITcuActivityManager instance using ClientType::MASTER

Parameters

in	*machineName*	Name of the machine
in	*machineEvent*	Machine event (MachineEvent)

4.34.1.3.2.4 virtual void telux::power::ITcuActivityListener::onSlaveAckStatusUpdate (telux↩ ::common::Status *status*) [virtual]

This function is called with the overall acknowledgement status from all the SLAVE clients, for state change triggered previously by MASTER client.

This API will be invoked only for the MASTER client. If at least one SLAVE client does not acknowledge within the configured timeout, then Status::EXPIRED would be reported.

On platforms with Access control enabled, the client needs to have TELUX_POWER_CONTROL_STATE permission for this listener API to be invoked.

Parameters

in	*status*	Status of the SLAVE clients' acknowledgements

Note

This API should not be used on virtual machines or on systems with hypervisor. The alternative API onSlaveAckStatusUpdate(telux::common::Status status, std::string machineName, std::vector<std::string> unresponsiveClients, std::vector<std::string> nackResponseClients) should be used.

Deprecated

Use onSlaveAckStatusUpdate(const telux::common::Status status, const std::string machineName, const std::vector<std::pair<std::string, std::string>> unresponsiveClients, const std::vector<std::pair<std::string, std::string>> nackResponseClients) API instead

4.34.1.3.2.5 virtual void telux::power::ITcuActivityListener::onTcuActivityStateUpdate (TcuActivityState *state*) [virtual]

This function is called when the TCU-activity state is going to change.

Parameters

in	*state*	TCU-activity state that system is about to enter

Deprecated

Use onTcuActivityStateUpdate(TcuActivityState state, bool isGlobalStateChange) API instead

4.34.1.4 class telux::power::ITcuActivityManager

ITcuActivityManager provides interface to register and de-register listeners to get TCU-activity state updates. And also API to initiate TCU-activity state transition.

An application can get the appropriate TCU-activity manager (i.e. ClientType::SLAVE or ClientType::MASTER) object from the power factory. The TCU-activity manager configured as the ClientType::MASTER is responsible for triggering state transitions. TCU-activity manager configured as a ClientType::SLAVE is responsible for listening to state change indications and acknowledging when it performs necessary tasks and prepares for the state transition. A machine in this power management framework represents an application processor subsystem or a host/guest virtual machine on hypervisor based platforms.

- Only one ClientType::MASTER is allowed in the system, and currently we only support allowing the ClientType::MASTER on the primary/host machine and not on the guest virtual machine.
- It is expected that all processes interested in a TCU-activity state change should register as

ClientType::SLAVE.

- When the ClientType::MASTER changes the TCU-activate state, ClientType::SLAVEs connected to the impacted machine are notified.
- ClientType::MASTER can trigger the TCU-activity state change of a specific machine or all machines at once.
- If the ClientType::SLAVE wants to differentiate between a state change indication that is the result of a trigger for all machines or a trigger for its specific machines, it can be detected using the machine name provided in the listener API.
- When the ClientType::MASTER triggers an all machines TCU-activity state change, only the machines that are not in the desired state will undergo the state transition, and the ClientType::SLAVEs to those machines will be notified.
- In the case of
  - TcuActivityState::SUSPEND or TcuActivityState::SHUTDOWN trigger:
    - After becoming ready for state change, all ClientType::SLAVE should acknowledge back.
    - The ClientType::MASTER will get notification about the consolidated acknowledgement status of all ClientType::SLAVEs.
    - On getting a successful consolidated acknowledgement from all the ClientType::SLAVE for the suspend trigger, the power framework allows the respective machine to suspend. On getting a successful consolidated acknowledgement from all the ClientType::SLAVEs for the shutdown trigger, the power framework triggers the respective machine shutdown without waiting further.
    - If the ClientType::SLAVE sends a NACK to indicate that it is not ready for state transition or fails to acknowledge before the configured time, then the ClientType::MASTER will get to know via a consolidated/slave acknowledgement status notification.
    - In such failed cases, if the ClientType::MASTER wants to stop the state transition considering the information in the consolidated acknowledgement, then the ClientType::MASTER is allowed to trigger a new TCU-activity state change, or else the state transition will proceed after the configured timeout.
  - TcuActivityState::RESUME trigger:
    - Power framework will prevent the respective machine from going into suspend.
    - No acknowledgement will be required from ClientType::SLAVE and the ClientType::MASTER will not be getting consolidated/slave acknowledgement as machine will be already resumed.

When the application is notified about the service being unavailable, the TCU-activity state notifications will be inactive. After the service becomes available, the existing listener registrations will be maintained.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual telux::common::Status registerListener (std::weak_ptr< ITcuActivityListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< ITcuActivityListener > listener)=0
- virtual telux::common::Status registerServiceStateListener (std::weak_ptr< telux::common::IServiceStatusListener > listener)=0
- virtual telux::common::Status deregisterServiceStateListener (std::weak_ptr<

telux::common::IServiceStatusListener > listener)=0

- virtual telux::common::Status getMachineName (std::string &machineName)=0
- virtual telux::common::Status getAllMachineNames (std::vector< std::string > &machineNames)=0
- virtual telux::common::Status setActivityState (TcuActivityState state, std::string machineName, telux::common::ResponseCallback callback=nullptr)=0
- virtual TcuActivityState getActivityState ()=0
- virtual telux::common::Status sendActivityStateAck (StateChangeResponse ack, TcuActivityState state)=0
- virtual telux::common::Status setModemActivityState (TcuActivityState state)=0
- virtual bool isReady ()=0
- virtual std::future< bool > onReady ()=0
- virtual telux::common::Status setActivityState (TcuActivityState state, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status sendActivityStateAck (TcuActivityStateAck ack)=0
- virtual ~ITcuActivityManager ()

4.34.1.4.1 Constructors and Destructors

4.34.1.4.1.1 virtual telux::power::ITcuActivityManager::~ITcuActivityManager ( ) [virtual]

Destructor of ITcuActivityManager

4.34.1.4.2 Member Function Documentation

4.34.1.4.2.1 virtual telux::common::ServiceStatus telux::power::ITcuActivityManager::getServiceStatus ( ) [pure virtual]

This status indicates whether the ITcuActivityManager object is in a usable state.

Returns

telux::common::ServiceStatus

4.34.1.4.2.2 virtual telux::common::Status telux::power::ITcuActivityManager::registerListener (std::weak_ptr< ITcuActivityListener > *listener*) [pure virtual]

Register a listener for updates on TCU-activity state changes.

Parameters

in	*listener*	Pointer of ITcuActivityListener object that processes the notification

Returns

Status of registerListener i.e success or suitable status code.

4.34.1.4.2.3 virtual telux::common::Status telux::power::ITcuActivityManager::deregisterListener (std::weak_ptr< ITcuActivityListener > *listener*) [pure virtual]

Remove a previously registered listener.

Parameters

in	*listener*	Previously registered ITcuActivityListener that needs to be removed

Returns

Status of deregisterListener, success or suitable status code

4.34.1.4.2.4 virtual telux::common::Status telux::power::ITcuActivityManager::registerServiceStateListener (std::weak_ptr< telux::common::IServiceStatusListener > *listener*) [pure virtual]

Register a listener for updates on TCU-activity management service status.

Parameters

in	*listener*	Pointer of IServiceStatusListener object that processes the notification

Returns

Status of registerServiceStateListener i.e success or suitable status code.

4.34.1.4.2.5 virtual telux::common::Status telux::power::ITcuActivityManager::deregisterServiceStateListener (std::weak_ptr< telux::common::IServiceStatusListener > *listener*) [pure virtual]

Remove a previously registered listener for service status updates.

Parameters

in	*listener*	Previously registered IServiceStatusListener that needs to be removed

Returns

Status of deregisterServiceStateListener, success or suitable status code

4.34.1.4.2.6 virtual telux::common::Status telux::power::ITcuActivityManager::getMachineName (std::string & *machineName*) [pure virtual]

This API allows the caller to get the machine name where the client is running. It is intended to identify the local machine name on a platform where multiple machines are available in the power framework.

Parameters

out	*machineName*	Machine name where the process is running

Returns

Status of getMachineName, success or suitable status code.

4.34.1.4.2.7 virtual telux::common::Status telux::power::ITcuActivityManager::getAllMachineNames (std::vector< std::string > & *machineNames*) [pure virtual]

Enumerates all machines in the system that are available and ready to be managed by the power framework.

This API is meant for clients that have instantiated the ITcuActivityManager instance using ClientType::MASTER. If the platform has multiple machines available, knowing their names will be useful if the master is interested in individually modifying the activity state of any available machine using setActivityState.

Parameters

out	*machineNames*	List of machine names that are available for power management.

Returns

Status of getAllMachineNames, success or suitable status code.

4.34.1.4.2.8 virtual telux::common::Status telux::power::ITcuActivityManager::setActivityState (TcuActivityState *state*, std::string *machineName*, telux::common::ResponseCallback *callback* = `nullptr`) [pure virtual]

Initiates a TCU-activity state transition.

This API also initiates the relevant internal operation if the platform is configured to change the modem activity state automatically when the TCU activity state changes.

This API needs to be used cautiously, as it could change the power-state of the system and may affect other processes. For example, if a master sets the SUSPEND state, all SLAVE processes will suspend their activity, allowing the system to suspend.

This API can only be invoked by clients that have instantiated the ITcuActivityManager instance using ClientType::MASTER.

Based on the final acknowledgements from all the slaves ITcuActivityListener::onSlaveAckStatusUpdate,

1. If the acknowledgement status is SUCCESS, then the framework attempts to state transition(SUSPEND/SHUTDOWN) immediately on the relevant machines.

2. If the acknowledgement status is not SUCCESS, then the framework waits for a configured timeout before attempting the state transition(SUSPEND/SHUTDOWN) on the relevant machines.

On platforms with Access control enabled, Caller needs to have TELUX_POWER_CONTROL_STATE permission to invoke this API successfully.

Parameters

in	*state*	TCU-activity state that the system is intended to enter
in	*machineName*	Machine name if the state transition is intended for the specific machine only. If assigned to ALL_MACHINES, then the state applies to the whole system.
in	*callback*	Optional callback to get the response for the TCU-activity state transition command

Returns

Status of setActivityState i.e. success or suitable status code.

4.34.1.4.2.9 virtual TcuActivityState telux::power::ITcuActivityManager::getActivityState ( ) [pure virtual]

Get the current TCU-activity state.

Returns

TcuActivityState

4.34.1.4.2.10 virtual telux::common::Status telux::power::ITcuActivityManager::sendActivityStateAck (StateChangeResponse *ack*, TcuActivityState *state*) [pure virtual]

Sends the acknowledgement after processing a TCU-activity state notification. This indicates that the client is prepared for the state transition. Only one acknowledgement should be sent per ClientType::SLAVE instance of ITcuActivityManager, even if multiple listeners are registered with that instance.

All slave clients that received a state change notification via TcuActivityListener::onTcuActivityStateUpdate must acknowledge using this API.

Parameters

in	*ack*	Acknowledgement for a TCU-activity state notification StateChangeResponse.
in	*state*	Represents the TCU activity state transition event corresponding to this acknowledgement.

Returns

Status of sendActivityStateAck i.e. success or suitable status code.

4.34.1.4.2.11 virtual telux::common::Status telux::power::ITcuActivityManager::setModemActivityState (TcuActivityState *state*) [pure virtual]

Explicitly sets the modem state change.

The platform could be configured to automatically manage the modem state when setTcuActivityState is called. For example, when suspend is called, the implementation will also set the modem to suspend. In that case, this API need not be invoked after setting the TCU state.

This API needs to be used cautiously, as it could affect WWAN functionalities.

This API is meant for clients that have instantiated the ITcuActivityManager instance using ClientType::MASTER

On platforms with Access control enabled, Caller needs to have TELUX_POWER_CONTROL_STATE permission to invoke this API successfully.

Parameters

in	*state*	Activity state that the modem is intended to enter TcuActivityState SUSPEND - Reduce/Throttle the modem activities RESUME - Restore the activities that were throttled earlier Any other input is considered invalid.

Returns

Status of setModemActivityState i.e. success or suitable status code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.34.1.4.2.12 virtual bool telux::power::ITcuActivityManager::isReady ( ) [pure virtual]

Checks the status of TCU-activity services, if other APIs are ready for use, and returns the results.

Returns

True if the services are ready; otherwise, false.

Deprecated

Use ITcuActivityManager::getServiceStatus() API.
telux::power::ITcuActivityManager::getServiceStatus

4.34.1.4.2.13 virtual std::future<bool> telux::power::ITcuActivityManager::onReady ( ) [pure virtual]

Waits for TCU-activity services to be ready.

Returns

A future that caller can wait on to be notified when TCU-activity services are ready.

Deprecated

Use InitResponseCb in PowerFactory::getTcuActivityManager instead, to get get notified about subsystem readiness telux::power::PowerFactory::getTcuActivityManager

4.34.1.4.2.14 virtual telux::common::Status telux::power::ITcuActivityManager::setActivityState (TcuActivityState *state*, telux::common::ResponseCallback *callback* = `nullptr`) [pure virtual]

Initiates a TCU-activity state transition. If platform is configured to change modem activity state automatically when TCU activity state is changed, this API initiates the relevant internal operation.

This API needs to be used cautiously, as it could change the power-state of the system and may affect other processes.

This API should only be invoked by a client that has instantiated the ITcuActivityManager instance using ClientType::MASTER

On platforms with access control enabled, the caller needs to have TELUX_POWER_CONTROL_STATE permission to invoke this API successfully.

Parameters

in	*state*	TCU-activity state that the system is intended to enter
in	*callback*	Optional callback to get the response for the TCU-activity state transition command

Returns

Status of setActivityState i.e. success or suitable status code.

Note

This API should not be used on virtual machines or on systems with hypervisor. The alternative API setActivityState(TcuActivityState state, std::string machineName = "",telux::common::ResponseCallback callback = nullptr) should be used.

Deprecated

Use setActivityState(TcuActivityState state, std::string machineName, telux::common::ResponseCallback) API instead

4.34.1.4.2.15 virtual telux::common::Status telux::power::ITcuActivityManager::sendActivityStateAck (TcuActivityStateAck *ack*) [pure virtual]

Sends acknowledgement after processing a TCU activity state notification. This indicates that the client is prepared for state transition. Only one acknowledgement is expected from a single client process, although it may have multiple listeners.

All slave clients that received a state change notification via TcuActivityListener::onTcuActivityStateUpdate must acknowledge using this API.

Parameters

in	*ack*	Acknowledgement for a TCU-activity state notification.

Returns

Status of sendActivityStateAck i.e. success or suitable status code.

Deprecated

Use sendActivityStateAck(TcuActivityState state, StateChangeResponse ack) API instead

4.34.2 Enumeration Type Documentation

4.34.2.1 enum telux::power::TcuActivityState [strong]

Defines the supported TCU-activity states that the listeners will be notified about.

Enumerator

UNKNOWN To indicate that system state information is not available
SUSPEND System is going to SUSPEND state
RESUME System is going to RESUME state
SHUTDOWN System is going to SHUTDOWN

4.34.2.2 enum telux::power::StateChangeResponse [strong]

Defines the acknowledgements to TCU-activity state transition. The client process sends this response via ITcuActivityManager::sendActivityStateAck after processing the TCU-activity state change notification received via ITcuActivityListener::onTcuActivityStateUpdate.

The framework does not require slave clients to respond when changing the state to TcuActivityState::RESUME.

Enumerator

ACK Processed TCU-activity state change
NACK Not prepared/ready for TCU-activity state change

4.34.2.3 enum telux::power::ClientType [strong]

Defines the type of client that would be using the ITcuActivityManager APIs. Client that just needs the TcuActivityState notifications needs to choose ClientType::SLAVE. And the client that determines the TcuActivityState would choose ClientType::MASTER. Only a Master client can set the TcuActivityState. In a system, there should be a single Master client.

The ClientType needs to be chosen while instantiating the ITcuActivityManager, using the API PowerFactory::getTcuActivityManager

Enumerator

SLAVE Client is a slave and interested in state change notification
MASTER Client makes the decision on when the TcuActivityState should change

4.34.2.4 enum telux::power::MachineEvent [strong]

Defines the type of event with respect to machine availability. This only represents the availability of the machine to manage its activity state and not whether the machine itself is enabled.

ITcuActivityListener::onMachineUpdate() can be used to listen to changes in machine availability.

Enumerator

AVAILABLE New machine available for power management
UNAVAILABLE machine unavailable for power management

4.34.2.5 enum telux::power::TcuActivityStateAck [strong]

Defines the acknowledgements to TCU-activity states. The client process sends this after processing the TcuActivityState notification, indicating that it is prepared for state transition

Acknowledgement for TcuActivityState::RESUME is not required, as the state transition has already happened.

Deprecated

The API ITcuActivityManager::sendActivityStateAck (TCUActivityStateAck) that uses this enum is deprecated. Instead, use ITcuActivityManager::sendActivityStateAck (StateChangeResponse, TcuActivityState).

Enumerator

SUSPEND_ACK processed TcuActivityState::SUSPEND notification
SHUTDOWN_ACK processed TcuActivityState::SHUTDOWN notification

4.34.3 Variable Documentation

4.34.3.1 const std::string telux::power::ALL_MACHINES = "ALL_MACHINES" [static]

This special name represents all the machines on the platform. A client could specify this name when using ClientInstanceConfig::machineName to mean all machines as opposed to a specific machine name.

4.34.3.2 const std::string telux::power::LOCAL_MACHINE = "LOCAL_MACHINE" [static]

This special name represents the machine name where the process is running. A client could specify this name when using ClientInstanceConfig::machineName to mean local machine's name.
"LOCAL_MACHINE" in case of a hypervisor environment specifies that the client is interested in the virtual machine the client is running on.

4.35 Modem Configuration

- Modem Config

This section contains APIs related to Modem config.

4.36 Modem Config

This section contains APIs related to Modem Config operations.

4.36.1 Data Structure Documentation

4.36.1.1 class telux::config::ConfigFactory

ConfigFactory allows creation of config related classes.

Public member functions

- virtual std::shared_ptr< IModemConfigManager > getModemConfigManager (telux::common::InitResponseCb callback=nullptr)=0
- virtual std::shared_ptr< IConfigManager > getConfigManager (telux::common::InitResponseCb callback=nullptr)=0

Static Public Member Functions

- static ConfigFactory & getInstance ()

4.36.1.1.1 Member Function Documentation

4.36.1.1.1.1 static ConfigFactory& telux::config::ConfigFactory::getInstance ( ) [static]

Get instance of Config Factory

4.36.1.1.1.2 virtual std::shared_ptr<IModemConfigManager> telux::config::ConfigFactory::getModemConfigManager (telux::common::InitResponseCb *callback* = `nullptr`) [pure virtual]

Get instance of ModemConfig manager

On platforms with Access control enabled, Caller needs to have TELUX_CONFIG_MODEM_CONFIG permission to invoke this API successfully.

Parameters

in	*callback*	Optional callback to get the response of Modem Config Manager initialization.

Returns

pointer of IModemConfigManager object.

4.36.1.1.3 virtual std::shared_ptr<IConfigManager> telux::config::ConfigFactory::getConfigManager (telux::common::InitResponseCb *callback* = `nullptr`) [pure virtual]

Get instance of the Config manager

On platforms with Access control enabled, Caller needs to have TELUX_CONFIG_APPS_CONFIG permission to invoke this API successfully.

Parameters

in	*callback*	Optional callback to get the response of Config Manager initialization.

Returns

pointer of IConfigManager object.

4.36.1.2 class telux::config::IConfigListener

IConfigListener interface is used to receive notifications related to any updates in the configurations dynamically.

Public member functions

- virtual void onConfigUpdate (std::string key, std::string value)
- virtual ~IConfigListener ()

4.36.1.2.1 Constructors and Destructors

4.36.1.2.1.1 virtual telux::config::IConfigListener::~IConfigListener ( ) [virtual]

4.36.1.2.2 Member Function Documentation

4.36.1.2.2.1 virtual void telux::config::IConfigListener::onConfigUpdate (std::string *key*, std::string *value*) [virtual]

This API is invoked when there is any update in the configurations dynamically.

Parameters

in	*key*	- The key updated in the configurations.
in	*value*	- The corresponding value for the key that was updated in the configurations.

4.36.1.3 class telux::config::IConfigManager

IConfigManager provides APIs to retrieve an instance of the manager, APIs for processes to update and retrieve configurations dynamically.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual telux::common::Status registerListener (std::weak_ptr< IConfigListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< IConfigListener > listener)=0
- virtual telux::common::Status setConfig (const std::string key, const std::string value)=0
- virtual const std::string getConfig (const std::string key)=0
- virtual const std::map< std::string, std::string > getAllConfigs ()=0
- virtual ~IConfigManager ()

4.36.1.3.1 Constructors and Destructors

4.36.1.3.1.1 virtual telux::config::IConfigManager::~IConfigManager ( ) [virtual]

Destructor of IConfigManager

4.36.1.3.2 Member Function Documentation

4.36.1.3.2.1 virtual telux::common::ServiceStatus telux::config::IConfigManager::getServiceStatus ( ) [pure virtual]

This status indicates whether the manager object is in a usable state or not.

Returns

SERVICE_AVAILABLE - if apps config manager is ready to use. SERVICE_UNAVAILABLE - if apps config manager is temporarily unavailable to use. SERVICE_FAILED - if apps config manager encountered an irrecoverable failure and can not be used.

4.36.1.3.2.2 virtual telux::common::Status telux::config::IConfigManager::registerListener (std::weak_ptr< IConfigListener > *listener*) [pure virtual]

This API is used to register a listener for getting the updates when the configurations are updated dynamically.

Parameters

in	*listener*	- Pointer of object that processes the notification.

Returns

Status of registerForUpdates i.e success or suitable status code.

4.36.1.3.2.3 virtual telux::common::Status telux::config::IConfigManager::deregisterListener (std::weak_ptr< IConfigListener > *listener*) [pure virtual]

This API is used to deregister a listener from getting the updates when the configurations are updated dynamically.

Parameters

in	*listener*	- Pointer of object that processes the notification.

Returns

Status of registerForUpdates i.e success or suitable status code.

4.36.1.3.2.4 virtual telux::common::Status telux::config::IConfigManager::setConfig (const std::string *key*, const std::string *value*) [pure virtual]

This API is used to update the key and the corresponding value in the configurations dynamically.

On platforms with Access control enabled, if -

1. /etc/tel.conf needs to be updated - caller needs to have TELUX_SET_GLOBAL_CONFIG permission to invoke this API successfully.
2. App specific conf needs to be updated - caller needs to have TELUX_SET_LOCAL_CONFIG permission to invoke this API successfully.

In order to update any configuration onto the file, all the permissions needed - DAC permissions and sepolicy requirements, should be taken care by the application.

The API does not perform any strict checking for the value being set. Please refer to tel.conf for valid values for configuration items.

Parameters

in	*key*	- The key that needs to be updated in the configurations.
in	*value*	- The corresponding value for the key that needs the update in the configurations.

Returns

Status of setConfig i.e success or suitable status code.

4.36.1.3.2.5 virtual const std::string telux::config::IConfigManager::getConfig (const std::string *key*) [pure virtual]

This API is used to retrieve the value for the corresponding key from the configurations dynamically.

Parameters

in	*key*	- The key whose value is to be retrieved from the configurations.

Returns

The value for the key passed.

4.36.1.3.2.6 virtual const std::map<std::string, std::string> telux::config::IConfigManager::getAll↩ Configs ( ) [pure virtual]

This API is used to retrieve all the configurations for the application at present.

Returns

A map of key-value pairs depicting all the application's configurations at present.

4.36.1.4 struct telux::config::ConfigInfo

Data fields

Type	Field	Description
ConfigId	id	id - stores the id of the configuration type - stores config type size - stores the size of the configuration desc - stores the configuration description version - stores version of the config file
ConfigType	type	
uint32_t	size	
string	desc	
uint32_t	version	

4.36.1.5 class telux::config::IModemConfigListener

Listener class for getting notifications related to configuration change detection. The client needs to implement these methods as briefly as possible and avoid blocking calls in it. The methods in this class can be invoked from multiple different threads. Client needs to make sure that the implementation is thread-safe.

Public member functions

- virtual void onConfigUpdateStatus (ConfigUpdateStatus status, int slotId)
- virtual ~IModemConfigListener ()

4.36.1.5.1 Constructors and Destructors

4.36.1.5.1.1 virtual telux::config::IModemConfigListener::~IModemConfigListener ( ) [virtual]

Destructor of IModemConfigListener

4.36.1.5.2 Member Function Documentation

4.36.1.5.2.1 virtual void telux::config::IModemConfigListener::onConfigUpdateStatus (ConfigUpdate↩ Status *status,* int *slotId*) [virtual]

This function is called when a configuration update is detected. It is applicable only to SOFTWARE config.

Parameters

in	*status*	update status of config.
in	*slotId*	slotId where update is detected.

4.36.1.6 class telux::config::IModemConfigManager

IModemConfigManager provides interface to list config files present in modem's storage. load a new config file in modem, activate a config file, get active config file information, deactivate a config file, delete config file from the modem's storage, get and set mode of config auto selection, register and deregister listener for config update in modem. The config files are also referred to as MBNs.

Public member functions

- virtual bool isSubsystemReady ()=0
- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual std::future< bool > onSubsystemReady ()=0
- virtual telux::common::Status requestConfigList (ConfigListCallback cb)=0
- virtual telux::common::Status loadConfigFile (std::string filePath, ConfigType configType, telux::common::ResponseCallback cb=nullptr)=0
- virtual telux::common::Status activateConfig (ConfigType configType, ConfigId configId, int slotId=DEFAULT_SLOT_ID, telux::common::ResponseCallback cb=nullptr)=0
- virtual telux::common::Status getActiveConfig (ConfigType configType, GetActiveConfigCallback cb, int slotId=DEFAULT_SLOT_ID)=0
- virtual telux::common::Status deactivateConfig (ConfigType configType, int slotId=DEFAULT_SLOT_ID, telux::common::ResponseCallback cb=nullptr)=0
- virtual telux::common::Status deleteConfig (ConfigType configType, ConfigId configId="", telux::common::ResponseCallback cb=nullptr)=0
- virtual telux::common::Status getAutoSelectionMode (GetAutoSelectionModeCallback cb, int slotId=DEFAULT_SLOT_ID)=0
- virtual telux::common::Status setAutoSelectionMode (AutoSelectionMode mode, int slotId=DEFAULT_SLOT_ID, telux::common::ResponseCallback cb=nullptr)=0
- virtual telux::common::Status registerListener (std::weak_ptr< IModemConfigListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< IModemConfigListener > listener)=0
- virtual ~IModemConfigManager ()

4.36.1.6.1 Constructors and Destructors

4.36.1.6.1.1 virtual telux::config::IModemConfigManager::~IModemConfigManager ( ) [virtual]

Destructor of IModemConfigManager

4.36.1.6.2 Member Function Documentation

4.36.1.6.2.1 virtual bool telux::config::IModemConfigManager::isSubsystemReady ( ) [pure virtual]

Checks the status of modem config subsystem and returns the result.

Returns

If true that means ModemConfigManager is ready for performing config operations.

Deprecated

Use getServiceStatus API

4.36.1.6.2.2 virtual telux::common::ServiceStatus telux::config::IModemConfigManager::getServiceStatus ( ) [pure virtual]

This status indicates whether the manager object is in a usable state or not.

Returns

SERVICE_AVAILABLE - if modem config manager is ready to use. SERVICE_UNAVAILABLE - if modem config manager is temporarily unavailable to use. SERVICE_FAILED - if modem config manager encountered an irrecoverable failure and can not be used.

4.36.1.6.2.3 virtual std::future<bool> telux::config::IModemConfigManager::onSubsystemReady ( ) [pure virtual]

Wait for modem config subsystem to be ready.

Returns

A future that caller can wait on to be notified when modem config subsystem is ready.

Deprecated

Use InitResponseCb callback in factory API ConfigFactory::getModemConfigManager.

4.36.1.6.2.4 virtual telux::common::Status telux::config::IModemConfigManager::requestConfigList (ConfigListCallback *cb*) [pure virtual]

Fetching the list of config files present in modem's storage.

Parameters

in	*cb*	- callback to the Response function.

returns SUCCESS if the request to get config list is sent successfully.

4.36.1.6.2.5 virtual telux::common::Status telux::config::IModemConfigManager::loadConfigFile (std::string *filePath,* ConfigType *configType,* telux::common::ResponseCallback *cb = nullptr*) [pure virtual]

Loads a new config file into the modem's storage. This is a persistent operation. Only the config files loaded into the modem's storage can be activated.

Parameters

in	*filePath*	- it defines the path to the config file.
in	*configType*	- type of the config file.
in	*cb*	- callback to the response function.

returns SUCCESS if the request to load config file is sent successfully.

4.36.1.6.2.6 virtual telux::common::Status telux::config::IModemConfigManager::activateConfig (ConfigType *configType,* ConfigId *configId,* int *slotId = DEFAULT_SLOT_ID,* telux::common::ResponseCallback *cb = nullptr*) [pure virtual]

Activates the config file on specified slot id. A file for activation must be loaded or should already be present in modem's storage.

Parameters

in	*configType*	- type of the config file.
in	*configId*	- id of the config file.
in	*slotId*	- it defines the slot id to be selected.
in	*cb*	- callback to the response function.

Returns

SUCCESS if the request to activate config file is sent successfully.

4.36.1.6.2.7 virtual telux::common::Status telux::config::IModemConfigManager::getActiveConfig (ConfigType *configType,* GetActiveConfigCallback *cb,* int *slotId = DEFAULT_SLOT_ID*) [pure virtual]

Get the currently active config file information for the specified slot id. In case default config files are activated, would return error.

Parameters

in	*configType*	- type of the config file.
in	*cb*	- callback to the response function.
in	*slotId*	- it defines the slot id to be selected.

Returns

SUCCESS if the request to get active config information is sent successfully.

4.36.1.6.2.8 virtual telux::common::Status telux::config::IModemConfigManager::deactivateConfig (ConfigType *configType,* int *slotId = DEFAULT_SLOT_ID,* telux::common::Response↩ Callback *cb = nullptr*) [pure virtual]

Deactivates the config file for the specified slot id.

Parameters

in	*configType*	- type of the config file.
in	*slotId*	- slot id to be selected for deactivation of config.
in	*cb*	- callback to the response function.

Returns

SUCCESS if the request to deactivate config file is sent successfully

4.36.1.6.2.9 virtual telux::common::Status telux::config::IModemConfigManager::deleteConfig (ConfigType *configType,* ConfigId *configId = "",* telux::common::ResponseCallback *cb = nullptr*) [pure virtual]

Deletes the config file from the modem's storage.

Parameters

in	*configType*	- type of the config file.
in	*configId*	- id of the config file. This parameter is optional if not provided all the config files of the given config type are deleted from modem's storage.
in	*cb*	- callback to the Response function.

Returns

SUCCESS if the request to delete config file is sent successfully

4.36.1.6.2.10 virtual telux::common::Status telux::config::IModemConfigManager::getAutoSelection↩ Mode (GetAutoSelectionModeCallback *cb,* int *slotId = DEFAULT_SLOT_ID*) [pure virtual]

Fetching the mode of config auto selection for specified slot id.

Parameters

in	*cb*	- callback to the response function.
in	*slotId*	- slot id of config.

Returns

SUCCESS if the request to get selection mode is sent successfully

4.36.1.6.2.11 virtual telux::common::Status telux::config::IModemConfigManager::setAutoSelectionMode (AutoSelectionMode *mode*, int *slotId = DEFAULT_SLOT_ID*, telux::common::ResponseCallback *cb = nullptr*) [pure virtual]

Setting the mode of config auto selection for specified slot id.

Parameters

in	*mode*	- auto selection mode status.
in	*slotId*	- slot id of the config.
in	*cb*	- callback to the response function.

Returns

SUCCESS if the request to set selection mode is sent successfully.

4.36.1.6.2.12 virtual telux::common::Status telux::config::IModemConfigManager::registerListener (std::weak_ptr< IModemConfigListener > *listener*) [pure virtual]

Registeres the listener for indications.

Parameters

in	*listener*	- pointer to implemented listener.

Returns

SUCCESS if the request to register listener is sent successfully.

4.36.1.6.2.13 virtual telux::common::Status telux::config::IModemConfigManager::deregisterListener (std::weak_ptr< IModemConfigListener > *listener*) [pure virtual]

Deregisteres the listener from indications.

Parameters

in	*listener*	- pointer to registered listener.

Returns

SUCCESS if the request to deregister listener is sent successfully.

4.36.2 Enumeration Type Documentation

4.36.2.1 enum telux::config::ConfigType [strong]

Enumerator

HARDWARE For hardware or platform related configuration files
SOFTWARE For software or carrier related configuration files

4.36.2.2 enum telux::config::AutoSelectionMode [strong]

Selection Mode defines status of auto selection mode for configs.

Enumerator

DISABLED Auto selection disabled
ENABLED Auto selection enabled

4.36.2.3 enum telux::config::ConfigUpdateStatus [strong]

ConfigUpdateStatus represent status of config update, a update of config happens when a software config is activated and all segments using the config are updated with new config.

Enumerator

START start of updation process
COMPLETE end of updation process

4.37 Sensor

- Sensor Service
- Sensor Control
- Sensor Feature Control

This section contains APIs related to sensor configuration, control, data acquisition and sensor feature control.

4.38 Sensor Service

This section contains APIs, data structures and components to access the sensor sub-system.

4.38.1 Data Structure Documentation

4.38.1.1 struct telux::sensor::SensorInfo

Information related to sensor.

Data fields

Type	Field	Description
int	id	Unique identifier for the sensor.
SensorType	type	The type of sensor, telux::sensor::SensorType
string	name	The name of the sensor This name is used to get a reference to a sensor with telux::sensor::ISensorManager::getSensorClient
string	vendor	The name of the vendor
vector< float >	samplingRates	List of supported sampling rates by the sensor hardware, number of samples per second (Hz)
float	maxSampling↩ Rate	The maximum sampling rate the sensor can be configured for. This can be set in /etc/sensors.conf for each sensor and should be less than the maximum sampling rate supported by the sensor hardware, number of samples per second (Hz) This attribute should be considered while using the API telux::sensor::ISensorClient::configure
uint32_t	maxBatch↩ Count↩ Supported	Maximum batch count supported by the sensor, i.e. the maximum number of sensor events that the underlying framework can buffer. This attribute should be considered while using the API telux::sensor::ISensorClient::configure
uint32_t	minBatch↩ Count↩ Supported	Minimum batch count supported by the sensor. This is set in /etc/sensors.conf for each sensor. This attribute should be considered while using the API telux::sensor::ISensorClient::configure
int	range	The range offered by the sensor. This configuration can be set in /etc/sensors.conf for each sensor. For accelerometers, this is the number of Gs (force per unit mass due to gravity) in either direction (+/-) on each axis For gyroscopes, this is the number of degrees per second (dps) in either direction (+/-) along each axis
int	version	The version of the sensor considering the hardware part and the driver
float	resolution	This is the smallest difference between two values reported by this sensor, in meter per second per second for accelerometer, radians per second for gyroscope

Type	Field	Description
float	maxRange	The maximum range this sensor offers, in meter per second per second for accelerometer, radians per second for gyroscope. This attribute depends on the SensorInfo::range of the sensor set in the configuration file. For example, a range of 1G results in a maximum range of approximately 9.8 m/s/s and a range of 2G gives a maximum range of about 19.6 m/s/s.

4.38.1.2 struct telux::sensor::SensorConfiguration

Configurable parameters of a sensor.

Data fields

Type	Field	Description
float	samplingRate	The sampling rate for the sensor, number of samples per second (Hz) In case of telux::sensor::ISensorClient::configure, the requested sampling rate should be one of the sampling rates provided in the telux::sensor::SensorInfo::samplingRates and should be less than the telux::sensor::SensorInfo::maxSamplingRate. If the requested sampling rate is less than the minimum value in the telux::sensor::SensorInfo::samplingRates, it will be set to the least of the values in telux::sensor::SensorInfo::samplingRates If the requested sampling rate is not one of the supported sampling rates in telux::sensor::SensorInfo::samplingRates, the requested value is floored to the nearest value in telux::sensor::SensorInfo::samplingRates Consider telux::sensor::SensorInfo::samplingRates having values 12, 26, 52. If requested sampling rate in configure API is 7, the sampling rate considered by the sensor framework would be 12. If requested sampling rate in configure API is 51, the sampling rate considered by the sensor framework would be 26. In case of a configuration update received via telux::sensor::ISensorEventListener::onConfigurationUpdate, the current sampling rate configuration is passed to the listener

Type	Field	Description
uint32_t	batchCount	The batch count of the sensor. Batch count is the count of number of samples the underlying framework would buffer before notifying the client of the data. The intention is to reduce the number of interactions between the hardware, framework and the user application to reduce power consumption, improve compute efficiency and reduce number of interactions between different components. It is important to consider latency while deciding the batch count for a sensor. Higher the batch count, more is the latency for the samples. In case of telux::sensor::ISensorClient::configure, the requested batch count should be lesser than the maximum supported batch count telux::sensor::SensorInfo::maxBatchCountSupported. Also, the batch count considered is impacted by the telux::sensor::SensorInfo::minBatchCountSupported. If the requested batch count is less than telux::sensor::SensorInfo::minBatchCountSupported, it will be set to telux::sensor::SensorInfo::minBatchCountSupported If the requested batch count is not a multiple of telux::sensor::SensorInfo::minBatchCountSupported, the requested value is floored to the nearest multiple of telux::sensor::SensorInfo::minBatchCountSupported Consider telux::sensor::SensorInfo::minBatchCountSupported having a value of 7. If requested batchCount in configure API is 2, the batchCount considered by the sensor framework would be 7. If requested batchCount in configure API is 23, the batchCount considered by the sensor framework would be 21. In case of a configuration update telux::sensor::ISensorEventListener::onConfigurationUpdate, this field indicates the current configuration for batch count.
SensorConfig↩ Mask	validityMask	Bitset indicating the validity of the received sensor configuration via telux::sensor::ISensorClient::getConfiguration and telux::sensor::ISensorEventListener::onConfigurationUpdate. The configuration items that were never set would have return false when tested for using std::bitset::test. Further, this bitset should be set by the user to indicate the valid fields while configuring the sensor using telux::sensor::ISensorClient::configure. For continuous stream of data from a sensor, the validity of SAMPLING_RATE and BATCH_COUNT from SensorConfigParams should be considered. If the sensor had been already configured with both sampling rate and batch count, it is possible to reconfigure the sensor partially with just one of these attributes and setting the required validity flag.
SensorConfig↩ Mask	updateMask	Bitset indicating the parameters that were updated since last notification via telux::sensor::ISensorEventListener::onConfigurationUpdate

4.38.1.3 struct telux::sensor::MotionSensorData

Structure of a single sample from a motion sensor.

Data fields

Type	Field	Description
float	x	x-axis data, meter per second per second for accelerometer, radians per second for gyroscope
float	y	y-axis data, meter per second per second for accelerometer, radians per second for gyroscope
float	z	z-axis data, meter per second per second for accelerometer, radians per second for gyroscope

4.38.1.4 struct telux::sensor::UncalibratedMotionSensorData

Structure of a single sample from uncalibrated motion sensor.

Data fields

Type	Field	Description
Motion↩ SensorData	data	Uncalibrated motion sensor data MotionSensorData
Motion↩ SensorData	bias	Bias for the uncalibrated data MotionSensorData

4.38.1.5 struct telux::sensor::SensorEvent

Structure of a single sensor event.

Data fields

Type	Field	Description
uint64_t	timestamp	Timestamp when the event was generated on the hardware, nanosecond since boot-up
union Sensor↩ Event	__unnamed_↩ _	Sensor data

4.38.1.6 struct telux::sensor::SensorFeature

Feature offered by sensor hardware and/or software framework.

Data fields

Type	Field	Description
string	name	Name of the feature

4.38.1.7 struct telux::sensor::SensorFeatureEvent

Structure of an event that is generated from a sensor feature.

Data fields

Type	Field	Description
uint64_t	timestamp	Best estimate of timestamp indicating the time of occurance of the event, nanosecond since boot-up
string	name	Name of the feature that generated the event
int	id	The ID of the generated event

4.38.1.8 class telux::sensor::SensorFactory

SensorFactory is the central factory to create instances of sensor objects.

Public member functions

- virtual std::shared_ptr< ISensorManager > getSensorManager (telux::common::InitResponseCb clientCallback=nullptr)=0
- virtual std::shared_ptr< ISensorFeatureManager > getSensorFeatureManager (telux::common::InitResponseCb clientCallback=nullptr)=0

Static Public Member Functions

- static SensorFactory & getInstance ()

4.38.1.8.1 Member Function Documentation

4.38.1.8.1.1 static SensorFactory& telux::sensor::SensorFactory::getInstance ( ) [static]

Get Sensor Factory instance.

Returns

The singleton instance of SensorFactory object

4.38.1.8.1.2 virtual std::shared_ptr<ISensorManager> telux::sensor::SensorFactory::getSensorManager (telux::common::InitResponseCb *clientCallback* = `nullptr`) [pure virtual]

Get an instance of Sensor Manager. The ownership of the returned object is with the caller of this method. The reference to the instance is not held by the SensorFactory. If the returned reference is released, any request for ISensorManager shall result in creation of a new instance

Parameters

in	*clientCallback*	Optional callback to get the initialization status of SensorManager telux::common::InitResponseCb

Returns

An instance of ISensorManager If the initialization of the manager and underlying system fails, nullptr is returned

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.38.1.8.1.3 virtual std::shared_ptr<ISensorFeatureManager> telux::sensor::SensorFactory::getSensorFeatureManager (telux::common::InitResponseCb *clientCallback* = `nullptr`) [pure virtual]

Get an instance of Sensor Feature Manager. The ownership of the returned object is with the caller of this method. The reference to the instance is not held by the SensorFactory. If the returned reference is released, any request for ISensorFeatureManager shall result in creation of a new instance

Parameters

in	*clientCallback*	Optional callback to get the initialization status of SensorFeatureManager telux::common::InitResponseCb

Returns

An instance of ISensorFeatureManager If the initialization of the manager and underlying system fails, nullptr is returned

4.38.1.9 union telux::sensor::SensorEvent.__unnamed__

Sensor data

Data fields

Type	Field	Description
MotionSensorData	calibrated	Calibrated data - should be accessed when the SensorType that generated the sensor event accounts for calibration - SensorType::ACCELEROMETER or SensorType::GYROSCOPE
UncalibratedMotionSensorData	uncalibrated	Uncalibrated data - should be accessed when the SensorType that generated the sensor event provides uncalibrated data along with bias information - SensorType::ACCELEROMETER_UNCALIBRATED or SensorType::GYROSCOPE_UNCALIBRATED

4.38.2 Enumeration Type Documentation

4.38.2.1 enum telux::sensor::SensorType [strong]

Enumeration of different sensors available.

Enumerator

ACCELEROMETER ID for the accelerometer sensor
GYROSCOPE ID for the gyroscope sensor
GYROSCOPE_UNCALIBRATED ID for the uncalibrated gyroscope sensor
ACCELEROMETER_UNCALIBRATED ID for the uncalibrated accelerometer sensor
INVALID Denotes that the sensor type is either unknown or invalid

4.38.2.2 enum telux::sensor::SensorConfigParams

Enumeration listing the different configuration parameters in SensorConfiguration

Enumerator

SAMPLING_RATE Corresponds to SensorConfiguration::samplingRate
BATCH_COUNT Corresponds to SensorConfiguration::batchCount
SENSOR_CONFIG_NUM_PARAMS

4.38.2.3 enum telux::sensor::SelfTestType [strong]

Types of self test the sensor can perform.

Enumerator

POSITIVE To initiate self test with positive values
NEGATIVE To initiate self test with negative values

4.39 Sensor Control

This section contains APIs related to sensor configuration, sensor control and data acquisition from sensors.

4.39.1 Data Structure Documentation

4.39.1.1 class telux::sensor::ISensorEventListener

ISensorEventListener interface is used to receive notifications related to sensor events and configuration updates.

The listener method can be invoked from multiple different threads. Client needs to make sure that implementation is thread-safe.

Public member functions

- virtual void onEvent (std::shared_ptr< std::vector< SensorEvent >> events)
- virtual void onConfigurationUpdate (SensorConfiguration configuration)
- virtual ~ISensorEventListener ()

4.39.1.1.1 Constructors and Destructors

4.39.1.1.1.1 virtual telux::sensor::ISensorEventListener::~ISensorEventListener ( ) [virtual]

The destructor for the sensor event listener

4.39.1.1.2 Member Function Documentation

4.39.1.1.2.1 virtual void telux::sensor::ISensorEventListener::onEvent (std::shared_ptr< std::vector< SensorEvent >> *events*) [virtual]

This function is called to notify about available sensor events. Note the following constraints on this listener API It shall not perform time consuming (compute or I/O intensive) operations on this thread It shall not inovke an sensor APIs on this thread due to the underlying concurrency model

On platforms with Access control enabled, the client needs to have TELUX_SENSOR_DATA_READ permission for this listener API to be invoked.

Parameters

in	*events*	- List of sensor events

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.39.1.1.2.2 virtual void telux::sensor::ISensorEventListener::onConfigurationUpdate (SensorConfiguration *configuration*) [virtual]

This function is called to notify any change in the configuration of the ISensorClient object this listener is associated with.

On platforms with Access control enabled, the client needs to have TELUX_SENSOR_DATA_READ permission for this listener API to be invoked.

Parameters

in	*configuration*	- The new configuration of the sensor client. telux::sensor::SensorConfiguration. Fields that have changed can be identified using the telux::sensor::SensorConfiguration::updateMask and fields that are valid can be identified using telux::sensor::SensorConfiguration::validityMask

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.39.1.2 class telux::sensor::ISensorClient

ISensorClient interface is used to access the different services provided by the sensor framework to configure, activate and acquire sensor data.

Each instance of this class is a unique sensor client to the underlying sensor framework and any number of such clients can exist in a given process. Each of these clients can acquire data from the underlying sensor framework with different configurations.

Public member functions

- virtual SensorInfo getSensorInfo ()=0
- virtual telux::common::Status configure (SensorConfiguration configuration)=0
- virtual SensorConfiguration getConfiguration ()=0
- virtual telux::common::Status activate ()=0
- virtual telux::common::Status deactivate ()=0
- virtual telux::common::Status selfTest (SelfTestType selfTestType, SelfTestResultCallback cb)=0
- virtual telux::common::Status registerListener (std::weak_ptr< ISensorEventListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< ISensorEventListener > listener)=0
- virtual ~ISensorClient ()
- virtual telux::common::Status enableLowPowerMode ()=0
- virtual telux::common::Status disableLowPowerMode ()=0

4.39.1.2.1 Constructors and Destructors

4.39.1.2.1.1 virtual telux::sensor::ISensorClient::~ISensorClient ( ) [virtual]

Destructor for ISensorClient.

Internally, the sensor client object is first deactivated and then destroyed.

4.39.1.2.2 Member Function Documentation

4.39.1.2.2.1 virtual SensorInfo telux::sensor::ISensorClient::getSensorInfo ( ) [pure virtual]

Get the information related to sensor

Returns

information retated to sensor - telux::sensor::SensorInfo

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.39.1.2.2.2 virtual telux::common::Status telux::sensor::ISensorClient::configure (SensorConfiguration *configuration*) [pure virtual]

Configure the sensor client with desired sampling rate and batch count. Any change in sampling rate or batch count of the sensor will be notified via telux::sensor::ISensorEventListener::onConfigurationUpdate.

In case a sensor client needs to be reconfigured after having been activated, the client should be deactivated, configured and activated again as a part of the reconfiguration process.

It is always recommended that configuration of a client is done before activating it. If a client is activated without configuration, the client is configured with a default configuration and activated. The default configuration would have the sampling rate set to minimum sampling rate supported telux::sensor::SensorInfo::samplingRates and the batch count set to maximum batch count supported telux::sensor::SensorInfo::maxBatchCountSupported

On platforms with Access control enabled, Caller needs to have TELUX_SENSOR_DATA_READ permission to invoke this API successfully.

Parameters

in	*configuration*	- The desired configuration for the client telux::sensor::SensorConfiguration. Ensure the required validity mask telux::sensor::SensorConfiguration::validityMask is set for the configuration.

Returns

status of configuration request - telux::common::Status

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.39.1.2.2.3 virtual SensorConfiguration telux::sensor::ISensorClient::getConfiguration ( ) [pure virtual]

Get the current configuration of this sensor client

On platforms with Access control enabled, Caller needs to have TELUX_SENSOR_DATA_READ permission to invoke this API successfully.

Returns

the current configuration of the client. telux::sensor::SensorConfiguration::validityMask should be checked to know which of the fields in the returned configuration is valid.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.39.1.2.2.4 virtual telux::common::Status telux::sensor::ISensorClient::activate ( ) [pure virtual]

Activate the sensor client. Once activated, any available sensor event will be notified via telux::sensor::ISensorEventListener::onEvent

It is always recommended that configuration of a client is done before activating it. If a client is activated without configuration, the client is configured with the default configuration and activated. The default configuration would have the sampling rate set to minimum sampling rate supported telux::sensor::SensorInfo::samplingRates and the batch count set to maximum batch count supported telux::sensor::SensorInfo::maxBatchCountSupported. Activating an already activated sensor would result in the API returning telux::common::Status::SUCCESS.

Activating this sensor client would not impact other inactive sensor clients.

On platforms with Access control enabled, Caller needs to have TELUX_SENSOR_DATA_READ permission to invoke this API successfully.

Returns

status of activation request - telux::common::Status

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.39.1.2.2.5 virtual telux::common::Status telux::sensor::ISensorClient::deactivate ( ) [pure virtual]

Deactivate the sensor client. Once deactivated, no further sensor events will be notified via telux::sensor::ISensorEventListener::onEvent. Deactivating an already inactive sensor would result in the API returning telux::common::Status::SUCCESS.

Deactivating this sensor client would not impact other active sensor clients.

On platforms with Access control enabled, Caller needs to have TELUX_SENSOR_DATA_READ permission to invoke this API successfully.

Returns

status of deactivation request - telux::common::Status

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.39.1.2.2.6 virtual telux::common::Status telux::sensor::ISensorClient::selfTest (SelfTestType *selfTestType,* SelfTestResultCallback *cb*) [pure virtual]

Initiate self test on this sensor

If there are active data acquisition sessions corresponding to this sensor, these will be paused and the self test is initiated. Once the self test is complete the sensor data sessions will be restored.

On platforms with Access control enabled, Caller needs to have TELUX_SENSOR_PRIVILEGED_OPS permission to invoke this API successfully.

Parameters

in	*selfTestType*	- The type of self test to be performed - telux::sensor::SelfTestType
in	*cb*	- Callback to get the result of the self test initiated

Returns

status of the request - telux::common::Status. Note that the result of the self test done by the sensor is provided via the callback - telux::sensor::SelfTestResultCallback

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.39.1.2.2.7 virtual telux::common::Status telux::sensor::ISensorClient::registerListener (std::weak_ptr< ISensorEventListener > *listener*) [pure virtual]

Register a listener for sensor related events

Returns

status of registration request - telux::common::Status

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.39.1.2.2.8 virtual telux::common::Status telux::sensor::ISensorClient::deregisterListener (std::weak_ptr< ISensorEventListener > *listener*) [pure virtual]

Deregister a sensor event listener

Returns

status of deregistration request - telux::common::Status

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.39.1.2.2.9 virtual telux::common::Status telux::sensor::ISensorClient::enableLowPowerMode ( ) [pure virtual]

Deprecated APIs Request the sensor to operate in low power mode. The sensor should be in deactivated state to exercise this API. The success of this request depends on the capabilities of the underlying hardware.

Returns

status of request - telux::common::Status

Deprecated

This API is no longer supported.

4.39.1.2.2.10 virtual telux::common::Status telux::sensor::ISensorClient::disableLowPowerMode ( ) [pure virtual]

Request the sensor to exit low power mode. The sensor should be in deactivated state to exercise this API. The success of this request depends on the capabilities of the underlying hardware.

Returns

status of request - telux::common::Status

Deprecated

This API is no longer supported.

4.39.1.3 class telux::sensor::ISensorManager

Sensor Manager class provides APIs to interact with the sensor sub-system and get access to other sensor objects which can be used to configure, activate or get data from the individual sensors available - Gyro, Accelero, etc.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual telux::common::Status getAvailableSensorInfo (std::vector< SensorInfo > &info)=0
- virtual telux::common::Status getSensor (std::shared_ptr< ISensorClient > &sensor, std::string name)=0
- virtual telux::common::Status getSensorClient (std::shared_ptr< ISensorClient > &sensor, std::string name)=0
- virtual ~ISensorManager ()

4.39.1.3.1 Constructors and Destructors

4.39.1.3.1.1 virtual telux::sensor::ISensorManager::~ISensorManager ( ) [virtual]

Destructor for ISensorManager

4.39.1.3.2 Member Function Documentation

4.39.1.3.2.1 virtual telux::common::ServiceStatus telux::sensor::ISensorManager::getServiceStatus ( ) [pure virtual]

Checks the status of sensor sub-system and returns the result.

Returns

the status of sensor sub-system status telux::common::ServiceStatus

4.39.1.3.2.2 virtual telux::common::Status telux::sensor::ISensorManager::getAvailableSensorInfo (std::vector< SensorInfo > & *info*) [pure virtual]

Get information related to the sensors available in the system.

Parameters

out	*info*	List of information on sensors available in the system telux::sensor::SensorInfo

Returns

status of the request telux::common::Status

4.39.1.3.2.3 virtual telux::common::Status telux::sensor::ISensorManager::getSensor (std::shared_ptr< ISensorClient > & *sensor,* std::string *name*) [pure virtual]

Get an instance of ISensorClient to interact with the underlying sensor. The provided instance is not a singleton. Everytime this method is called a new sensor object is created. It is the caller's responsibility to manage the object's lifetime. Every instance of the sensor returned acts as new client and can configure the underlying sensor with it's own configuration and it's own callbacks for telux::sensor::SensorEvent and configuration update among other events telux::sensor::ISensorEventListener.

Parameters

out	*sensor*	- An instance of telux::sensor::ISensorClient to interact with the underlying sensor is provided as a result of the method If the initialization of the sensor and underlying system fails, sensor is set to nullptr
in	*name*	- The unique name of the sensor telux::sensor::SensorInfo::name that was provided in the list of sensor information by telux::sensor::ISensorManager::getAvailableSensorInfo

Returns

Status of request telux::common::Status

Deprecated

Use getSensorClient API.

4.39.1.3.2.4 virtual telux::common::Status telux::sensor::ISensorManager::getSensorClient (std::shared_ptr< ISensorClient > & *sensor,* std::string *name*) [pure virtual]

Get an instance of ISensorClient to interact with the underlying sensor. The provided instance is not a singleton. Everytime this method is called a new sensor object is created. It is the caller's responsibility to manage the object's lifetime. Every instance of the sensor returned acts as new client and can configure the underlying sensor with it's own configuration and it's own callbacks for telux::sensor::SensorEvent and configuration update among other events telux::sensor::ISensorEventListener.

Parameters

out	*sensor*	- An instance of telux::sensor::ISensorClient to interact with the underlying sensor is provided as a result of the method If the initialization of the sensor and underlying system fails, sensor is set to nullptr
in	*name*	- The unique name of the sensor telux::sensor::SensorInfo↩ ::name that was provided in the list of sensor information by telux::sensor::ISensorManager::getAvailableSensorInfo

Returns

Status of request telux::common::Status

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.40 Sensor Feature Control

This section contains APIs related to contrlling the features the sensor sub-system offers.

4.40.1 Data Structure Documentation

4.40.1.1 class telux::sensor::ISensorFeatureEventListener

ISensorFeatureEventListener interface is used to receive notifications related to sensor feature events.

The listener method can be invoked from multiple different threads. Client needs to make sure that implementation is thread-safe.

Public member functions

- virtual void onEvent (SensorFeatureEvent event)
- virtual void onBufferedEvent (std::string sensorName, std::shared_ptr< std::vector< SensorEvent >> events, bool isLast)
- virtual ~ISensorFeatureEventListener ()

4.40.1.1.1 Constructors and Destructors

4.40.1.1.1.1 virtual telux::sensor::ISensorFeatureEventListener::~ISensorFeatureEventListener ( ) [virtual]

The destructor for the sensor feature event listener

4.40.1.1.2 Member Function Documentation

4.40.1.1.2.1 virtual void telux::sensor::ISensorFeatureEventListener::onEvent (SensorFeatureEvent *event*) [virtual]

This function is called to notify about sensor feature events

Parameters

in	*event*	- The sensor feature event telux::sensor::SensorFeatureEvent that got triggered

On platforms with Access control enabled, the client needs to have TELUX_SENSOR_FEATURE_CONTROL permission for this listener API to be invoked.

4.40.1.1.2.2 virtual void telux::sensor::ISensorFeatureEventListener::onBufferedEvent (std::string *sensorName,* std::shared_ptr< std::vector< SensorEvent >> *events,* bool *isLast*) [virtual]

This function is called to notify about available sensor events that caused one or more sensor feature events SensorFeatureEvent to occur.

The sensor events that occurred when the apps processor was in sleep mode and triggered the sensor feature to occur will be buffered and delivered using this method instead of telux::sensor::ISensorEventListener::onEvent.

In case a sensor event occurs when the system is active, this listener is not invoked. In this case, the required sensor data that triggered the feature can be obtained from the telux::sensor::ISensorEventListener::onEvent listener interface.

Note the following constraints on this listener API It shall not perform time consuming (compute or I/O intensive) operations on this thread It shall not inovke an sensor APIs on this thread due to the underlying concurrency model

On platforms with Access control enabled, the client needs to have TELUX_SENSOR_FEATURE_CONTROL permission for this listener API to be invoked.

Parameters

in	*sensorName*	- The name of the sensor that generated the buffered events
in	*events*	- List of sensor events
in	*isLast*	- Indicate if this is last notification for the buffered events. Multiple @ref telux::sensor::SensorFeature can be @ref telux::sensor::enableFeature, whose notifica in sequence. isLast will be set to true to signify last event

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.40.1.2 class telux::sensor::ISensorFeatureManager

Sensor Feature Manager class provides APIs to interact with the sensor framework to list the available features, enable them or disable them. The availability of sensor features depends on the capabilities of the underlying hardware.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual telux::common::Status getAvailableFeatures (std::vector< SensorFeature > &features)=0
- virtual telux::common::Status enableFeature (std::string name)=0
- virtual telux::common::Status disableFeature (std::string name)=0
- virtual telux::common::Status registerListener (std::weak_ptr< ISensorFeatureEventListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< ISensorFeatureEventListener > listener)=0
- virtual ~ISensorFeatureManager ()

4.40.1.2.1 Constructors and Destructors

4.40.1.2.1.1 virtual telux::sensor::ISensorFeatureManager::~ISensorFeatureManager ( ) [virtual]

Destructor for ISensorFeatureManager

4.40.1.2.2 Member Function Documentation

4.40.1.2.2.1 virtual telux::common::ServiceStatus telux::sensor::ISensorFeatureManager::getServiceStatus ( ) [pure virtual]

Checks the status of sensor sub-system and returns the result.

Returns

the status of sensor sub-system status telux::common::ServiceStatus

4.40.1.2.2.2 virtual telux::common::Status telux::sensor::ISensorFeatureManager::getAvailableFeatures (std::vector< SensorFeature > & *features*) [pure virtual]

Request the sensor framework to provide the available features. The feature could be offered by the sensor framework or the underlying hardware.

Parameters

out	*features*	List of sensor features the sensor framework offers

Returns

status of the request telux::common::Status

4.40.1.2.2.3 virtual telux::common::Status telux::sensor::ISensorFeatureManager::enableFeature (std::string *name*) [pure virtual]

Enable the requested feature.

Enabling a sensor feature when the system is active would additionally require enabling the corresponding sensor which is used by the sensor feature. For instance, if the sensor feature uses the accelerometer data, in addition to calling this method, the telux::sensor::ISensorClient::activate should also be invoked for the required sensor, in this case, the accelerometer.

If the sensor feature only needs to be enabled during suspend mode, just enabling the sensor feature using this method would be sufficient. The underlying framework would take care to enable the required sensor when the system is about to enter suspend state.

On platforms with Access control enabled, Caller needs to have TELUX_SENSOR_FEATURE_CONTROL permission to invoke this API successfully.

Parameters

in	*name*	The name of the feature to be enabled. Enabling an already enabled feature would result in the API returning telux::common::Status::SUCCESS.

Returns

status of the request telux::common::Status

4.40.1.2.2.4 virtual telux::common::Status telux::sensor::ISensorFeatureManager::disableFeature (std::string *name*) [pure virtual]

Disable the requested feature

Parameters

in	*name*	The name of the feature to be disabled. Disabling an already disabled feature would result in the API returning telux::common::Status::SUCCESS. On platforms with Access control enabled, Caller needs to have TELUX_SENSOR_FEATURE_CONTROL permission to invoke this API successfully.

Returns

status of the request telux::common::Status

4.40.1.2.2.5 virtual telux::common::Status telux::sensor::ISensorFeatureManager::registerListener (std::weak_ptr< ISensorFeatureEventListener > *listener*) [pure virtual]

Register a listener for sensor feature related events

Returns

status of registration request - telux::common::Status

4.40.1.2.2.6 virtual telux::common::Status telux::sensor::ISensorFeatureManager::deregisterListener (std::weak_ptr< ISensorFeatureEventListener > *listener*) [pure virtual]

Deregister a sensor feature event listener

Returns

status of deregistration request - telux::common::Status

4.41 Platform

- Filesystem

This section contains APIs related to configure platform functionalities and acquire information from the sub-components.

4.41.1 Data Structure Documentation

4.41.1.1 class telux::platform::PlatformFactory

PlatformFactory allows creation of Platform services related classes.

Public member functions

- virtual std::shared_ptr< IFsManager > getFsManager (telux::common::InitResponseCb callback=nullptr)=0
- virtual std::shared_ptr< IDeviceInfoManager > getDeviceInfoManager (telux::common::InitResponseCb callback=nullptr)=0

Static Public Member Functions

- static PlatformFactory & getInstance ()

4.41.1.1.1 Member Function Documentation

4.41.1.1.1.1 static PlatformFactory& telux::platform::PlatformFactory::getInstance ( ) [static]

Get instance of platform Factory

4.41.1.1.1.2 virtual std::shared_ptr<IFsManager> telux::platform::PlatformFactory::getFsManager (telux::common::InitResponseCb *callback* = `nullptr`) [pure virtual]

Get instance of filesystem manager (IFsManager). The filesystem manager supports notification of filesystem events like EFS restore indications.

Parameters

in	*callback*	Optional callback to get the initialization status of FsManager. telux::common::InitResponseCb

Returns

pointer of IFsManager object.

4.41.1.1.1.3 virtual std::shared_ptr<IDeviceInfoManager> telux::platform::PlatformFactory::getDeviceInfoManager (telux::common::InitResponseCb *callback* = `nullptr`) [pure virtual]

Get instance of device info manager (IDeviceInfoManager). The device info manager supports device info request like retrieving IMEI and platform version.

Parameters

in	*callback*	Optional callback to get the initialization status of FsManager. telux::common::InitResponseCb

Returns

pointer of IDeviceInfoManager object.

4.42 Filesystem

This section contains APIs, data structures and components to configure and acquire information from the filesystem manager.

4.42.1 Data Structure Documentation

4.42.1.1 struct telux::platform::EfsEventInfo

Data fields

Type	Field	Description
EfsEvent	event	The event being notified
ErrorCode	error	telux::common::ErrorCode associated with the event

4.42.1.2 class telux::platform::IFsListener

Listener class for getting notifications related to EFS backup/restore operations. The client needs to implement these methods as briefly as possible and avoid blocking calls in it. The methods in this class can be invoked from multiple different threads. Client needs to make sure that the implementation is thread-safe.

Public member functions

- virtual void OnEfsRestoreEvent (EfsEventInfo event)
- virtual void OnEfsBackupEvent (EfsEventInfo event)
- virtual void OnFsOperationImminentEvent (uint32_t timeLeftToStart)
- virtual ~IFsListener ()

4.42.1.2.1 Constructors and Destructors

4.42.1.2.1.1 virtual telux::platform::IFsListener::~IFsListener ( ) [virtual]

Destructor of IFsListener

4.42.1.2.2 Member Function Documentation

4.42.1.2.2.1 virtual void telux::platform::IFsListener::OnEfsRestoreEvent (EfsEventInfo *event*) [virtual]

This function is called when a EFS restore operation is detected.

On platforms with Access control enabled, the client needs to have TELUX_PLATFORM_LISTEN_FS_EVENTS permission to receive this event.

Parameters

in	*event*	Event related data. telux::platform::EfsEventInfo.

4.42.1.2.2.2 virtual void telux::platform::IFsListener::OnEfsBackupEvent (EfsEventInfo *event*) [virtual]

This function is called when a EFS backup operation is detected.

On platforms with Access control enabled, the client needs to have TELUX_PLATFORM_LISTEN_FS_EVENTS permission to receive this event.

Parameters

in	*event*	Event related data. telux::platform::EfsEventInfo.

4.42.1.2.2.3 virtual void telux::platform::IFsListener::OnFsOperationImminentEvent (uint32_t *timeLeftToStart*) [virtual]

When the client is about to make an eCall it is expected to invoke prepareForEcall. This starts a timer within the FsManager which represents the max duration of the eCall. After which the filesystem operations will resume. This API will be invoked to let the client know that resumption of Fs operations is imminent. If the eCall has not yet ended, the client should call prepareForEcall again to reset the timer, which will continue to suspend the FS operations.

Parameters

in	*timeLeftToStart*	The time in seconds after which filesystem operations shall re-enable.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.42.1.3 class telux::platform::IFsManager

IFsManager provides interface to to control and get notified about file system operations. This includes Embedded file system (EFS) operations.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual telux::common::Status registerListener (std::weak_ptr< IFsListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< IFsListener > listener)=0
- virtual telux::common::Status startEfsBackup ()=0
- virtual telux::common::Status prepareForEcall ()=0

- virtual telux::common::Status eCallCompleted ()=0
- virtual telux::common::Status prepareForOta (OtaOperation otaOperation, telux::common::ResponseCallback responseCb)=0
- virtual telux::common::Status otaCompleted (OperationStatus operationStatus, telux::common::ResponseCallback responseCb)=0
- virtual telux::common::Status startAbSync (telux::common::ResponseCallback responseCb)=0
- virtual ~IFsManager ()

4.42.1.3.1 Constructors and Destructors

4.42.1.3.1.1 virtual telux::platform::IFsManager::~IFsManager ( ) [virtual]

Destructor of IFsManager

4.42.1.3.2 Member Function Documentation

4.42.1.3.2.1 virtual telux::common::ServiceStatus telux::platform::IFsManager::getServiceStatus ( ) [pure virtual]

This status indicates whether the object is in a usable state.

Returns

telux::common::ServiceStatus indicating the current status of the file system service.

4.42.1.3.2.2 virtual telux::common::Status telux::platform::IFsManager::registerListener (std::weak_ptr< IFsListener > *listener*) [pure virtual]

Registers the listener for FileSystem Manager indications.

Parameters

in	*listener*	- pointer to implemented listener.

Returns

status of the registration request.

4.42.1.3.2.3 virtual telux::common::Status telux::platform::IFsManager::deregisterListener (std::weak_ptr< IFsListener > *listener*) [pure virtual]

Deregisters the previously registered listener.

Parameters

in	*listener*	- pointer to registered listener that needs to be removed.

Returns

status of the deregistration request.

4.42.1.3.2.4 virtual telux::common::Status telux::platform::IFsManager::startEfsBackup ( ) [pure virtual]

Request to trigger an EFS backup. If the request is successful, the status of EFS backup is notified via telux::platform::IFsListener::OnEfsBackupEvent.

On platforms with Access control enabled, Caller needs to have TELUX_PLATFORM_FS_OPS_CTRL permission to invoke this API successfully.

Returns

The status of the request - telux::common::Status

4.42.1.3.2.5 virtual telux::common::Status telux::platform::IFsManager::prepareForEcall ( ) [pure virtual]

The Filesystem Manager performs periodic operations which might be resource intensive. Such operations are not desired during other crucial events like an eCall. To avoid performing such operations during such events, the client is recommended to invoke this API before it initiates an eCall. This allows the filesystem manager to prepare the system to restrict any resource intensive operations like filesystem scrubbing during the eCall.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_ECALL_MGMT permission to invoke this API successfully.

Note

- The client would need to periodically invoke this API to ensure that the timer gets reset so that operations do not get re-enabled.

Returns

- telux::common::Status

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.42.1.3.2.6 virtual telux::common::Status telux::platform::IFsManager::eCallCompleted ( ) [pure virtual]

Once ecall complete, the client should invoke this API to re-enable filesystem operations like filesystem scrubbing.If the API invocation results in telux::common::Status::NOTREADY,indicating that the sub-system is not ready, the client should retry.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_ECALL_MGMT permission to invoke this API successfully.

Returns

- telux::common::Status

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.42.1.3.2.7 virtual telux::common::Status telux::platform::IFsManager::prepareForOta (OtaOperation *otaOperation,* telux::common::ResponseCallback *responseCb*) [pure virtual]

This API should be invoked to allow the filesystem manager to perform operations like prepare the filesystem for an OTA. In addition to this preparation, any on-going operations like scrubbing is stopped.

On platforms with Access control enabled, Caller needs to have TELUX_PLATFORM_OTA_MGMT permission to invoke this API successfully.

Parameters

in	*otaOperation*	- telux::platform::OtaOperation.
out	*responseCb*	- telux::common::ResponseCallback The callback method to be invoked upon completion of OTA preparation and the response is indicated asynchronously.

Returns

- telux::common::Status

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.42.1.3.2.8 virtual telux::common::Status telux::platform::IFsManager::otaCompleted (Operation↩ Status *operationStatus,* telux::common::ResponseCallback *responseCb*) [pure virtual]

This API should be invoked upon completion of OTA, this will allow the filesystem manager to perform post OTA verifications and re-enable operations that were disabled for performing the OTA, like scrubbing.

On platforms with Access control enabled, Caller needs to have TELUX_PLATFORM_OTA_MGMT permission to invoke this API successfully.

Parameters

in	*operationStatus*	- telux::platform::OperationStatus The status of the OTA operation that the client attempted.
out	*responseCb*	- telux::common::ResponseCallback The callback method to be invoked upon completion of OTA related filesystem verifications and the response is indicated asynchronously.

Returns

- telux::common::Status

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.42.1.3.2.9 virtual telux::common::Status telux::platform::IFsManager::startAbSync (telux↩ ::common::ResponseCallback *responseCb*) [pure virtual]

This API should be invoked when the client decides to mirror the active partition to the inactive partition.

On platforms with Access control enabled, Caller needs to have TELUX_PLATFORM_OTA_MGMT permission to invoke this API successfully.

Parameters

out	*responseCb*	- telux::common::ResponseCallback The callback method to be invoked when the mirroring operation is completed and the response is indicated asynchronously.

Returns

- telux::common::Status

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.42.2 Enumeration Type Documentation

4.42.2.1 enum telux::platform::EfsEvent [strong]

Enumerator

START Indicating the beginning of Backup/Restore operation
END Indicating the completion of Backup/Restore operation

4.42.2.2 enum telux::platform::OperationStatus [strong]

Enumerator

UNKNOWN
SUCCESS
FAILURE

4.42.2.3 enum telux::platform::OtaOperation [strong]

Enumerator

INVALID
START
RESUME

4.43 Wlan

- WLAN Device Management
- Access Point Management
- Station Management

This section contains APIs related to WLAN management operations.

4.44 WLAN Device Management

This section contains APIs related to device configuration management, such as the number of access points and stations enabled, WLAN enable/disable, etc

4.44.1 Data Structure Documentation

4.44.1.1 struct telux::wlan::ApInfo

AP Info - captures ap type (private/guest)

Data fields

Type	Field	Description
ApType	apType	Ap type (private/guest)

4.44.1.2 struct telux::wlan::ApNetInfo

Ap Network Info

Data fields

Type	Field	Description
ApInfo	info	Ap information (AP type)

4.44.1.3 struct telux::wlan::ApStatus

AP Status for enabled Networks

Data fields

Type	Field	Description
Id	id	AP id
string	name	AP network interface name
string	ipv4Address	Local AP IP V4 address
string	macAddress	AP MAC address
vector< ApNetInfo >	network	Settings for AP info

4.44.1.4 struct telux::wlan::StaStatus

Station Status

Data fields

Type	Field	Description
Id	id	Station Id
string	name	Network interface name
string	ipv4Address	Public IP V4 address

Type	Field	Description
string	ipv6Address	Public IP V6 address
string	macAddress	MAC address
StaInterface↩ Status	status	Interface status

4.44.1.5 struct telux::wlan::InterfaceStatus

Wlan Interface status

Data fields

Type	Field	Description
HwDeviceType	device	WiFi hardware type
vector< Ap↩ Status >	apStatus	Vector of active APs status
vector< Sta↩ Status >	staStatus	Vector of active Sta status

4.44.1.6 class telux::wlan::IWlanDeviceManager

WlanDeviceManager is a primary interface for configuring Wireless LAN. it provide APIs to enable, configure, activate, and modify modes.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual telux::common::ErrorCode enable (bool enable)=0
- virtual telux::common::ErrorCode setMode (int numOfAp, int numOfSta)=0
- virtual telux::common::ErrorCode getConfig (int &numAp, int &numSta)=0
- virtual telux::common::ErrorCode getStatus (bool &isEnabled, std::vector< InterfaceStatus > &status)=0
- virtual telux::common::ErrorCode registerListener (std::weak_ptr< IWlanListener > listener)=0
- virtual telux::common::ErrorCode deregisterListener (std::weak_ptr< IWlanListener > listener)=0
- virtual ~IWlanDeviceManager ()

4.44.1.7 class telux::wlan::IWlanListener

Public member functions

- virtual void onServiceStatusChange (telux::common::ServiceStatus status)
- virtual void onEnableChanged (bool enable)
- virtual ~IWlanListener ()

4.44.1.8 class telux::wlan::WlanFactory

WlanFactory is the central factory to create all wlan classes.

Public member functions

- virtual std::shared_ptr< IWlanDeviceManager > getWlanDeviceManager (telux::common::InitResponseCb clientCallback=nullptr)=0
- virtual std::shared_ptr< IApInterfaceManager > getApInterfaceManager ()=0
- virtual std::shared_ptr< IStaInterfaceManager > getStaInterfaceManager ()=0

Static Public Member Functions

- static WlanFactory & getInstance ()

Protected Member Functions

- WlanFactory ()
- virtual ~WlanFactory ()

4.44.1.8.1 Constructors and Destructors

4.44.1.8.1.1 telux::wlan::WlanFactory::WlanFactory ( ) [protected]

4.44.1.8.1.2 virtual telux::wlan::WlanFactory::~WlanFactory ( ) [protected], [virtual]

4.44.1.8.2 Member Function Documentation

4.44.1.8.2.1 static WlanFactory& telux::wlan::WlanFactory::getInstance ( ) [static]

Get Wlan Factory instance.

4.44.1.8.2.2 virtual std::shared_ptr<IWlanDeviceManager> telux::wlan::WlanFactory::getWlanDeviceManager (telux::common::InitResponseCb *clientCallback* = `nullptr`) [pure virtual]

Get Wlan Device Manager

Parameters

in	*clientCallback*	Optional callback to get the initialization status of WlanDeviceManager telux::common::InitResponseCb

Returns

instance of IWlanDeviceManager

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.44.1.8.2.3 virtual std::shared_ptr<IApInterfaceManager> telux::wlan::WlanFactory::getApInterfaceManager ( ) [pure virtual]

Get Access Point Interface Manager

Returns

instance of IApInterfaceManager

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.44.1.8.2.4 virtual std::shared_ptr<IStaInterfaceManager> telux::wlan::WlanFactory::getStaInterfaceManager ( ) [pure virtual]

Get Station Interface Manager

Returns

instance of IStaInterfaceManager

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.44.2 Enumeration Type Documentation

4.44.2.1 enum telux::wlan::BandType [strong]

Radio Band Types:

Enumerator

BAND_5GHZ
BAND_2GHZ

4.44.2.2 enum telux::wlan::ConnectionStatus [strong]

Connection Status

Enumerator

UNKNOWN Device connection is unknown

CONNECTED Device is connected
DISCONNECTED Device is disconnected

4.44.2.3 enum telux::wlan::Id [strong]

Identifiers for Ap, Sta, P2p

Enumerator

PRIMARY
SECONDARY
TERTIARY
QUATERNARY

4.44.2.4 enum telux::wlan::ApType [strong]

AP Types:

Enumerator

UNKNOWN
PRIVATE
GUEST

4.44.2.5 enum telux::wlan::StaInterfaceStatus [strong]

Station Interface Status

Enumerator

UNKNOWN Station interface is unknown
CONNECTING Station interface is connecting
CONNECTED Station interface is connected
DISCONNECTED Station interface is disconnected
ASSOCIATION_FAILED Station is unable to associate with AP
IP_ASSIGNMENT_FAILED Station in unable to get IP address via DHCP

4.44.2.6 enum telux::wlan::OperationType [strong]

This applies in architectures where the modem is attached to an External Application Processor(EAP). An API that sets or configure Wlan can be invoked from the EAP or from the modems Internal Application Processor (IAP). This type specifies where the operation should be carried out.

Enumerator

WLAN_LOCAL Perform the operation on the processor where the API is invoked.
WLAN_REMOTE Perform the operation on the application processor other than where the API is invoked.

4.44.2.7 enum telux::wlan::IpFamilyType [strong]

Preferred IP family for the connection

Enumerator

UNKNOWN
IPV4 IPv4 data connection
IPV6 IPv6 data connection
IPV4V6 IPv4 and IPv6 data connection

4.44.2.8 enum telux::wlan::ServiceOperation [strong]

Service operations to be performed

Enumerator

STOP Stop service
START Start service
RESTART Restart service

4.44.2.9 enum telux::wlan::InterfaceState [strong]

Wlan Interface State

Enumerator

INACTIVE Interface is Inactive
ACTIVE Interface is Active

4.44.2.10 enum telux::wlan::HwDeviceType [strong]

Wlan Interface Device

Enumerator

UNKNOWN Wlan device is Unknown
QCA6574 Wlan device is QCA6574
QCA6696 Wlan device is QCA6696
QCA6595 Wlan device is QCA6595

4.44.3 Function Documentation

4.44.3.1 virtual telux::common::ServiceStatus telux::wlan::IWlanDeviceManager↩ ::getServiceStatus ( ) [pure virtual]

Checks the readiness status of wlan manager and returns the result.

Returns

SERVICE_AVAILABLE - If wlan manager is ready for service. SERVICE_UNAVAILABLE - If

wlan manager is temporarily unavailable. SERVICE_FAILED - If wlan manager encountered an irrecoverable failure.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.44.3.2 virtual telux::common::ErrorCode telux::wlan::IWlanDeviceManager::enable (bool *enable*) [pure virtual]

Enable or Disable Wlan Service. Configurations set by telux::wlan::IWlanDeviceManager::setMode must be completed before enabling Wlan. If any of configurations need to be changed after Wlan is enabled, this API must be called with enable set to false followed by a call with enable set to true for the new configurations to take effect. Calling this API with enable, will start hostapd and wpa_supplicant daemons. Further changes to hostapd and wpa_supplicant will require calling telux::wlan::IApInterfaceManager::manageApService and telux::wlan::IStaInterfaceManager::manageStaService respectively. Client shall wait for IWlanListener::onEnableChanged indication to confirm WLAN was enabled/disabled successfully

On platforms with Access control enabled, Caller needs to have TELUX_WLAN_DEVICE_CONFIG permission to invoke this API successfully.

Parameters

in	*enable*	true : Enable Wlan, false: Disable Wlan.

Returns

operation error code (if any). telux::common::ErrorCode

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.44.3.3 virtual telux::common::ErrorCode telux::wlan::IWlanDeviceManager::setMode (int *numOfAp,* int *numOfSta*) [pure virtual]

Set Wlan mode - number of supported APs, and stations. This API shall be called when wlan is disabled. On enablement, wlan will enable APs and Stations set in this API.

Parameters

in	*numOfAp*	Num of Access Points to be enabled. If no Access Point is enabled, this argument should be set to 0. Configuration of each AP is accomplished through telux::data::wlan::IApManager instance requested from factory.
in	*numOfSta*	Num of Stations to be enabled. If no station is enabled, this argument should be set to 0. Configuration of each Station is accomplished through telux::data::wlan::IStaManager instance requested from factory.

On platforms with Access control enabled, Caller needs to have TELUX_WLAN_DEVICE_CONFIG permission to invoke this API successfully.

Returns

operation error code (if any). telux::common::ErrorCode.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.44.3.4 virtual telux::common::ErrorCode telux::wlan::IWlanDeviceManager::getConfig (int & *numAp*, int & *numSta*) [pure virtual]

Request Wlan configuration: Returns the configuration that was set using telus::wlan::IWlanDeviceManager::setMode. This might differ from what configuration is has actually been enabled in the system, for instance, when the hardware cannot fully support the configuration that was set. To get the status of current configuration an Wlan enablement, telux::wlan::IWlanDeviceManager::getStatus should be used.

Parameters

in	*numAp*	Num of configured APs
in	*numSta*	Num of configured Stations

Returns

operation error code (if any). telux::common::ErrorCode

Note

Eval: This is a new API and is being evaluated.It is subject to change and could break backwards compatibility.

4.44.3.5 virtual telux::common::ErrorCode telux::wlan::IWlanDeviceManager::getStatus (bool & *isEnabled,* std::vector< InterfaceStatus > & *status*) [pure virtual]

Request Wlan status: Return Wlan enablement status and Interface status of APs and Station such as active/inactive, network interface name and hardware device they are mapped to. Results are valid only if Wlan is enabled.

Parameters

in	*isEnabled*	true: Wlan is enabled. false: Wlan is Disabled.
in	*status*	vector of interface status InterfaceStatus.

Returns

operation error code (if any). telux::common::ErrorCode

Note

Eval: This is a new API and is being evaluated.It is subject to change and could break backwards compatibility.

4.44.3.6 virtual telux::common::ErrorCode telux::wlan::IWlanDeviceManager::registerListener (std::weak_ptr< IWlanListener > *listener*) [pure virtual]

Register a listener for specific events in the Wlan Manager

Parameters

in	*listener*	pointer of IWlanListener object that processes the notification

Returns

Status of registerListener success or suitable status code

4.44.3.7 virtual telux::common::ErrorCode telux::wlan::IWlanDeviceManager::deregisterListener (std::weak_ptr< IWlanListener > *listener*) [pure virtual]

Removes a previously added listener.

Parameters

in	*listener*	pointer of IWlanListener object that needs to be removed

Returns

Status of deregisterListener success or suitable status code

4.44.3.8 virtual telux::wlan::IWlanDeviceManager::~IWlanDeviceManager ( ) [virtual]

Destructor for IWlanDeviceManager

4.44.3.9 virtual void telux::wlan::IWlanListener::onServiceStatusChange (telux::common::ServiceStatus *status*) [virtual]

This function is called when service status changes.

Parameters

in	*status*	- ServiceStatus

4.44.3.10 virtual void telux::wlan::IWlanListener::onEnableChanged (bool *enable*) [virtual]

This function is called when Wlan enablement has changed

Parameters

in	*enable*	True: Wlan is enabled, False: Wlan is disabled

4.44.3.11 virtual telux::wlan::IWlanListener::~IWlanListener ( ) [virtual]

4.44.4 Variable Documentation

4.44.4.1 HwDeviceType telux::wlan::InterfaceStatus::device

4.44.4.2 std::vector<ApStatus> telux::wlan::InterfaceStatus::apStatus

WiFi hardware type

Vector of active APs status

4.44.4.3 std::vector<StaStatus> telux::wlan::InterfaceStatus::staStatus

Vector of active Sta status

4.45 Access Point Management

This section contains APIs related to access point configuration management, such as private/guest, internet/local access, etc.

4.45.1 Define Documentation

4.45.1.1 #define INVALID_AP_ID 0

4.45.2 Data Structure Documentation

4.45.2.1 struct telux::wlan::ApNetConfig

Ap Network Configuration

Data fields

Type	Field	Description
ApInfo	info	AP type
ApInterworking	interworking	AP network access (internet/local)

4.45.2.2 struct telux::wlan::ApConfig

Ap Configuration

Data fields

Type	Field	Description
Id	id	AP id
vector< ApNetConfig >	network	Configurations supported by AP

4.45.2.3 struct telux::wlan::DeviceIndInfo

Wlan Client Device Indication Info

Data fields

Type	Field	Description
Id	id	AP id device is connected to
string	macAddress	MAC Address of Wi-Fi device

4.45.2.4 struct telux::wlan::DeviceInfo

Wlan Client Device Info

Data fields

Type	Field	Description
Id	id	AP id device is connected to
string	name	User friendly string that identifies Wi-Fi device
string	ipv4Address	IPv4 Address of Wi-Fi device
string	ipv6Address	IPv6 Address of Wi-Fi device
string	macAddress	MAC Address of Wi-Fi device

4.45.2.5 class telux::wlan::IApInterfaceManager

Manager class for configuring Wlan Access Points.

Public member functions

- virtual telux::common::ErrorCode setConfig (ApConfig config)=0
- virtual telux::common::ErrorCode getConfig (std::vector< ApConfig > &config)=0
- virtual telux::common::ErrorCode getStatus (std::vector< ApStatus > &status)=0
- virtual telux::common::ErrorCode getConnectedDevices (std::vector< DeviceInfo > &clientsInfo)=0
- virtual telux::common::ErrorCode manageApService (Id apId, ServiceOperation opr)=0
- virtual telux::common::ErrorCode registerListener (std::weak_ptr< IApListener > listener)=0
- virtual telux::common::ErrorCode deregisterListener (std::weak_ptr< IApListener > listener)=0
- virtual ~IApInterfaceManager ()

4.45.2.6 class telux::wlan::IApListener

Public member functions

- virtual void onApDeviceStatusChanged (ApDeviceConnectionEvent event, std::vector< DeviceIndInfo > info)
- virtual void onApBandChanged (BandType radio)
- virtual ~IApListener ()

4.45.3 Enumeration Type Documentation

4.45.3.1 enum telux::wlan::ApInterworking [strong]

AP Interworking Information

Enumerator

INTERNET_ACCESS AP with internet access only - No LAN access
FULL_ACCESS AP Can Access LAN and Internet

4.45.3.2 enum telux::wlan::ApDeviceConnectionEvent [strong]

AP Client Connection Status

Enumerator

CONNECTED
DISCONNECTED
IPV4_UPDATED
IPV6_UPDATED

4.45.4 Function Documentation

4.45.4.1 virtual telux::common::ErrorCode telux::wlan::IApInterfaceManager::setConfig (ApConfig *config*) [pure virtual]

Set Access Point config: Used to fully configure access points including venue type, radio type (2.4/5 GHz), private/guest network and all other related settings. Configurations will take effect after hostapd service is restarted by calling telux::wlan::IApInterfaceManager::manageApService.

On platforms with Access control enabled, Caller needs to have TELUX_WLAN_AP_CONFIG permission to invoke this API successfully.

Parameters

in	*config*	AP configuration parameters telux::wlan::ApConfig

Returns

operation error code (if any). telux::common::ErrorCode telux::common::Status::NOTALLOWED is returned if AP to be configured was not enabled in telux::wlan::WlanDeviceManager::setMode.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.45.4.2 virtual telux::common::ErrorCode telux::wlan::IApInterfaceManager::getConfig (std::vector< ApConfig > & *config*) [pure virtual]

Request Access Point Configurations

Parameters

in	*config*	Vector of AP configurations telux::wlan::ApConfig as set by telux::wlan::IApInterfaceManager::setConfig

Returns

operation error code (if any). telux::common::ErrorCode

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.45.4.3 virtual telux::common::ErrorCode telux::wlan::IApInterfaceManager::getStatus (std::vector< ApStatus > & *status*) [pure virtual]

Request AP Status

Parameters

in	*status*	Vector of AP network Status telux::wlan::ApStatus

Returns

operation error code (if any). telux::common::ErrorCode

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.45.4.4 virtual telux::common::ErrorCode telux::wlan::IApInterfaceManager::getConnectedDevices (std::vector< DeviceInfo > & *clientsInfo*) [pure virtual]

Request Connected Devices to all enabled access points. Each entry in returned list will contain information about a device such as access point it is connected to and IP and MAC address as defined in telux::wlan::DeviceInfo

On platforms with Access control enabled, Caller needs to have TELUX_WLAN_AP_DEVICES permission to invoke this API successfully.

Parameters

in	*clientsInfo*	List of connected devices Info telux::wlan::DeviceInfo

Returns

operation error code (if any). telux::common::ErrorCode

Note

Eval: This is a new API and is being evaluated.It is subject to change and could break backwards compatibility.

4.45.4.5 virtual telux::common::ErrorCode telux::wlan::IApInterfaceManager↩ ::manageApService (Id *apId,* ServiceOperation *opr*) [pure virtual]

Execute an operation on hostapd service. Provides ability for client to either stop/start or restart hostapd service for selected access point. Restarting hostapd service is required for any changes made to hosapd.conf file and changes made by telux::wlan::IApInterfaceManager::setConfig to take effect. Stop/Start operation telux::wlan::ServiceOperation will Stop/Start WiFi service for access point. Access points selected to execute operation on, will temporarily go out of service when this API is called. This API should be called only when access point is configured through

On platforms with Access control enabled, Caller needs to have TELUX_WLAN_AP_CONFIG permission to invoke this API successfully.

telux::wlan::IDeviceManager::setMode

Parameters

in	*apId*	AP identifier to execute operation on. telux::wlan::Id
in	*opr*	Operation to be performed on hostapd telux::wlan::ServiceOperation

Returns

operation error code (if any). telux::common::ErrorCode

Note

Eval: This is a new API and is being evaluated.It is subject to change and could break backwards compatibility.

4.45.4.6 virtual telux::common::ErrorCode telux::wlan::IApInterfaceManager↩ ::registerListener (std::weak_ptr< IApListener > *listener*) [pure virtual]

Register a listener for specific events in Access Point Manager

Parameters

in	*listener*	pointer of IApListener object that processes the notification

Returns

operation error code (if any). telux::common::ErrorCode

4.45.4.7 virtual telux::common::ErrorCode telux::wlan::IApInterfaceManager↩ ::deregisterListener (std::weak_ptr< IApListener > *listener*) [pure virtual]

Removes a previously added listener.

Parameters

in	*listener*	pointer of IApListener object that needs to be removed

Returns

operation error code (if any). telux::common::ErrorCode

4.45.4.8 virtual telux::wlan::IApInterfaceManager::~IApInterfaceManager ( ) [virtual]

4.45.4.9 virtual void telux::wlan::IApListener::onApDeviceStatusChanged (ApDeviceConnectionEvent *event*, std::vector< DeviceIndInfo > *info*) [virtual]

This function is called when AP device status has changed

Parameters

in	*event*	Event detected on device telux::wlan::ApDeviceConnectionEvent
in	*info*	Info about devices telux::wlan::DeviceIndInfo

4.45.4.10 virtual void telux::wlan::IApListener::onApBandChanged (BandType *radio*) [virtual]

This function is called when AP switch to different operation band

Parameters

in	*radio*	New AP operation band telux::wlan::BandType

4.45.4.11 virtual telux::wlan::IApListener::~IApListener ( ) [virtual]

4.45.5 Variable Documentation

4.45.5.1 ApInfo telux::wlan::ApNetConfig::info

AP type

4.45.5.2 ApInterworking telux::wlan::ApNetConfig::interworking

AP network access (internet/local)

4.45.5.3 Id telux::wlan::ApConfig::id

AP id

4.45.5.4 std::vector<ApNetConfig> telux::wlan::ApConfig::network

Configurations supported by AP

4.45.5.5 Id telux::wlan::DeviceIndInfo::id

AP id device is connected to

4.45.5.6 std::string telux::wlan::DeviceIndInfo::macAddress

MAC Address of Wi-Fi device

4.45.5.7 Id telux::wlan::DeviceInfo::id

AP id device is connected to

4.45.5.8 std::string telux::wlan::DeviceInfo::name

User friendly string that identifies Wi-Fi device

4.45.5.9 std::string telux::wlan::DeviceInfo::ipv4Address

IPv4 Address of Wi-Fi device

4.45.5.10 std::string telux::wlan::DeviceInfo::ipv6Address

IPv6 Address of Wi-Fi device

4.45.5.11 std::string telux::wlan::DeviceInfo::macAddress

MAC Address of Wi-Fi device

4.46 Station Management

This section contains APIs related to station configuration management, such as static/dynamic IP, bridge/router mode, etc.

4.46.1 Data Structure Documentation

4.46.1.1 struct telux::wlan::StaStaticIpConfig

Static IP Configuration

Data fields

Type	Field	Description
string	ipAddr	IPv4 address to be assigned.
string	gwIpAddr	IPv4 address of the gateway.
string	netMask	Subnet mask.
string	dnsAddr	DNS IPv4 address.

4.46.1.2 struct telux::wlan::StaConfig

Station Configuration

Data fields

Type	Field	Description
Id	staId	Id of station backhaul
StaIpConfig	ipConfig	IP configuration of station backhaul
StaStaticIp↩ Config	staticIpConfig	Static IP configuration if selected
StaBridgeMode	bridgeMode	Station configuration as Router/bridge

4.46.1.3 class telux::wlan::IStaInterfaceManager

Manager class for configuring Wlan Station Mode.

Public member functions

- virtual telux::common::ErrorCode setIpConfig (Id staId, StaIpConfig ipConfig, StaStaticIpConfig staticIpConfig)=0
- virtual telux::common::ErrorCode setBridgeMode (Id staId, StaBridgeMode bridgeMode)=0
- virtual telux::common::ErrorCode getConfig (std::vector< StaConfig > &config)=0
- virtual telux::common::ErrorCode getStatus (std::vector< StaStatus > &status)=0
- virtual telux::common::ErrorCode manageStaService (Id staId, ServiceOperation opr)=0
- virtual telux::common::ErrorCode registerListener (std::weak_ptr< IStaListener > listener)=0
- virtual telux::common::ErrorCode deregisterListener (std::weak_ptr< IStaListener > listener)=0

- virtual ~IStaInterfaceManager ()

4.46.1.4 class telux::wlan::IStaListener

Public member functions

- virtual void onStationStatusChanged (std::vector< StaStatus > staStatus)
- virtual void onStationBandChanged (BandType radio)
- virtual ~IStaListener ()

4.46.2 Enumeration Type Documentation

4.46.2.1 enum telux::wlan::StaIpConfig [strong]

Station Connection IP Type

Enumerator

DYNAMIC_IP Station is configured with dynamic IP
STATIC_IP Station is configured with Static IP

4.46.2.2 enum telux::wlan::StaBridgeMode [strong]

Bridge/Router Mode

Enumerator

ROUTER Station is in Router Mode
BRIDGE Station is in Bridge Mode

4.46.3 Function Documentation

4.46.3.1 virtual telux::common::ErrorCode telux::wlan::IStaInterfaceManager::setIp↩ Config (Id *staId,* StaIpConfig *ipConfig,* StaStaticIpConfig *staticIpConfig*) [pure virtual]

Set Station IP Configurations: Set Station IP configuration dynamic/static and static IP address if selected. If API is called when WLAN is disabled, changes will take effect when WLAN is enabled using telux::wlan::IWlanDeviceManager::enable API. If API is called when WLAN is enabled, changes will take effect after restarting wpa_supplicant by calling telux::wlan::IStaInterfaceManager::manageStaService

Parameters

in	*staId*	Station Identifier telux::wlan::Id
in	*ipConfig*	Static/Dynamic IP configuration telux::wlan::StaIpConfig.
in	*staticIpConfig*	Static IP configuration, not used if station was configured to use dynamic IP.

On platforms with Access control enabled, Caller needs to have TELUX_WLAN_STA_CONFIG

permission to invoke this API successfully.

Returns

operation error code (if any). telux::common::ErrorCode.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.46.3.2 virtual telux::common::ErrorCode telux::wlan::IStaInterfaceManager::setBridgeMode (Id *staId,* StaBridgeMode *bridgeMode*) [pure virtual]

Set Station backhaul to act as router or bridge: Sets Station to act as router or bridge where station internal clients get public IP addresses. If API is called when WLAN is disabled, changes will take effect when WLAN is enabled using telux::wlan::IWlanDeviceManager::enable API. If API is called when WLAN is enabled, changes will take effect after restarting wpa_supplicant by calling telux::wlan::IStaInterfaceManager::manageStaService

On platforms with Access control enabled, Caller needs to have TELUX_WLAN_STA_CONFIG permission to invoke this API successfully.

Parameters

in	*staId*	Station Identifier telux::wlan::Id
in	*bridgeMode*	bridgeMode telux::wlan::StaBridgeMode

Returns

operation error code (if any). telux::common::ErrorCode.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.46.3.3 virtual telux::common::ErrorCode telux::wlan::IStaInterfaceManager::getConfig (std::vector< StaConfig > & *config*) [pure virtual]

Request current station configurations: Returns configurations set by telux::wlan::IStaInterfaceManager::setIpConfig and telux::wlan::IStaInterfaceManager::setBridgeMode

Parameters

in	*config*	Station configurations telux::wlan::StaConfig

Returns

operation error code (if any). telux::common::ErrorCode.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.46.3.4 virtual telux::common::ErrorCode telux::wlan::IStaInterfaceManager::getStatus (std::vector< StaStatus > & *status*) [pure virtual]

Request current station status: Returns current Sta interface status such as network interface name and IP address.

Parameters

in	*status*	Station Status telux::wlan::StaStatus

Returns

operation error code (if any). telux::common::ErrorCode.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.46.3.5 virtual telux::common::ErrorCode telux::wlan::IStaInterfaceManager::manageStaService (Id *staId,* ServiceOperation *opr*) [pure virtual]

Execute an operation on wpa_supplicant service. Provides ability for client to either stop/start or restart wpa_supplicant service for selected station. Restarting wpa_supplicant service is required for any changes made to wpa_supplicant.conf file to take effect. Station selected to execute operation on, will temporarily go out of service when this API is called. This API should be called only when station mode is configured through telux::wlan::IDeviceManager::setMode

On platforms with Access control enabled, Caller needs to have TELUX_WLAN_STA_CONFIG permission to invoke this API successfully.

Parameters

in	*staId*	Station identifier to execute operation on. telux::wlan::Id
in	*opr*	Operation to be performed on wpa_supplicant telux::wlan::ServiceOperation

Returns

operation error code (if any). telux::common::ErrorCode.

Note

Eval: This is a new API and is being evaluated.It is subject to change and could break backwards compatibility.

4.46.3.6 virtual telux::common::ErrorCode telux::wlan::IStaInterfaceManager↩ ::registerListener (std::weak_ptr< IStaListener > *listener*) [pure virtual]

Register as a listener for specific events defined in telux::wlan::IStaListener

Parameters

in	*listener*	pointer of IStaListener object that processes the notification

Returns

operation error code (if any). telux::common::ErrorCode.

4.46.3.7 virtual telux::common::ErrorCode telux::wlan::IStaInterfaceManager↩ ::deregisterListener (std::weak_ptr< IStaListener > *listener*) [pure virtual]

Removes a previously added listener.

Parameters

in	*listener*	pointer of IStaListener object that needs to be removed

Returns

operation error code (if any). telux::common::ErrorCode.

4.46.3.8 virtual telux::wlan::IStaInterfaceManager::~IStaInterfaceManager ( ) [virtual]

4.46.3.9 virtual void telux::wlan::IStaListener::onStationStatusChanged (std↩ ::vector< StaStatus > *staStatus*) [virtual]

This function is called when Station Status Changes

Parameters

in	*status*	List of station state telux::wlan::StaStatus

4.46.3.10 virtual void telux::wlan::IStaListener::onStationBandChanged (BandType *radio*) [virtual]

This function is called when Station switch to different operation band

Parameters

in	*radio*	New Station operation band telux::wlan::BandType

4.46.3.11 virtual telux::wlan::IStaListener::~IStaListener ( ) [virtual]

4.46.4 Variable Documentation

4.46.4.1 std::string telux::wlan::StaStaticIpConfig::ipAddr

IPv4 address to be assigned.

4.46.4.2 std::string telux::wlan::StaStaticIpConfig::gwIpAddr

IPv4 address of the gateway.

4.46.4.3 std::string telux::wlan::StaStaticIpConfig::netMask

Subnet mask.

4.46.4.4 std::string telux::wlan::StaStaticIpConfig::dnsAddr

DNS IPv4 address.

4.46.4.5 Id telux::wlan::StaConfig::staId

Id of station backhaul

4.46.4.6 StaIpConfig telux::wlan::StaConfig::ipConfig

IP configuration of station backhaul

4.46.4.7 StaStaticIpConfig telux::wlan::StaConfig::staticIpConfig

Static IP configuration if selected

4.46.4.8 StaBridgeMode telux::wlan::StaConfig::bridgeMode

Station configuration as Router/bridge

4.47 Cellular Data

- net

This section contains APIs related to Cellular Data Services.

4.47.1 Define Documentation

4.47.1.1 #define PROFILE_ID_MAX 0x7FFFFFFF

Default data profile id.

4.47.1.2 #define MAX_QOS_FILTERS 16

Max filters in one flow

4.47.1.3 #define IP_PROT_UNKNOWN 0xFF

Default IP Protocol number in IPv4 or IPv6 headers.

4.47.2 Data Structure Documentation

4.47.2.1 struct telux::data::IpFamilyInfo

IP Family related Info

Data fields

Type	Field	Description
DataCallStatus	status	
IpAddrInfo	addr	

4.47.2.2 struct telux::data::QosFilterRule

Encapsulate the Qos Filter rule

Data fields

Type	Field	Description
vector< shared_ptr< IIpFilter > >	filter	IIpFilter
uint16_t	filterId	Unique identifier for each filter.
uint16_t	filterPrecedence	Specifies the order in which filters are applied. A lower numerical value has a higher precedence.

4.47.2.3 struct telux::data::TrafficFlowTemplate

QOS TFT Flow info

Data fields

Type	Field	Description
QosFlowId	qosId	Mandatory defines current flow id
QosFlow↩ StateChange↩ Event	stateChange	Flow state change event
QosFlowMask	mask	bitmask to denote which of the optional fields in TrafficFlowTemplate are valid
QosIPFlowInfo	txGrantedFlow	Optional
QosIPFlowInfo	rxGrantedFlow	
uint32_t	txFiltersLength	
QosFilterRule	txFilters[MA↩ X_QOS_FIL↩ TERS]	
uint32_t	rxFiltersLength	
QosFilterRule	rxFilters[MA↩ X_QOS_FIL↩ TERS]	

4.47.2.4 struct telux::data::TftChangeInfo

QOS TFT flow change info

Data fields

Type	Field	Description
shared_ptr< TrafficFlow↩ Template >	tft	TFT flow info TrafficFlowTemplate
QosFlow↩ StateChange↩ Event	stateChange	Flow state change event

4.47.2.5 struct telux::data::BitRateInfo

Data call bit rate info

Data fields

Type	Field	Description
uint64_t	txRate	**Deprecated** Unused
uint64_t	rxRate	**Deprecated** Unused
uint64_t	maxTxRate	Maximum transmit rate that can be assigned to device in bits/sec
uint64_t	maxRxRate	Maximum receive rate that can be assigned to device in bits/sec

4.47.2.6 class telux::data::IDataConnectionManager

IDataConnectionManager is a primary interface for cellular connectivity This interface provides APIs for start and stop data call connections, get data call information and listener for monitoring data calls. It also provides interface to Subsystem Restart events by registering as listener. Notifications will be received when modem is ready/not ready.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual bool isSubsystemReady ()=0
- virtual std::future< bool > onSubsystemReady ()=0
- virtual telux::common::Status setDefaultProfile (OperationType operationType, uint8_t profileId, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status getDefaultProfile (OperationType operationType, DefaultProfileIdResponseCb callback)=0
- virtual telux::common::Status setRoamingMode (bool enable, uint8_t profileId, OperationType operationType, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestRoamingMode (uint8_t profileId, OperationType operationType, requestRoamingModeResponseCb callback)=0
- virtual telux::common::Status startDataCall (int profileId, IpFamilyType ipFamilyType=IpFamilyType::IPV4V6, DataCallResponseCb callback=nullptr, OperationType operationType=OperationType::DATA_LOCAL, std::string apn="")=0
- virtual telux::common::Status stopDataCall (int profileId, IpFamilyType ipFamilyType=IpFamilyType::IPV4V6, DataCallResponseCb callback=nullptr, OperationType operationType=OperationType::DATA_LOCAL, std::string apn="")=0
- virtual telux::common::Status registerListener (std::weak_ptr< IDataConnectionListener > listener)=0

- virtual telux::common::Status deregisterListener (std::weak_ptr< IDataConnectionListener > listener)=0
- virtual int getSlotId ()=0
- virtual telux::common::Status requestDataCallList (OperationType operationType, DataCallListResponseCb callback)=0
- virtual ~IDataConnectionManager ()

4.47.2.6.1 Constructors and Destructors

4.47.2.6.1.1 virtual telux::data::IDataConnectionManager::~IDataConnectionManager ( ) [virtual]

Destructor for IDataConnectionManager

4.47.2.6.2 Member Function Documentation

4.47.2.6.2.1 virtual telux::common::ServiceStatus telux::data::IDataConnectionManager::getServiceStatus ( ) [pure virtual]

Returns current initialization status of data connection manager.

Returns

SERVICE_AVAILABLE If data connection manager is ready for service.
SERVICE_UNAVAILABLE If data connection manager is temporarily unavailable.
SERVICE_FAILED If data connection manager encountered an irrecoverable failure.

4.47.2.6.2.2 virtual bool telux::data::IDataConnectionManager::isSubsystemReady ( ) [pure virtual]

Checks if the data subsystem is ready.

Returns

True if Data Connection Manager is ready for service, otherwise returns false.

Deprecated

Use getServiceStatus API.

4.47.2.6.2.3 virtual std::future<bool> telux::data::IDataConnectionManager::onSubsystemReady ( ) [pure virtual]

Wait for data subsystem to be ready.

Returns

A future that caller can wait on to be notified when card manager is ready.

Deprecated

Use InitResponseCb callback in factory API getDataConnectionManager.

4.47.2.6.2.4 virtual telux::common::Status telux::data::IDataConnectionManager::setDefaultProfile (OperationType *operationType,* uint8_t *profileId,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

Set a profile as default which results in following: Traffic from devices tethered to MDM via default network interfaces such as eth0, ecm0, and mhi0 will be directed to rmnet_data that was brought up with default profile. Traffic initiated within MDM or EAP that is destined for WAN network and is not bound to a WAN interface will be routed by default to WAN network corresponding to the default profile ID

On platforms with Access control enabled, Caller needs to have TELUX_DATA_SETTING permission to invoke this API successfully.

Parameters

in	*operationType*	telux::data::OperationType
in	*profileId*	Profile identifier to be set as default
in	*callback*	optional callback to get the response setDefaultProfile

Returns

Immediate status of setDefaultProfile i.e. success or suitable status.

4.47.2.6.2.5 virtual telux::common::Status telux::data::IDataConnectionManager::getDefaultProfile (OperationType *operationType,* DefaultProfileIdResponseCb *callback*) [pure virtual]

Get current default profile

Parameters

in	*operationType*	telux::data::OperationType
in	*callback*	callback to get the response getDefaultProfile

Returns

Immediate status of getDefaultProfile i.e. success or suitable status.

4.47.2.6.2.6 virtual telux::common::Status telux::data::IDataConnectionManager::setRoamingMode (bool *enable,* uint8_t *profileId,* OperationType *operationType,* telux::common::Response↩ Callback *callback = nullptr*) [pure virtual]

Enable roaming mode for profile id. If disabled, any client attempt to bring up data call on such profile id will be prevented by system when device is in roaming area. System will report NO_NETWORK_FOUND error in such scenario. if enabled, clients can bring up data call made on such profile id and slot id successfully even if device is in roaming area. Configuration changes will be persistant across multiple boots.

On platforms with Access control enabled, Caller needs to have TELUX_DATA_SETTING permission to invoke this API successfully.

Parameters

in	*enable*	enable/disable roaming mode (True: enable, False:disable).
in	*profileId*	profile id on which roaming mode to be enabled/disabled.
in	*operationType*	telux::data::OperationType
in	*callback*	optional. Callback to get response for enableRoamingMode.

Returns

Status of enableRoamingMode i.e. success or suitable status code. NO_NETWORK_FOUND error is returned if roaming is not enabled on profile id.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.47.2.6.2.7 virtual telux::common::Status telux::data::IDataConnectionManager::requestRoamingMode (uint8_t *profileId*, OperationType *operationType*, requestRoamingModeResponseCb *callback*) [pure virtual]

Request current roaming mode for profile id.

Parameters

in	*profileId*	profile id on which roaming mode is requested.
in	*operationType*	telux::data::OperationType
in	*callback*	callback to get response for requestRoamingMode.

Returns

Status of requestRoamingMode i.e. success or suitable status code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.47.2.6.2.8 virtual telux::common::Status telux::data::IDataConnectionManager::startDataCall (int *profileId*, IpFamilyType *ipFamilyType* = IpFamilyType::IPV4V6, DataCallResponseCb *callback* = `nullptr`, OperationType *operationType* = OperationType::DATA_LOCAL, std::string *apn* = `""`) [pure virtual]

Starts a data call corresponding to default or specified profile identifier.

This will bring up data call connection based on specified profile identifier, IP family type, and operation

type (local/remote). This is an asynchronous API. If telux::common::Status::SUCCESS is returned, client provided callback will be invoked at later time with error code and DataCall object associated with requested call. Clients might receive additional notification for the final data call status. For details see telux::data::DataCallResponseCb.

On platforms with Access control enabled, Caller needs to have TELUX_DATA_CALL_OPS permission to invoke this API successfully.

Note

if application starts data call on IPV4V6 then it's expected to stop the data call on same ip family type (i.e IPV4V6).

Parameters

in	*profileId*	Profile identifier corresponding to which data call bring up will be done. Use IDataProfileManager::requestProfileList to get list of available profiles.
in	*ipFamilyType*	Identifies IP family type
out	*callback*	Optional callback to get the response of start data call.
in	*operationType*	Optional telux::data::OperationType
in	*apn*	Deprecated and currently unused

Returns

Immediate status of startDataCall() request sent i.e. success or suitable status code.

4.47.2.6.2.9 virtual telux::common::Status telux::data::IDataConnectionManager::stopDataCall (int *profileId*, IpFamilyType *ipFamilyType* = IpFamilyType::IPV4V6, DataCallResponseCb *callback* = `nullptr`, OperationType *operationType* = OperationType::DATA_LOCAL, std::string *apn* = `""`) `[pure virtual]`

Tear down data call connection based on specified profile identifier, IP family type, and operation type (local/remote). This is an asynchronous API. If telux::common::Status::SUCCESS is returned, client provided callback will be invoked at later time with error code and DataCall object associated with requested call. Clients might receive additional notification for the final data call status. For details see telux::data::DataCallResponseCb.

This will tear down specific data call connection based on profile identifier.

On platforms with Access control enabled, Caller needs to have TELUX_DATA_CALL_OPS permission to invoke this API successfully.

Note

If application starts data call on IPV4V6 then it's expected to stop the data call on same ip family type (i.e IPV4V6). Client can only stop data call it started.

Parameters

in	*profileId*	Profile identifier corresponding to which data call tear down will be done. Use data profile manager to get the list of available profiles.

in	*ipFamilyType*	Identifies IP family type
out	*callback*	Optional callback to get the response of stop data call
in	*operationType*	Optional telux::data::OperationType
in	*apn*	Deprecated and currently unused

Returns

Immediate status of stopDataCall() request sent i.e. success or suitable status code. The client receives asynchronous notifications indicating the data call tear-down.

4.47.2.6.2.10 virtual telux::common::Status telux::data::IDataConnectionManager::registerListener (std::weak_ptr< IDataConnectionListener > *listener*) [pure virtual]

Register a listener for specific events in the Connection Manager like establishment of new data call, data call info change and call failure.

Parameters

in	*listener*	pointer of IDataConnectionListener object that processes the notification

Returns

Status of registerListener success or suitable status code

4.47.2.6.2.11 virtual telux::common::Status telux::data::IDataConnectionManager::deregisterListener (std::weak_ptr< IDataConnectionListener > *listener*) [pure virtual]

Removes a previously added listener.

Parameters

in	*listener*	pointer of IDataConnectionListener object that needs to be removed

Returns

Status of deregisterListener success or suitable status code

4.47.2.6.2.12 virtual int telux::data::IDataConnectionManager::getSlotId ( ) [pure virtual]

Get associated slot id for the Data Connection Manager.

Returns

SlotId

4.47.2.6.2.13 virtual telux::common::Status telux::data::IDataConnectionManager::requestDataCallList (OperationType *operationType*, DataCallListResponseCb *callback*) [pure virtual]

Request list of data calls available in the system

Parameters

out	*operationType*	telux::data::OperationType
out	*callback*	Callback with list of supported data calls

4.47.2.7 class telux::data::IDataCall

Represents single established data call on the device.

Public member functions

- virtual const std::string & getInterfaceName ()=0
- virtual DataCallEndReason getDataCallEndReason ()=0
- virtual DataCallStatus getDataCallStatus ()=0
- virtual IpFamilyInfo getIpv4Info ()=0
- virtual IpFamilyInfo getIpv6Info ()=0
- virtual TechPreference getTechPreference ()=0
- virtual std::list< IpAddrInfo > getIpAddressInfo ()=0
- virtual IpFamilyType getIpFamilyType ()=0
- virtual int getProfileId ()=0
- virtual SlotId getSlotId ()=0
- virtual OperationType getOperationType ()=0
- virtual telux::common::Status requestTrafficFlowTemplate (IpFamilyType ipFamilyType, TrafficFlowTemplateCb callback)=0
- virtual telux::common::Status requestDataCallStatistics (StatisticsResponseCb callback=nullptr)=0
- virtual telux::common::Status resetDataCallStatistics (telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestDataCallBitRate (requestDataCallBitRateResponseCb callback)=0
- virtual ~IDataCall ()
- virtual DataBearerTechnology getCurrentBearerTech ()=0

4.47.2.7.1 Constructors and Destructors

4.47.2.7.1.1 virtual telux::data::IDataCall::~IDataCall ( ) [virtual]

Destructor for IDataCall

4.47.2.7.2 Member Function Documentation

4.47.2.7.2.1 virtual const std::string& telux::data::IDataCall::getInterfaceName ( ) [pure virtual]

Get interface name associated with the data call.

Returns

Interface Name.

4.47.2.7.2.2 virtual DataCallEndReason telux::data::IDataCall::getDataCallEndReason ( ) [pure virtual]

Get failure reason for the data call.

Returns

DataCallEndReason

4.47.2.7.2.3 virtual DataCallStatus telux::data::IDataCall::getDataCallStatus ( ) [pure virtual]

Get data call status like connected, disconnected and IP address changes.

Returns

DataCallStatus.

4.47.2.7.2.4 virtual IpFamilyInfo telux::data::IDataCall::getIpv4Info ( ) [pure virtual]

Get IPv4 Family info like connected, disconnected and IP address changes.

Returns

IpFamilyInfo.

4.47.2.7.2.5 virtual IpFamilyInfo telux::data::IDataCall::getIpv6Info ( ) [pure virtual]

Get IPv6 Family info like connected, disconnected and IP address changes.

Returns

IpFamilyInfo.

4.47.2.7.2.6 virtual TechPreference telux::data::IDataCall::getTechPreference ( ) [pure virtual]

Get the technology on which the call was brought up.

Returns

TechPreference.

4.47.2.7.2.7 virtual std::list<IpAddrInfo> telux::data::IDataCall::getIpAddressInfo ( ) [pure virtual]

Get list of IP address information.

Returns

List of IP address details.

4.47.2.7.2.8 virtual IpFamilyType telux::data::IDataCall::getIpFamilyType ( ) [pure virtual]

Get IP Family Type i.e. IPv4, IPv6 or Both

Returns

IpFamilyType.

4.47.2.7.2.9 virtual int telux::data::IDataCall::getProfileId ( ) [pure virtual]

Get Profile Id

Returns

Profile Identifier.

4.47.2.7.2.10 virtual SlotId telux::data::IDataCall::getSlotId ( ) [pure virtual]

Get Slot Id

Returns

Subscription Slot Identifier.

4.47.2.7.2.11 virtual OperationType telux::data::IDataCall::getOperationType ( ) [pure virtual]

Get data operation used for the DataCall.

Returns

OperationType

4.47.2.7.2.12 virtual telux::common::Status telux::data::IDataCall::requestTrafficFlowTemplate (IpFamilyType *ipFamilyType,* TrafficFlowTemplateCb *callback*) [pure virtual]

Get the current installed QOS Traffic flow template information.

Parameters

in	*ipFamilyType*	- IP Family type IpFamilyType. TFT's are installed per IP Family.
in	*callback*	- callback function to get the result of API.

Returns

Status of requestTrafficFlowTemplate i.e. success or suitable status code.

4.47.2.7.2.13 virtual telux::common::Status telux::data::IDataCall::requestDataCallStatistics (StatisticsResponseCb *callback = nullptr*) [pure virtual]

Request the data transfer statistics for data call corresponding to specified profile identifier.

Parameters

in	*callback*	Optional callback to get the response of request Data Call Statistics

Returns

Status of getDataCallStatistics i.e. success or suitable status code.

4.47.2.7.2.14 virtual telux::common::Status telux::data::IDataCall::resetDataCallStatistics (telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

Reset data transfer statistics for data call corresponding to specified profile identifier.

On platforms with Access control enabled, Caller needs to have TELUX_DATA_CALL_PROPS permission to invoke this API successfully.

Parameters

in	*callback*	optional callback to get the response of reset Data call statistics

Returns

Status of resetDataCallStatistics i.e. success or suitable status code.

4.47.2.7.2.15 virtual telux::common::Status telux::data::IDataCall::requestDataCallBitRate (requestDataCallBitRateResponseCb *callback*) [pure virtual]

Request data call bit rate in (bits/sec).

Parameters

out	*callback*	callback to be called with bit rate results requestDataCallBitRateResponseCb

Returns

Status of requestDataCallBitRate success or suitable status code

4.47.2.7.2.16 virtual DataBearerTechnology telux::data::IDataCall::getCurrentBearerTech ( ) [pure virtual]

Get the bearer technology on which earlier data call was brought up like LTE, WCDMA and etc. This is synchronous API called by client to get bearer technology corresponding to data call.

Returns

DataBearerTechnology

Deprecated

, use telux::data::IServingSystemManager::requestServiceStatus instead

4.47.2.8 class telux::data::IDataConnectionListener

Interface for Data connection listener object. Client needs to implement this interface to get access to data services notifications like onNewDataCall, onDataCallStatusChanged and onDataCallFailure.

The methods in listener can be invoked from multiple different threads. The implementation should be thread safe.

The notification delivery mechanism uses the same thread to deliver all the queued notifications to ensure they are delivered in order. Considering this, the thread on which the notifications are delivered should not be blocked for longer operations since this would result in delay in delivery of further notifications that are in the queue waiting to be dispatched.

Public member functions

- virtual void onDataCallInfoChanged (const std::shared_ptr< IDataCall > &dataCall)
- virtual void onHwAccelerationChanged (const ServiceState state)
- virtual void onTrafficFlowTemplateChange (const std::shared_ptr< IDataCall > &dataCall, const std::vector< std::shared_ptr< TftChangeInfo >> &tft)
- virtual void onWwanConnectivityConfigChange (SlotId slotId, bool isConnectivityAllowed)

- virtual ~IDataConnectionListener ()

4.47.2.8.1 Constructors and Destructors

4.47.2.8.1.1 virtual telux::data::IDataConnectionListener::~IDataConnectionListener ( ) [virtual]

Destructor for IDataConnectionListener

4.47.2.8.2 Member Function Documentation

4.47.2.8.2.1 virtual void telux::data::IDataConnectionListener::onDataCallInfoChanged (const std::shared_ptr< IDataCall > & *dataCall*) [virtual]

This function is called when there is a change in the data call.

Parameters

in	*dataCall*	Pointer to IDataCall

4.47.2.8.2.2 virtual void telux::data::IDataConnectionListener::onHwAccelerationChanged (const ServiceState *state*) [virtual]

This function is called when a change occur in hardware acceleration service. If reported state is ServiceState::INACTIVE: All existing data calls will take software acceleration path. If reported state is ServiceState::ACTIVE: All new data calls that are started after this API invocation will be H/w accelerated. Data calls that existed before this API was invoked will continue without h/w acceleration. Client could stop and re-start pre-existing data calls in order to use H/w acceleration.

Parameters

in	*state*	New state of hardware Acceleration service (Active/Inactive)

Note

This is global state

4.47.2.8.2.3 virtual void telux::data::IDataConnectionListener::onTrafficFlowTemplateChange (const std::shared_ptr< IDataCall > & *dataCall*, const std::vector< std::shared_ptr< TftChangeInfo >> & *tft*) [virtual]

This function is called when the TFT's parameters are changed for a packet data session.

Parameters

in	*dataCall*	Pointer to IDataCall
in	*tft*	vector of TftChangeInfo TftChangeInfo

4.47.2.8.2.4 virtual void telux::data::IDataConnectionListener::onWwanConnectivityConfigChange (SlotId *slotId,* bool *isConnectivityAllowed*) [virtual]

This function is called when WWAN backhaul connectivity config changes.

Parameters

in	*slotId*	- Slot Id for which connectivity has changed.
in	*isConnectivity↩ Allowed*	- Connectivity status allowed/disallowed.

4.47.2.9 struct telux::data::DataRestrictMode

Defines the supported powersave filtering mode and autoexit for the packet data session.
DataRestrictModeType

Data fields

Type	Field	Description
DataRestrict↩ ModeType	filterMode	Disable or enable data filter mode. When disabled all the data packets will be forwarded from modem to the apps. When enabled only the data matching the filters will be forwarded from modem to the apps.
DataRestrict↩ ModeType	filterAutoExit	Disable or enable autoexit feature. When enabled, once an incoming packet matching the filter is received, filter mode will we disable automatically and any packet will be allowed to be forwarded from modem to apps.

4.47.2.10 struct telux::data::PortInfo

Used to define the Port number and range (number of ports following port value) Ex- for ports ranging from 1000-3000 port = 1000 and range= 2000

for single port 5000 port = 5000 and range= 0

Data fields

Type	Field	Description
uint16_t	port	Port.
uint16_t	range	Range.

4.47.2.11 struct telux::data::ProfileParams

Profile Parameters used for profile creation, query and modify

Data fields

Type	Field	Description
string	profileName	Profile Name

Type	Field	Description
string	apn	APN name
string	userName	APN user name (if any)
string	password	APN password (if any)
TechPreference	techPref	Technology preference, default is TechPreference::UNKNOWN
AuthProtocol↩ Type	authType	Authentication protocol type, default is AuthProtocolType::AUTH_NONE
IpFamilyType	ipFamilyType	Preferred IP family for the call, default is IpFamilyType::UNKNOWN
ApnTypes	apnTypes	APN Types ApnMaskType

4.47.2.12 struct telux::data::DataCallStats

Data transfer statistics structure.

Data fields

Type	Field	Description
uint64_t	packetsTx	Number of packets transmitted
uint64_t	packetsRx	Number of packets received
uint64_t	bytesTx	Number of bytes transmitted
uint64_t	bytesRx	Number of bytes received
uint64_t	packets↩ DroppedTx	Number of transmit packets dropped
uint64_t	packets↩ DroppedRx	Number of receive packets dropped

4.47.2.13 struct telux::data::IpAddrInfo

IP address information structure

Data fields

Type	Field	Description
string	ifAddress	Interface IP address.
unsigned int	ifMask	Subnet mask.
string	gwAddress	Gateway IP address.
unsigned int	gwMask	Subnet mask.
string	primaryDns↩ Address	Primary DNS address.
string	secondary↩ DnsAddress	Secondary DNS address.

4.47.2.14 struct telux::data::DataCallEndReason

Structure represents data call failure reason type and code.

Data fields

Type	Field	Description
EndReason↩ Type	type	Data call terminated due to reason type, default is CE_UNKNOWN
union Data↩ CallEndReason	__unnamed_↩ _	

4.47.2.15 struct telux::data::VlanConfig

Structure for vlan configuration

Data fields

Type	Field	Description
InterfaceType	iface	PHY interfaces (i.e. ETH, ECM and RNDIS)
int16_t	vlanId	Vlan identifier (i.e 1-4094)
bool	isAccelerated	is acceleration allowed
uint8_t	priority	Vlan priority - A 3-bit field which refers to the IEEE 802.1p class of service to traffic priority level. Don't care = 0

4.47.2.16 struct telux::data::FlowDataRate

QOS Flow data min max rate bits per seconds

Data fields

Type	Field	Description
uint64_t	maxRate	QOS Flow maximum data rate
uint64_t	minRate	QOS Flow minimum data rate

4.47.2.17 struct telux::data::QosIPFlowInfo

QOS Flow IP info

Data fields

Type	Field	Description
QosIPFlow↩ Mask	mask	Valid parameters of QosIPFlowInfo QosIPFlowMaskType
IpTraffic↩ ClassType	tfClass	IP Traffic class type IpTrafficClassType
FlowDataRate	dataRate	Flow data rate FlowDataRate

4.47.2.18 class telux::data::DataFactory

DataFactory is the central factory to create all data classes.

Public member functions

- virtual std::shared_ptr< IDataConnectionManager > getDataConnectionManager (SlotId slotId=DEFAULT_SLOT_ID, telux::common::InitResponseCb clientCallback=nullptr)=0
- virtual std::shared_ptr< IDataProfileManager > getDataProfileManager (SlotId slotId=DEFAULT_SLOT_ID, telux::common::InitResponseCb clientCallback=nullptr)=0
- virtual std::shared_ptr< IServingSystemManager > getServingSystemManager (SlotId slotId=DEFAULT_SLOT_ID, telux::common::InitResponseCb clientCallback=nullptr)=0
- virtual std::shared_ptr< IDataFilterManager > getDataFilterManager (SlotId slotId=DEFAULT_SLOT_ID, telux::common::InitResponseCb clientCallback=nullptr)=0
- virtual std::shared_ptr< telux::data::net::INatManager > getNatManager (telux::data::OperationType oprType, telux::common::InitResponseCb clientCallback=nullptr)=0
- virtual std::shared_ptr< telux::data::net::IFirewallManager > getFirewallManager (telux::data::OperationType oprType, telux::common::InitResponseCb clientCallback=nullptr)=0
- virtual std::shared_ptr< telux::data::net::IFirewallEntry > getNewFirewallEntry (IpProtocol proto, Direction direction, IpFamilyType ipFamilyType)=0
- virtual std::shared_ptr< IIpFilter > getNewIpFilter (IpProtocol proto)=0
- virtual std::shared_ptr< telux::data::net::IVlanManager > getVlanManager (telux::data::OperationType oprType, telux::common::InitResponseCb clientCallback=nullptr)=0
- virtual std::shared_ptr< telux::data::net::ISocksManager > getSocksManager (telux::data::OperationType oprType, telux::common::InitResponseCb clientCallback=nullptr)=0
- virtual std::shared_ptr< telux::data::net::IBridgeManager > getBridgeManager (telux::common::InitResponseCb clientCallback=nullptr)=0
- virtual std::shared_ptr< telux::data::net::IL2tpManager > getL2tpManager (telux::common::InitResponseCb clientCallback=nullptr)=0
- virtual std::shared_ptr< telux::data::IDataSettingsManager > getDataSettingsManager (telux::data::OperationType oprType, telux::common::InitResponseCb clientCallback=nullptr)=0

Static Public Member Functions

- static DataFactory & getInstance ()

4.47.2.18.1 Member Function Documentation

4.47.2.18.1.1 static DataFactory& telux::data::DataFactory::getInstance ( ) [static]

Get Data Factory instance.

4.47.2.18.1.2 virtual std::shared_ptr<IDataConnectionManager> telux::data::DataFactory::getDataConnectionManager (SlotId *slotId* = *DEFAULT_SLOT_ID*, telux::common::InitResponseCb *clientCallback* = *nullptr*) [pure virtual]

Get Data Connection Manager

Parameters

in	*slotId*	Unique identifier for the SIM slot
in	*clientCallback*	Optional callback to get the initialization status of DataConnectionManager telux::common::InitResponseCb

Returns

instance of IDataConnectionManager

4.47.2.18.1.3 virtual std::shared_ptr<IDataProfileManager> telux::data::DataFactory::getDataProfileManager (SlotId *slotId = DEFAULT_SLOT_ID,* telux::common::InitResponseCb *clientCallback = nullptr*) [pure virtual]

Get Data Profile Manager

Parameters

in	*slotId*	Unique identifier for the SIM slot
in	*clientCallback*	Optional callback to get the initialization status of DataProfileManager telux::common::InitResponseCb

Returns

instance of IDataProfileManager

4.47.2.18.1.4 virtual std::shared_ptr<IServingSystemManager> telux::data::DataFactory::getServingSystemManager (SlotId *slotId = DEFAULT_SLOT_ID,* telux::common::InitResponseCb *clientCallback = nullptr*) [pure virtual]

Get Serving System Manager

Parameters

in	*slotId*	Unique identifier for the SIM slot
in	*clientCallback*	Optional callback to get the initialization status of ServingSystemManager telux::common::InitResponseCb

Returns

instance of IServingSystemManager

4.47.2.18.1.5 virtual std::shared_ptr<IDataFilterManager> telux::data::DataFactory::getDataFilterManager (SlotId *slotId = DEFAULT_SLOT_ID,* telux::common::InitResponseCb *clientCallback = nullptr*) [pure virtual]

Get Data Filter Manager instance

Parameters

in	*slotId*	Unique identifier for the SIM slot
in	*clientCallback*	Optional callback to get the initialization status of Serving System Manager telux::common::InitResponseCb

Returns

instance of IDataFilterManager.

4.47.2.18.1.6 virtual std::shared_ptr<telux::data::net::INatManager> telux::data::DataFactory::getNatManager (telux::data::OperationType *oprType*, telux::common::InitResponseCb *clientCallback* = `nullptr`) [pure virtual]

Get Network Address Translation(NAT) Manager

Parameters

in	*oprType*	Required operation type telux::data::OperationType
in	*clientCallback*	Optional callback to get the initialization status of NAT manager telux::common::InitResponseCb

Returns

instance of INatManager or nullptr if NAT management is not supported

4.47.2.18.1.7 virtual std::shared_ptr<telux::data::net::IFirewallManager> telux::data::DataFactory::getFirewallManager (telux::data::OperationType *oprType*, telux::common::InitResponseCb *clientCallback* = `nullptr`) [pure virtual]

Get Firewall Manager

Parameters

in	*oprType*	Required operation type telux::data::OperationType
in	*clientCallback*	Optional callback to get the initialization status of Firewall manager telux::common::InitResponseCb

Returns

instance of IFirewallManager or nullptr if Firewall management is not supported

4.47.2.18.1.8 virtual std::shared_ptr<telux::data::net::IFirewallEntry> telux::data::DataFactory::getNewFirewallEntry (IpProtocol *proto*, Direction *direction*, IpFamilyType *ipFamilyType*) [pure virtual]

Get Firewall entry based on IP protocol and set respective filter (i.e. TCP or UDP)

Parameters

in	*proto*	telux::data::IpProtocol
in	*direction*	telux::data::Direction
in	*ipFamilyType*	Identifies IP family type telux::data::IpFamilyType

Returns

instance of IFirewallEntry

4.47.2.18.1.9 virtual std::shared_ptr<IIpFilter> telux::data::DataFactory::getNewIpFilter (IpProtocol *proto*) [pure virtual]

Get IIpFilter instance based on IP Protocol, This can be used in Firewall Manager and Data Filter Manager

Parameters

in	*proto*	telux::data::IpProtocol Some sample protocol values are ICMP = 1 # Internet Control Message Protocol - RFC 792 IGMP = 2 # Internet Group Management Protocol - RFC 1112 TCP = 6 # Transmission Control Protocol - RFC 793 UDP = 17 # User Datagram Protocol - RFC 768 ESP = 50 # Encapsulating Security Payload - RFC 4303

Returns

instance of IIpFilter based on IpProtocol filter (i.e TCP, UDP)

4.47.2.18.1.10 virtual std::shared_ptr<telux::data::net::IVlanManager> telux::data::DataFactory::getVlanManager (telux::data::OperationType *oprType,* telux::common::InitResponseCb *clientCallback = nullptr*) [pure virtual]

Get VLAN Manager

Parameters

in	*oprType*	Required operation type telux::data::OperationType
in	*clientCallback*	Optional callback to get the initialization status of Vlan manager telux::common::InitResponseCb

Returns

instance of IVlanManager

4.47.2.18.1.11 virtual std::shared_ptr<telux::data::net::ISocksManager> telux::data::DataFactory::getSocksManager (telux::data::OperationType *oprType,* telux::common::InitResponseCb *clientCallback = nullptr*) [pure virtual]

Get Socks Manager

Parameters

in	*oprType*	Required operation type telux::data::OperationType
in	*clientCallback*	Optional callback to get the initialization status of Socks manager telux::common::InitResponseCb

Returns

instance of ISocksManager or nullptr if Socks management is not supported

4.47.2.18.1.12 virtual std::shared_ptr<telux::data::net::IBridgeManager> telux::data::DataFactory::getBridgeManager (telux::common::InitResponseCb *clientCallback = nullptr*) [pure virtual]

Get Software Bridge Manager

Parameters

in	*clientCallback*	Optional callback to get the initialization status of Bridge manager telux::common::InitResponseCb

Returns

instance of IBridgeManager

4.47.2.18.1.13 virtual std::shared_ptr<telux::data::net::IL2tpManager> telux::data::DataFactory::getL2tpManager (telux::common::InitResponseCb *clientCallback = nullptr*) [pure virtual]

Get L2TP Manager

Parameters

in	*clientCallback*	Optional callback to get the initialization status of L2TP manager telux::common::InitResponseCb

Returns

instance of IL2tpManager

4.47.2.18.1.14 virtual std::shared_ptr<telux::data::IDataSettingsManager> telux::data::DataFactory::getDataSettingsManager (telux::data::OperationType *oprType,* telux::common::InitResponseCb *clientCallback = nullptr*) [pure virtual]

Get Data Settings Manager

Parameters

in	*oprType*	Required operation type telux::data::OperationType
in	*clientCallback*	Optional callback to get the initialization status of Data Settings manager telux::common::InitResponseCb

Returns

instance of IDataSettingsManager

4.47.2.19 class telux::data::IDataFilterListener

Listener class for listening to filtering mode notifications, like Data filtering mode change. Client need to implement these methods. The methods in listener can be invoked from multiple threads. So the client needs to make sure that the implementation is thread-safe.

Public member functions

- virtual void onDataRestrictModeChange (DataRestrictMode mode)
- virtual ~IDataFilterListener ()

4.47.2.19.1 Constructors and Destructors

4.47.2.19.1.1 virtual telux::data::IDataFilterListener::~IDataFilterListener ( ) [virtual]

Destructor of IDataFilterListener

4.47.2.19.2 Member Function Documentation

4.47.2.19.2.1 virtual void telux::data::IDataFilterListener::onDataRestrictModeChange (DataRestrictMode *mode*) [virtual]

This function is called when the data filtering mode is changed for the packet data session.

Parameters

in	*mode*	- state the current data filter mode

4.47.2.20 class telux::data::IDataFilterManager

IDataFilterManager class provides interface to enable/disable the data restrict filters and register for data restrict filter. The filtering can be done at any time. One such use case is to do it when we want the AP to suspend so that we are not waking up the AP due to spurious incoming messages. Also to make sure the DataRestrict mode is enabled.

In contrary to when DataRestrict mode is disabled, modem will forward all the incoming data packets to AP and might wake up AP unnecessarily.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual bool isReady ()=0
- virtual std::future< bool > onReady ()=0
- virtual telux::common::Status registerListener (std::weak_ptr< IDataFilterListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< IDataFilterListener > listener)=0
- virtual telux::common::Status setDataRestrictMode (DataRestrictMode mode, telux::common::ResponseCallback callback=nullptr, int profileId=PROFILE_ID_MAX, IpFamilyType ipFamilyType=IpFamilyType::UNKNOWN)=0
- virtual telux::common::Status requestDataRestrictMode (std::string ifaceName, DataRestrictModeCb callback)=0
- virtual telux::common::Status addDataRestrictFilter (std::shared_ptr< IIpFilter > &filter, telux::common::ResponseCallback callback=nullptr, int profileId=PROFILE_ID_MAX, IpFamilyType ipFamilyType=IpFamilyType::UNKNOWN)=0
- virtual telux::common::Status removeAllDataRestrictFilters (telux::common::ResponseCallback callback=nullptr, int profileId=PROFILE_ID_MAX, IpFamilyType ipFamilyType=IpFamilyType::UNKNOWN)=0
- virtual SlotId getSlotId ()=0
- virtual ~IDataFilterManager ()

4.47.2.20.1 Constructors and Destructors

4.47.2.20.1.1 virtual telux::data::IDataFilterManager::~IDataFilterManager ( ) [virtual]

Destructor of IDataFilterManager

4.47.2.20.2 Member Function Documentation

4.47.2.20.2.1 virtual telux::common::ServiceStatus telux::data::IDataFilterManager::getServiceStatus ( ) [pure virtual]

Checks the status of data filter manager and returns the result.

Returns

the status of sensor sub-system status telux::common::ServiceStatus

4.47.2.20.2.2 virtual bool telux::data::IDataFilterManager::isReady ( ) [pure virtual]

Checks the status of Data Filter Service and if the other APIs are ready for use, and returns the result.

Returns

True if the services are ready otherwise false.

Deprecated

Use getServiceStatus API.

4.47.2.20.2.3 virtual std::future<bool> telux::data::IDataFilterManager::onReady ( ) [pure virtual]

Wait for Data Filter Service to be ready.

Returns

A future that caller can wait on to be notified when Data Filter Service are ready.

Deprecated

Use InitResponseCb callback in factory API getDataFilterManager.

4.47.2.20.2.4 virtual telux::common::Status telux::data::IDataFilterManager::registerListener (std::weak_ptr< IDataFilterListener > *listener*) [pure virtual]

Register a listener for powersave filtering mode notifications.

Parameters

in	*listener*	- Pointer of IDataFilterListener object that processes the notification

Returns

Status of registerListener i.e success or suitable status code.

4.47.2.20.2.5 virtual telux::common::Status telux::data::IDataFilterManager::deregisterListener (std::weak_ptr< IDataFilterListener > *listener*) [pure virtual]

Remove a previously registered listener.

Parameters

in	*listener*	- Previously registered IDataFilterListener that needs to be removed

Returns

Status of deregisterListener, success or suitable status code

4.47.2.20.2.6 virtual telux::common::Status telux::data::IDataFilterManager::setDataRestrictMode (DataRestrictMode *mode*, telux::common::ResponseCallback *callback* = `nullptr`, int *profileId* = PROFILE_ID_MAX, IpFamilyType *ipFamilyType* = IpFamilyType::UNKNOWN) `[pure virtual]`

Changes the Data Powersave filter mode and auto exit feature.

This API enables or disables the powersave filtering mode of the running packet data session. If a data connection is torn down and brought up again, then previous filter mode setting does not persist for that data call session, and requires to be enabled again.

On platforms with Access control enabled, Caller needs to have TELUX_DATA_FILTER_OPS permission to invoke this API successfully.

Parameters

in	*mode*	- Enable or disable the powersave filtering mode.
in	*callback*	- Optional callback to get the response for the change in filter mode.
in	*profileId*	- Optional Profile ID for data connection. If user does not specify the profile id, then the API applies to all the currently running data connection. If user wants to apply the changes to any specific data connection, then its profile id can be specified as input.
in	*ipFamilyType*	- Optional IP Family type IpFamilyType . If user does not specify the ip family type, then the API applies to all the currently running data connection. If user wants to apply the changes to any specific data connection, then its ip family type can be specified as input.

Returns

Status of setDataRestrictMode i.e. success or suitable status code.

4.47.2.20.2.7 virtual telux::common::Status telux::data::IDataFilterManager::requestDataRestrictMode (std::string *ifaceName*, DataRestrictModeCb *callback*) `[pure virtual]`

Get the current Data Powersave filter mode

Parameters

in	*ifaceName*	- Interface name for data connection. Note: For global pdn , ifaceName must be empty, as global restrict mode is reported. Per-pdn requests are not supported.
in	*callback*	- callback function to get the result of API.

Returns

Status of requestDataRestrictMode i.e. success or suitable status code.

4.47.2.20.2.8 virtual telux::common::Status telux::data::IDataFilterManager::addDataRestrictFilter (std::shared_ptr< IIpFilter > & *filter*, telux::common::ResponseCallback *callback* = `nullptr`, int *profileId* = PROFILE_ID_MAX, IpFamilyType *ipFamilyType* = IpFamilyType::UNKNOWN) [pure virtual]

This API adds a filter rules for a packet data session to achieve power savings. In case when DataRestrict mode is enabled and AP is in suspended state, Modem will filter all the incoming data packet and route them to AP only if filter rules added via addDataRestrictFilter API matches the criteria, else they are queued at Modem itself and not forwarded to AP, until filter mode is disabled.

On platforms with Access control enabled, Caller needs to have TELUX_DATA_FILTER_OPS permission to invoke this API successfully.

Parameters

in	*filter*	- Filter rule.
in	*callback*	- Optional callback to get the response.
in	*profileId*	- Optional Profile ID for data connection. If user does not specify the profile id, then the API applies to all the currently running data connection. If user wants to apply the changes to any specific data connection, then its profile id can be specified as input.
in	*ipFamilyType*	- Optional IP Family type IpFamilyType. If user does not specify the ip family type, then the API applies to all the currently running data connection. If user wants to apply the changes to any specific data connection, then its ip family type can be specified as input.

Returns

Status of addDataRestrictFilter i.e. success or suitable status code.

4.47.2.20.2.9 virtual telux::common::Status telux::data::IDataFilterManager::removeAllDataRestrictFilters (telux::common::ResponseCallback *callback* = `nullptr`, int *profileId* = PROFILE_ID_MAX, IpFamilyType *ipFamilyType* = IpFamilyType::UNKNOWN) [pure virtual]

This API removes all the previous added powersave filter for a packet data session

On platforms with Access control enabled, Caller needs to have TELUX_DATA_FILTER_OPS permission to invoke this API successfully.

Parameters

in	*callback*	- Optional callback to get the response.
in	*profileId*	- Optional Profile ID for data connection. If user does not specify the profile id, then the API applies to all the currently running data connection. If user wants to apply the changes to any specific data connection, then its profile id can be specified as input.
in	*ipFamilyType*	- Optional IP Family type IpFamilyType. If user does not specify the ip family type, then the API applies to all the currently running data connection. If user wants to apply the changes to any specific data connection, then its ip family type can be specified as input.

Returns

Status of removeAllDataRestrictFilters i.e. success or suitable status code.

4.47.2.20.2.10 virtual SlotId telux::data::IDataFilterManager::getSlotId ( ) [pure virtual]

Get associated slot id for the Data Filter Manager.

Returns

SlotId

4.47.2.21 class telux::data::DataProfile

DataProfile class represents single data profile on the modem.

Public member functions

- DataProfile (int id, const std::string &name, const std::string &apn, const std::string &username, const std::string &password, IpFamilyType ipFamilyType, TechPreference techPref, AuthProtocolType authType, ApnTypes apnTypes)
- int getId ()
- std::string getName ()
- std::string getApn ()
- std::string getUserName ()
- std::string getPassword ()
- TechPreference getTechPreference ()
- AuthProtocolType getAuthProtocolType ()
- IpFamilyType getIpFamilyType ()
- ApnTypes getApnTypes ()

- std::string toString ()

Static Public Attributes

- static constexpr int PROFILE_ID_INVALID = -1

4.47.2.21.1 Constructors and Destructors

4.47.2.21.1.1 telux::data::DataProfile::DataProfile (int *id,* const std::string & *name,* const std::string & *apn,* const std::string & *username,* const std::string & *password,* IpFamilyType *ipFamilyType,* TechPreference *techPref,* AuthProtocolType *authType,* ApnTypes *apnTypes*)

4.47.2.21.2 Member Function Documentation

4.47.2.21.2.1 int telux::data::DataProfile::getId ( )

Get profile identifier.

Returns

profile id

4.47.2.21.2.2 std::string telux::data::DataProfile::getName ( )

Get profile name.

Returns

profile name

4.47.2.21.2.3 std::string telux::data::DataProfile::getApn ( )

Get Access Point Name (APN) name.

Returns

APN name

4.47.2.21.2.4 std::string telux::data::DataProfile::getUserName ( )

Get profile user name.

Returns

user name

4.47.2.21.2.5 std::string telux::data::DataProfile::getPassword ( )

Get profile password.

Returns

profile password

4.47.2.21.2.6 TechPreference telux::data::DataProfile::getTechPreference ( )

Get technology preference.

Returns

TechPreference TechPreference

4.47.2.21.2.7 AuthProtocolType telux::data::DataProfile::getAuthProtocolType ( )

Get authentication preference.

Returns

AuthProtocolType AuthProtocolType

4.47.2.21.2.8 IpFamilyType telux::data::DataProfile::getIpFamilyType ( )

Get IP Family type.

Returns

IpFamilyType IpFamilyType

4.47.2.21.2.9 ApnTypes telux::data::DataProfile::getApnTypes ( )

Get Apn type mask.

Returns

ApnTypes ApnTypes

4.47.2.21.2.10 std::string telux::data::DataProfile::toString ( )

Get the text related informative representation of this object.

Returns

String containing informative string.

4.47.2.21.3 Field Documentation

4.47.2.21.3.1 constexpr int telux::data::DataProfile::PROFILE_ID_INVALID = -1 [static]

4.47.2.22 class telux::data::IDataProfileListener

Listener class for getting profile change notification.

The methods in the listener can be invoked from multiple threads. It is client's responsibility to make sure the implementation is thread safe.

Public member functions

- virtual void onServiceStatusChange (telux::common::ServiceStatus status)
- virtual void onProfileUpdate (int profileId, TechPreference techPreference, ProfileChangeEvent event)
- virtual ~IDataProfileListener ()

4.47.2.22.1 Constructors and Destructors

4.47.2.22.1.1 virtual telux::data::IDataProfileListener::~IDataProfileListener ( ) [virtual]

Destructor of IDataProfileListener

4.47.2.22.2 Member Function Documentation

4.47.2.22.2.1 virtual void telux::data::IDataProfileListener::onServiceStatusChange (telux::common::ServiceStatus *status*) [virtual]

This function is called when service status changes.

Parameters

in	*status*	- ServiceStatus

4.47.2.22.2.2 virtual void telux::data::IDataProfileListener::onProfileUpdate (int *profileId*, TechPreference *techPreference*, ProfileChangeEvent *event*) [virtual]

This function is called when profile change happens.

Parameters

in	*profileId*	- ID of the updated profile.
in	*techPreference*	- TechPreference.
in	*event*	- Event that caused the change in profile.

4.47.2.23 class telux::data::IDataProfileManager

IDataProfileManager is a primary interface for profile management.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual bool isSubsystemReady ()=0
- virtual std::future< bool > onSubsystemReady ()=0
- virtual telux::common::Status requestProfileList (std::shared_ptr< IDataProfileListCallback > callback=nullptr)=0
- virtual telux::common::Status createProfile (const ProfileParams &profileParams, std::shared_ptr< IDataCreateProfileCallback > callback=nullptr)=0
- virtual telux::common::Status deleteProfile (uint8_t profileId, TechPreference techPreference, std::shared_ptr< telux::common::ICommandResponseCallback > callback=nullptr)=0
- virtual telux::common::Status modifyProfile (uint8_t profileId, const ProfileParams &profileParams, std::shared_ptr< telux::common::ICommandResponseCallback > callback=nullptr)=0
- virtual telux::common::Status queryProfile (const ProfileParams &profileParams, std::shared_ptr< IDataProfileListCallback > callback=nullptr)=0
- virtual telux::common::Status requestProfile (uint8_t profileId, TechPreference techPreference, std::shared_ptr< IDataProfileCallback > callback=nullptr)=0
- virtual int getSlotId ()=0
- virtual telux::common::Status registerListener (std::weak_ptr< telux::data::IDataProfileListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< telux::data::IDataProfileListener > listener)=0
- virtual ~IDataProfileManager ()

4.47.2.23.1 Constructors and Destructors

4.47.2.23.1.1 virtual telux::data::IDataProfileManager::~IDataProfileManager ( ) [virtual]

Destructor for IDataProfileManager

4.47.2.23.2 Member Function Documentation

4.47.2.23.2.1 virtual telux::common::ServiceStatus telux::data::IDataProfileManager::getServiceStatus ( ) [pure virtual]

Checks the status of Data profile manager and returns the result.

Returns

SERVICE_AVAILABLE If Data profile manager is ready for service. SERVICE_UNAVAILABLE If Data profile manager is temporarily unavailable. SERVICE_FAILED - If Data profile manager encountered an irrecoverable failure.

4.47.2.23.2.2 virtual bool telux::data::IDataProfileManager::isSubsystemReady ( ) [pure virtual]

Checks if the data profile manager is ready.

Returns

True if data profile subsystem is ready for service otherwise false.

Deprecated

Use getServiceStatus API.

4.47.2.23.2.3 virtual std::future<bool> telux::data::IDataProfileManager::onSubsystemReady ( ) [pure virtual]

Waits for data profile subsystem to be ready.

Returns

A future that caller can wait on to be notified when data profile subsystem is ready.

Deprecated

Use InitResponseCb callback in factory API getDataProfileManager.

4.47.2.23.2.4 virtual telux::common::Status telux::data::IDataProfileManager::requestProfileList (std::shared_ptr< IDataProfileListCallback > *callback = nullptr*) [pure virtual]

Request list of profiles supported by the device.

Parameters

in,out	*callback*	Callback pointer to get the response.

Returns

Status of request profile i.e. success or suitable error code.

4.47.2.23.2.5 virtual telux::common::Status telux::data::IDataProfileManager::createProfile (const ProfileParams & *profileParams,* std::shared_ptr< IDataCreateProfileCallback > *callback = nullptr*) [pure virtual]

Create profile based on data profile params.

On platforms with Access control enabled, Caller needs to have TELUX_DATA_PROFILE_OPS permission to invoke this API successfully.

Parameters

in	*profileParams*	profileParams configuration to be passed for creating profile either for 3GPP or 3GPP2
in,out	*callback*	Callback pointer to get the result of create profile

Returns

Status of create profile i.e. success or suitable error code.

4.47.2.23.2.6 virtual telux::common::Status telux::data::IDataProfileManager::deleteProfile (uint8_t *profileId,* TechPreference *techPreference,* std::shared_ptr< telux::common::ICommand↩ ResponseCallback > *callback = nullptr*) [pure virtual]

Delete profile corresponding to profile identifier.

The deletion of a profile does not affect profile index assignments.

On platforms with Access control enabled, Caller needs to have TELUX_DATA_PROFILE_OPS permission to invoke this API successfully.

Parameters

in	*profileId*	Profile identifier
in	*techPreference*	Technology Preference like 3GPP / 3GPP2
in	*callback*	Callback pointer to get the result of delete profile

Returns

Status of delete profile i.e. success or suitable error code.

4.47.2.23.2.7 virtual telux::common::Status telux::data::IDataProfileManager::modifyProfile (uint8_t *profileId,* const ProfileParams & *profileParams,* std::shared_ptr< telux::common::ICommandResponseCallback > *callback = nullptr*) [pure virtual]

Modify existing profile with new profile params.

On platforms with Access control enabled, Caller needs to have TELUX_DATA_PROFILE_OPS permission to invoke this API successfully.

Parameters

in	*profileId*	Profile identifier of profile to be modified
in	*profileParams*	New profileParams configuration passed for updating existing profile
in	*callback*	Callback pointer to get the result of modify profile

Returns

Status of modify profile i.e. success or suitable error code.

4.47.2.23.2.8 virtual telux::common::Status telux::data::IDataProfileManager::queryProfile (const ProfileParams & *profileParams*, std::shared_ptr< IDataProfileListCallback > *callback* = *nullptr*) [pure virtual]

Lookup modem profile/s based on given profile params.

Parameters

in	*profileParams*	ProfileParams configuration to be passed
in	*callback*	Callback pointer to get the result of query profile

Returns

Status of query profile i.e. success or suitable error code.

4.47.2.23.2.9 virtual telux::common::Status telux::data::IDataProfileManager::requestProfile (uint8_t *profileId*, TechPreference *techPreference*, std::shared_ptr< IDataProfileCallback > *callback* = *nullptr*) [pure virtual]

Get data profile corresponding to profile identifier.

Parameters

in	*profileId*	Profile identifier
in	*techPreference*	Technology preference • TechPreference
in	*callback*	Callback pointer to get the result of get profile by ID

Returns

Status of requestProfile i.e. success or suitable error code.

4.47.2.23.2.10 virtual int telux::data::IDataProfileManager::getSlotId ( ) [pure virtual]

Get associated slot id for the Data Profile Manager.

Returns

SlotId

4.47.2.23.2.11 virtual telux::common::Status telux::data::IDataProfileManager::registerListener (std::weak_ptr< telux::data::IDataProfileListener > *listener*) [pure virtual]

Listen for create, delete and modify profile events.

Parameters

in	*listener*	- Listener that processes the notification.

Returns

telux::common::Status

4.47.2.23.2.12 virtual telux::common::Status telux::data::IDataProfileManager::deregisterListener (std::weak_ptr< telux::data::IDataProfileListener > *listener*) [pure virtual]

De-register listener.

Parameters

in	*listener*	- Listener to be de-registered.

Returns

telux::common::Status

4.47.2.24 class telux::data::IDataCreateProfileCallback

Interface for create profile callback object. Client needs to implement this interface to get single shot responses for command like create profile.

The methods in callback can be invoked from multiple different threads. The implementation should be thread safe.

Public member functions

- virtual void onResponse (int profileId, telux::common::ErrorCode error)

4.47.2.24.1 Member Function Documentation

4.47.2.24.1.1 virtual void telux::data::IDataCreateProfileCallback::onResponse (int *profileId,* telux::common::ErrorCode *error*) [virtual]

This function is called with the response to IDataProfileManager::createProfile API.

Parameters

in	*profileId*	created profile Id for the response. Use IDataProfileManager::requestProfile to get the data profile
in	*error*	telux::common::ErrorCode

4.47.2.25 class telux::data::IDataProfileListCallback

Interface for getting list of DataProfile using callback. Client needs to implement this interface to get single shot responses for commands like get profile list and query profile.

The methods in callback can be invoked from different threads. The implementation should be thread safe.

Public member functions

- virtual void onProfileListResponse (const std::vector< std::shared_ptr< DataProfile >> &profiles, telux::common::ErrorCode error)

4.47.2.25.1 Member Function Documentation

4.47.2.25.1.1 virtual void telux::data::IDataProfileListCallback::onProfileListResponse (const std::vector< std::shared_ptr< DataProfile >> & *profiles,* telux::common::ErrorCode *error*) [virtual]

This function is called with the response to requestProfileList API or queryProfile API.

Parameters

in	*profiles*	List of profiles supported by the device
in	*error*	telux::common::ErrorCode

4.47.2.26 class telux::data::IDataProfileCallback

Interface for getting DataProfile using callback. Client needs to implement this interface to get single shot responses for command like create profile.

The methods in callback can be invoked from multiple different threads. The implementation should be thread safe.

Public member functions

- virtual void onResponse (const std::shared_ptr< DataProfile > &profile, telux::common::ErrorCode error)

4.47.2.26.1 Member Function Documentation

4.47.2.26.1.1 virtual void telux::data::IDataProfileCallback::onResponse (const std::shared_ptr< DataProfile > & *profile,* telux::common::ErrorCode *error*) [virtual]

This function is called with the response to IDataProfileManager::requestProfile API.

Parameters

in	*profile*	Response of data profile
in	*error*	telux::common::ErrorCode

4.47.2.27 struct telux::data::DdsInfo

Specifies the DDS switch information.

Data fields

Type	Field	Description
DdsType	type	
SlotId	slotId	Specifies DDS switch type

4.47.2.28 struct telux::data::BandInterferenceConfig

N79 5G/Wlan 5GHz interference avoidance configuration

Data fields

Type	Field	Description
BandPriority	priority	
uint32_t	wlanWait↩ TimeInSec	Priority settings for N79/Wlan 5G
uint32_t	n79WaitTime↩ InSec	If Wlan 5GHz has higher priority and suffers signal drop, modem will wait for period of time specified here for Wlan signal to recover before enabeling N79 5G.

4.47.2.29 class telux::data::IDataSettingsManager

Data Settings Manager class provides APIs related to the data subsystem settings. For example, ability to reset current network settings to factory settings, setting backhaul priority, and enabling roaming per PDN.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual telux::common::Status restoreFactorySettings (OperationType operationType, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status setBackhaulPreference (std::vector< BackhaulType > backhaulPref, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestBackhaulPreference (RequestBackhaulPrefResponseCb callback)=0
- virtual telux::common::Status setBandInterferenceConfig (bool enable, std::shared_ptr< BandInterferenceConfig > config=nullptr, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestBandInterferenceConfig (RequestBandInterferenceConfigResponseCb callback)=0
- virtual telux::common::Status requestDdsSwitch (DdsInfo request, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestCurrentDds (RequestCurrentDdsResponseCb callback)=0
- virtual telux::common::Status setWwanConnectivityConfig (SlotId slotId, bool allow, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestWwanConnectivityConfig (SlotId slotId, requestWwanConnectivityConfigResponseCb callback)=0
- virtual telux::common::Status setMacSecState (bool enable, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestMacSecState (RequestMacSecSateResponseCb callback)=0
- virtual telux::common::Status registerListener (std::weak_ptr< IDataSettingsListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< IDataSettingsListener > listener)=0

4.47.2.29.1 Member Function Documentation

4.47.2.29.1.1 virtual telux::common::ServiceStatus telux::data::IDataSettingsManager::getServiceStatus ( ) [pure virtual]

Checks the status of Data Settings manager object and returns the result.

Returns

SERVICE_AVAILABLE - If Data Settings manager object is ready for service.
SERVICE_UNAVAILABLE - If Data Settings manager object is temporarily unavailable.
SERVICE_FAILED - If Data Settings manager object encountered an irrecoverable failure.

4.47.2.29.1.2 virtual telux::common::Status telux::data::IDataSettingsManager::restoreFactorySettings (OperationType *operationType*, telux::common::ResponseCallback *callback* = `nullptr`) [pure virtual]

Resets current network settings to initial setting configured in factory. Factory settings are the initial network settings generated during manufacturing process. After successful reset, device will reboot with factory network settings.

Parameters

in	*operationType*	telux::data::OperationType
in	*callback*	callback to get the response to restoreFactorySettings

Returns

Immediate status of restoreFactorySettings i.e. success or suitable status.

Note

Eval: This is a new API and is being evaluated. It is subject to change.

4.47.2.29.1.3 virtual telux::common::Status telux::data::IDataSettingsManager::setBackhaulPreference (std::vector< BackhaulType > *backhaulPref*, telux::common::ResponseCallback *callback* = `nullptr`) [pure virtual]

Set backhaul preference for bridge0 (default bridge) traffic. Bridge0 Traffic routing to backhaul will be attempted on first to least preferred. For instance if backhaul vector contains ETH, USB, and WWAN, bridge0 traffic routing will be attempted on ETH first, then USB and finally WWAN backhaul. Configuration changes will be persistent across reboots.

On platforms with Access control enabled, Caller needs to have TELUX_DATA_SETTING permission to invoke this API successfully.

Parameters

in	*backhaulPref*	vector of telux::data::BackhaulType which contains the order of backhaul preference to be used when connecting to external network. First element is most preferred and last element is least preferred backhaul.
in	*callback*	callback to get response for setBackhaulPreference.

Returns

Status of setBackhaulPreference i.e. success or suitable status code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.47.2.29.1.4 virtual telux::common::Status telux::data::IDataSettingsManager::requestBackhaulPreference (RequestBackhaulPrefResponseCb *callback*) [pure virtual]

Request current backhaul preference for bridge0 (default bridge) traffic.

Parameters

in	*callback*	callback to get response for requestBackhaulPreference.

Returns

Status of requestBackhaulPreference i.e. success or suitable status code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.47.2.29.1.5 virtual telux::common::Status telux::data::IDataSettingsManager::setBandInterferenceConfig (bool *enable*, std::shared_ptr< BandInterferenceConfig > *config* = *nullptr*, telux::common::ResponseCallback *callback* = *nullptr*) [pure virtual]

Configure N79 5G and Wlan 5GHz band priority. Sets priority for modem to use either 5GHz Wlan or N79 5G band when they are both available to avoid interference. In case N79 5G is configured as higher priority: If N79 5G becomes available while 5G Wlan is enabled, Wlan (AP/Sta) will be moved to 2.4 GHz. If N79 5G becomes unavailable for telux::data::BandInterferenceConfig::n79WaitTimeInSec time period, Wlan will be moved to 5GHz. In case Wlan 5GHz is configured as higher priority: If Wlan 5GHz (AP/Sta) becomes available while N79 5G is enabled, N79 5G will be disabled. If Wlan 5GHz becomes unavailable for telux::data::BandInterferenceConfig::wlanWaitTimeInSec period and N79 5G is available, N79 will be enabled.

On platforms with Access control enabled, Caller needs to have TELUX_DATA_SETTING permission to invoke this API successfully.

Parameters

in	*enable*	True: enable interference management. False: disable interference management
in	*config*	N79 5G /Wlan 5GHz band interference configuration telux::data::BandInterferenceConfig
in	*callback*	callback to get response for setBandInterferenceConfig.

Returns

Status of setBandInterferenceConfig i.e. success or suitable status code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.47.2.29.1.6 virtual telux::common::Status telux::data::IDataSettingsManager::requestBand↩ InterferenceConfig (RequestBandInterferenceConfigResponseCb *callback*) [pure virtual]

Request N79 5G and Wlan 5GHz band priority settings. Request the configurations set by telux::data::setBandInterferenceConfig

Parameters

in	*callback*	callback to get response for requestBandInterferenceConfig.

Returns

Status of requestBandInterferenceConfig i.e. success or suitable status code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.47.2.29.1.7 virtual telux::common::Status telux::data::IDataSettingsManager::requestDdsSwitch (DdsInfo *request*, telux::common::ResponseCallback *callback* = *nullptr*) [pure virtual]

Allows the client to perform the DDS switch. Client has the option to either select permanent or temporary switch.

Parameters

in	*request*	Client has to provide the request telux::data::DdsInfo.

in	*callback*	Callback to get response for requestDdsSwitch. Possible ErrorCode in telux::common::ResponseCallback: • If the DDS switch is performed succesfully telux::common::ErrorCode::SUCCESS • If the DDS switch request is rejected telux::common::ErrorCode::OPERATION_NOT_ALLOWED The following scenarios are example of when a switch request will be rejected: 1. Slot1 is permanent DDS and the client attempts to trigger a permanent DDS switch on slot 1. 2. During an MT/MO voice call and the client attempts to trigger a permanent DDS switch. • If the DDS switch is allowed but due to some reason DDS switch failed telux::common::ErrorCode::GENERIC_FAILURE

Returns

Status of requestDdsSwitch, i.e., success or suitable status code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.47.2.29.1.8 virtual telux::common::Status telux::data::IDataSettingsManager::requestCurrentDds (RequestCurrentDdsResponseCb *callback*) [pure virtual]

Request the current DDS slot information

Parameters

in	*callback*	Callback to get response for requestCurrentDds.

Returns

Status of requestCurrentDds, i.e., success or suitable status code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.47.2.29.1.9 virtual telux::common::Status telux::data::IDataSettingsManager::setWwanConnectivityConfig (SlotId *slotId,* bool *allow,* telux::common::ResponseCallback *callback* = `nullptr`) [pure virtual]

Allow/Disallow WWAN connectivity. Controls whether system should allow/disallow WWAN connectivity to cellular network. Default setting is allow WWAN connectivity to cellular network.

- If client selects to disallow WWAN connectivity, any further attempts to start data calls using telux::data::IDataConnectionManager::startDataCall will fail with telux::common::ErrorCode::NOT_SUPPORTED. Data calls can be connected again only if client selects to allow WWAN connectivity.
- If client selects to disallow WWAN connectivity while data calls are already connected, all WWAN data calls will also be disconnected. Client will also receive telux::data::IDataConnectionListener::onDataCallInfoChanged notification with telux::data::IDataCall object status telux::data::DataCallStatus::NET_NO_NET for all impacted data calls. Configuration changes will be persistent across reboots.

On platforms with Access control enabled, Caller needs to have TELUX_DATA_SETTING permission to invoke this API successfully.

Parameters

in	*slotId*	Slot id on which WWAN connectivity to be allowed/disallowed
in	*allow*	True: allow connectivity, False: disallow connectivity
in	*callback*	optional callback to get response for setWwanConnectivityConfig.

Returns

Status of setWwanConnectivityConfig i.e. success or suitable status code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.47.2.29.1.10 virtual telux::common::Status telux::data::IDataSettingsManager::requestWwanConnectivityConfig (SlotId *slotId,* requestWwanConnectivityConfigResponseCb *callback*) [pure virtual]

Request current WWAN connectivity Configuration.

Parameters

in	*slotId*	Slot id for which WWAN connectivity to be reported.
in	*callback*	callback to get response for requestWwanConnectivityConfig.

Returns

Status of requestWwanConnectivityConfig i.e. success or suitable status code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.47.2.29.1.11 virtual telux::common::Status telux::data::IDataSettingsManager::setMacSecState (bool *enable,* telux::common::ResponseCallback *callback* = `nullptr`) [pure virtual]

Allows the client to set the MacSec state.

- If client enables the MacSec, post that the packets over the ethernet link will be encrypted.
- If client disables the MacSec, post that the packets over the ethernet link will not be encrypted.

Parameters

in	*enable*	True: enable the MacSec, False: disable the MacSec.
in	*callback*	Callback to get the setMacSecState response.

Returns

Status of setMacSecState, i.e., success or suitable status code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.47.2.29.1.12 virtual telux::common::Status telux::data::IDataSettingsManager::requestMacSecState (RequestMacSecSateResponseCb *callback*) [pure virtual]

Requests the current MacSec state.

Parameters

in	*callback*	callback to get response for requestMacSecState.

Returns

Status of requestMacSecState, i.e., success or suitable status code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.47.2.29.1.13 virtual telux::common::Status telux::data::IDataSettingsManager::registerListener (std::weak_ptr< IDataSettingsListener > *listener*) [pure virtual]

Register Data Settings Manager as listener for Data Service heath events like data service available or data service not available.

Parameters

in	*listener*	pointer of IDataSettingsListener object that processes the notification

Returns

Status of registerListener success or suitable status code

4.47.2.29.1.14 virtual telux::common::Status telux::data::IDataSettingsManager::deregisterListener (std::weak_ptr< IDataSettingsListener > *listener*) [pure virtual]

Removes a previously added listener.

Parameters

in	*listener*	pointer of IDataSettingsListener object that needs to be removed

Returns

Status of deregisterListener success or suitable status code

4.47.2.30 class telux::data::IDataSettingsListener

Interface for Data Settings listener object. Client needs to implement this interface to get access to Data Settings services notifications like onServiceStatusChange.

The methods in listener can be invoked from multiple different threads. The implementation should be thread safe.

Public member functions

- virtual void onServiceStatusChange (telux::common::ServiceStatus status)
- virtual void onWwanConnectivityConfigChange (SlotId slotId, bool isConnectivityAllowed)
- virtual void onDdsChange (DdsInfo currentState)
- virtual ~IDataSettingsListener ()

4.47.2.30.1 Constructors and Destructors

4.47.2.30.1.1 virtual telux::data::IDataSettingsListener::~IDataSettingsListener ( ) [virtual]

Destructor for IDataSettingsListener

4.47.2.30.2 Member Function Documentation

4.47.2.30.2.1 virtual void telux::data::IDataSettingsListener::onServiceStatusChange (telux::common::ServiceStatus *status*) [virtual]

This function is called when service status changes.

Parameters

in	*status*	- ServiceStatus

4.47.2.30.2.2 virtual void telux::data::IDataSettingsListener::onWwanConnectivityConfigChange (SlotId *slotId,* bool *isConnectivityAllowed*) [virtual]

This function is called when WWAN backhaul connectivity config changes.

Parameters

in	*slotId*	- Slot Id for which connectivity has changed.
in	*isConnectivityAllowed*	- Connectivity status allowed/disallowed.

4.47.2.30.2.3 virtual void telux::data::IDataSettingsListener::onDdsChange (DdsInfo *currentState*) [virtual]

Provides the current DDS state and is called whenever a DDS switch occurs.

Parameters

in	*currentState*	Provides the current DDS status. • Slot ID on which the DDS switch occured. • DDS switch type telux::data::DdsType.

4.47.2.31 struct telux::data::IPv4Info

IPv4 header info

Data fields

Type	Field	Description
string	srcAddr	address of the device that sends the packet.

Type	Field	Description
string	srcSubnetMask	
string	destAddr	address of receiving end
string	destSubnet↩ Mask	
TypeOfService	value	level of throughput, reliability, and delay
TypeOfService	mask	
IpProtocol	nextProtoId	Protocol ID (i.e TCP, UDP or ICMP)

4.47.2.32 struct telux::data::IPv6Info

IPv6 header info

Data fields

Type	Field	Description
string	srcAddr	address of the device that sends the packet.
uint8_t	srcPrefixLen	source prefix length used to create subnet
string	destAddr	address of receiving end
uint8_t	dstPrefixLen	destination prefix length used to create subnet
IpProtocol	nextProtoId	Protocol ID (i.e TCP, UDP or ICMP)
TrafficClass	val	indicates the class or priority of the IPv6 packet, enables the ability to track specific traffic flows at the network layer.
TrafficClass	mask	
FlowLabel	flowLabel	Indicates that this packet belongs to a specific sequence of packets between a source and destination, requiring special handling by intermediate IPv6 routers.
uint8_t	natEnabled	

4.47.2.33 struct telux::data::UdpInfo

UDP header info

Data fields

Type	Field	Description
PortInfo	src	Source port and range
PortInfo	dest	Destination port and range

4.47.2.34 struct telux::data::TcpInfo

TCP header info

Data fields

Type	Field	Description
PortInfo	src	Source port and range
PortInfo	dest	Destination port and range

4.47.2.35 struct telux::data::IcmpInfo

Internet Control Message Protocol (ICMP)

Data fields

Type	Field	Description
uint8_t	type	ICMP message type - RFC2780
uint8_t	code	ICMP message code - RFC2780

4.47.2.36 struct telux::data::EspInfo

Encapsulating Security Payload

Data fields

Type	Field	Description
uint32_t	spi	Security Parameters Index

4.47.2.37 class telux::data::IIpFilter

A IP filter class to add specific filters like what data will be allowed from the modem to the application processor. Only data packets that match the filter will be sent to the apps processor. Also used to configure Firewall rules.

Public member functions

- virtual IPv4Info getIPv4Info ()=0
- virtual telux::common::Status setIPv4Info (const IPv4Info &ipv4Info)=0
- virtual IPv6Info getIPv6Info ()=0
- virtual telux::common::Status setIPv6Info (const IPv6Info &ipv6Info)=0
- virtual IpProtocol getIpProtocol ()=0
- virtual IpFamilyType getIpFamily ()=0
- virtual ~IIpFilter ()

4.47.2.37.1 Constructors and Destructors

4.47.2.37.1.1 virtual telux::data::IIpFilter::~IIpFilter ( ) [virtual]

Destructor for IIpFilter

4.47.2.37.2 Member Function Documentation

4.47.2.37.2.1 virtual IPv4Info telux::data::IIpFilter::getIPv4Info ( ) [pure virtual]

Get the IPv4 header info

Returns

telux::data::IPv4Info

4.47.2.37.2.2 virtual telux::common::Status telux::data::IIpFilter::setIPv4Info (const IPv4Info & *ipv4Info*) [pure virtual]

sets the IPv4 header info

Parameters

in	*ipv4Info*	IPv4 structure telux::data::IPv4Info

Returns

Immediate status of setIPv4Info() request sent i.e. success or suitable status code.

4.47.2.37.2.3 virtual IPv6Info telux::data::IIpFilter::getIPv6Info ( ) [pure virtual]

Get the IPv6 header info

Returns

telux::data::IPv6Info

4.47.2.37.2.4 virtual telux::common::Status telux::data::IIpFilter::setIPv6Info (const IPv6Info & *ipv6Info*) [pure virtual]

sets the IPv6 header info

Parameters

in	*ipv6Info*	IPv6 structure telux::data::IPv6Info

Returns

Immediate status of setIPv6Info() request sent i.e. success or suitable status code.

4.47.2.37.2.5 virtual IpProtocol telux::data::IIpFilter::getIpProtocol ( ) [pure virtual]

Get the IpProtocol Number

Returns

telux::data::IpProtocol

4.47.2.37.2.6 virtual IpFamilyType telux::data::IIpFilter::getIpFamily ( ) [pure virtual]

Get the IP family type

Returns

telux::data::IpFamilyType

4.47.2.38 class telux::data::IUdpFilter

This class represents a IP Filter for the UDP, get the new instance from telux::data::DataFactory.

Public member functions

- virtual UdpInfo getUdpInfo ()=0
- virtual telux::common::Status setUdpInfo (const UdpInfo &udpInfo)=0
- virtual ~IUdpFilter ()

4.47.2.38.1 Constructors and Destructors

4.47.2.38.1.1 virtual telux::data::IUdpFilter::~IUdpFilter ( ) [virtual]

Destructor for IUdpFilter

4.47.2.38.2 Member Function Documentation

4.47.2.38.2.1 virtual UdpInfo telux::data::IUdpFilter::getUdpInfo ( ) [pure virtual]

Get the UDP header info

Returns

telux::data::UdpInfo

4.47.2.38.2.2 virtual telux::common::Status telux::data::IUdpFilter::setUdpInfo (const UdpInfo & *udpInfo*) [pure virtual]

sets the UDP header info

Parameters

in	*udpInfo*	UdpInfo structure telux::data::UdpInfo

Returns

Immediate status of setUdpInfo() request sent i.e. success or suitable status code.

4.47.2.39 class telux::data::ITcpFilter

This class represents a IP Filter for the TCP, get the new instance from telux::data::DataFactory.

Public member functions

- virtual TcpInfo getTcpInfo ()=0
- virtual telux::common::Status setTcpInfo (const TcpInfo &tcpInfo)=0
- virtual ~ITcpFilter ()

4.47.2.39.1 Constructors and Destructors

4.47.2.39.1.1 virtual telux::data::ITcpFilter::~ITcpFilter ( ) [virtual]

Destructor for ITcpFilter

4.47.2.39.2 Member Function Documentation

4.47.2.39.2.1 virtual TcpInfo telux::data::ITcpFilter::getTcpInfo ( ) [pure virtual]

Get the TCP header info

Returns

telux::data::TcpInfo

4.47.2.39.2.2 virtual telux::common::Status telux::data::ITcpFilter::setTcpInfo (const TcpInfo & *tcpInfo*) [pure virtual]

sets the TCP header info

Parameters

in	*tcpInfo*	TcpInfo structure telux::data::TcpInfo

Returns

Immediate status of setTcpInfo() request sent i.e. success or suitable status code.

4.47.2.40 class telux::data::IIcmpFilter

This class represents a IP Filter for the ICMP, get the new instance from telux::data::DataFactory.

Public member functions

- virtual IcmpInfo getIcmpInfo ()=0
- virtual telux::common::Status setIcmpInfo (const IcmpInfo &icmpInfo)=0

- virtual ~IIcmpFilter ()

4.47.2.40.1 Constructors and Destructors

4.47.2.40.1.1 virtual telux::data::IIcmpFilter::~IIcmpFilter ( ) [virtual]

Destructor for IIcmpFilter

4.47.2.40.2 Member Function Documentation

4.47.2.40.2.1 virtual IcmpInfo telux::data::IIcmpFilter::getIcmpInfo ( ) [pure virtual]

Get the ICMP header info

Returns

telux::data::IcmpInfo

4.47.2.40.2.2 virtual telux::common::Status telux::data::IIcmpFilter::setIcmpInfo (const IcmpInfo & *icmpInfo*) [pure virtual]

sets the ICMP header info

Parameters

in	*icmpInfo*	TcpInfo structure telux::data::IcmpInfo

Returns

Immediate status of setIcmpInfo() request sent i.e. success or suitable status code.

4.47.2.41 class telux::data::IEspFilter

This class represents a IP Filter for the ESP, get the new instance from telux::data::DataFactory.

Public member functions

- virtual EspInfo getEspInfo ()=0
- virtual telux::common::Status setEspInfo (const EspInfo &espInfo)=0
- virtual ~IEspFilter ()

4.47.2.41.1 Constructors and Destructors

4.47.2.41.1.1 virtual telux::data::IEspFilter::~IEspFilter ( ) [virtual]

Destructor for IEspFilter

4.47.2.41.2 Member Function Documentation

4.47.2.41.2.1 virtual EspInfo telux::data::IEspFilter::getEspInfo ( ) [pure virtual]

Get the ESP header info

Returns

telux::data::EspInfo

4.47.2.41.2.2 virtual telux::common::Status telux::data::IEspFilter::setEspInfo (const EspInfo & *espInfo*) [pure virtual]

sets the ICMP header info

Parameters

in	*espInfo*	EspInfo structure telux::data::EspInfo

Returns

Immediate status of setEspInfo() request sent i.e. success or suitable status code.

4.47.2.42 struct telux::data::RoamingStatus

Roaming Status.

Data fields

Type	Field	Description
bool	isRoaming	True: Roaming on, False: Roaming off
RoamingType	type	International/Domestic. Valid only if roaming is on

4.47.2.43 struct telux::data::ServiceStatus

Data Service Status Info.

Data fields

Type	Field	Description
DataService↩ State	serviceState	
NetworkRat	networkRat	

4.47.2.44 class telux::data::IServingSystemManager

Serving System Manager class provides APIs related to the serving system for data functionality. For example, ability to query or be notified about the state of the platform's WWAN PS data serving information.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual DrbStatus getDrbStatus ()=0
- virtual telux::common::Status requestServiceStatus (RequestServiceStatusResponseCb callback)=0
- virtual telux::common::Status requestRoamingStatus (RequestRoamingStatusResponseCb callback)=0
- virtual telux::common::Status requestNrIconType (RequestNrIconTypeResponseCb callback)=0
- virtual telux::common::Status makeDormant (telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status registerListener (std::weak_ptr< IServingSystemListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< IServingSystemListener > listener)=0
- virtual SlotId getSlotId ()=0
- virtual ~IServingSystemManager ()

4.47.2.44.1 Constructors and Destructors

4.47.2.44.1.1 virtual telux::data::IServingSystemManager::~IServingSystemManager ( ) [virtual]

Destructor of IServingSystemManager

4.47.2.44.2 Member Function Documentation

4.47.2.44.2.1 virtual telux::common::ServiceStatus telux::data::IServingSystemManager::getServiceStatus ( ) [pure virtual]

Checks the status of serving manager object and returns the result.

Returns

SERVICE_AVAILABLE - If serving manager object is ready for service.
SERVICE_UNAVAILABLE - If serving manager object is temporarily unavailable.
SERVICE_FAILED - If serving manager object encountered an irrecoverable failure.

4.47.2.44.2.2 virtual DrbStatus telux::data::IServingSystemManager::getDrbStatus ( ) [pure virtual]

Get the dedicated radio bearer (DRB) status

Returns

current DrbStatus DrbStatus.

4.47.2.44.2.3 virtual telux::common::Status telux::data::IServingSystemManager::requestServiceStatus (RequestServiceStatusResponseCb *callback*) [pure virtual]

Queries the current serving network status

Parameters

in	*callback*	callback to get response for requestServiceStatus

Returns

Status of requestServiceStatus i.e. success or suitable status code. if requestServiceStatus returns failure, callback will not be invoked.

4.47.2.44.2.4 virtual telux::common::Status telux::data::IServingSystemManager::requestRoaming↩ Status (RequestRoamingStatusResponseCb *callback*) [pure virtual]

Queries the current roaming status

Parameters

in	*callback*	callback to get response for requestRoamingStatus

Returns

Status of requestRoamingStatus i.e. success or suitable status code.

4.47.2.44.2.5 virtual telux::common::Status telux::data::IServingSystemManager::requestNrIconType (RequestNrIconTypeResponseCb *callback*) [pure virtual]

Queries the NR icon type to be displayed based on the serving system that the device has acquired service on.

Parameters

in	*callback*	callback to get response for requestNrIconType

Returns

Status of requestNrIconType i.e. success or suitable status code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.47.2.44.2.6 virtual telux::common::Status telux::data::IServingSystemManager::makeDormant (telux::common::ResponseCallback *callback* = `nullptr`) [pure virtual]

Request modem switch to dormant state: Certain network operations can only be performed when modem is in dormant state. This API provides an ability for clients to request modem to immediately transition to dormant state for such scenarios.

Clients must ensure no data calls are in process of bring up/tear down and there is no traffic on any active data calls when this API is called.

Parameters

in	*callback*	optional callback to get the response of makeDormancy

Returns

telux::common::ErrorCode::SUCCESS if request is honored by network.
telux::common::ErrorCode::INVALID_STATE is returned if:

- There is no active data calls
- Any Data calls is going through bring up/tear down
- There is data traffic on any active data calls If API fails, application is responsible for re-attempting operation at later time once the above conditions are met.

On platforms with Access control enabled, Caller needs to have TELUX_DATA_SERVICE_MGMT permission to invoke this API successfully.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.47.2.44.2.7 virtual telux::common::Status telux::data::IServingSystemManager::registerListener (std::weak_ptr< IServingSystemListener > *listener*) [pure virtual]

Register a listener for specific updates from serving system.

Parameters

in	*listener*	Pointer of IServingSystemListener object that processes the notification

Returns

Status of registerListener i.e success or suitable status code.

4.47.2.44.2.8 virtual telux::common::Status telux::data::IServingSystemManager::deregisterListener (std::weak_ptr< IServingSystemListener > *listener*) [pure virtual]

Deregister the previously added listener.

Parameters

in	*listener*	Previously registered IServingSystemListener that needs to be removed

Returns

Status of removeListener i.e. success or suitable status code

4.47.2.44.2.9 virtual SlotId telux::data::IServingSystemManager::getSlotId ( ) [pure virtual]

Get associated slot id for the Serving System Manager.

Returns

SlotId

4.47.2.45 class telux::data::IServingSystemListener

Listener class for data serving system change notification.

The listener method can be invoked from multiple different threads. Client needs to make sure that implementation is thread-safe.

Public member functions

- virtual void onServiceStatusChange (telux::common::ServiceStatus status)
- virtual void onDrbStatusChanged (DrbStatus status)
- virtual void onServiceStateChanged (ServiceStatus status)
- virtual void onRoamingStatusChanged (RoamingStatus status)
- virtual void onNrIconTypeChanged (NrIconType type)
- virtual ~IServingSystemListener ()

4.47.2.45.1 Constructors and Destructors

4.47.2.45.1.1 virtual telux::data::IServingSystemListener::~IServingSystemListener ( ) [virtual]

Destructor of IServingSystemListener

4.47.2.45.2 Member Function Documentation

4.47.2.45.2.1 virtual void telux::data::IServingSystemListener::onServiceStatusChange (telux::common::ServiceStatus *status*) [virtual]

This function is called when service status changes.

Parameters

in	*status*	- ServiceStatus

4.47.2.45.2.2 virtual void telux::data::IServingSystemListener::onDrbStatusChanged (DrbStatus *status*) [virtual]

This function is called whenever Drb status is changed.

Parameters

in	*status*	DrbStatus

4.47.2.45.2.3 virtual void telux::data::IServingSystemListener::onServiceStateChanged (ServiceStatus *status*) [virtual]

This function is called whenever service state is changed.

Parameters

in	*status*	ServiceStatus

4.47.2.45.2.4 virtual void telux::data::IServingSystemListener::onRoamingStatusChanged (RoamingStatus *status*) [virtual]

This function is called whenever roaming status is changed.

Parameters

in	*status*	RoamingStatus

4.47.2.45.2.5 virtual void telux::data::IServingSystemListener::onNrIconTypeChanged (NrIconType *type*) [virtual]

This function is called whenever NR icon type is changed.

Parameters

in	*type*	NrIconType

4.47.2.46 union telux::data::DataCallEndReason.__unnamed__

Data fields

Type	Field	Description
MobileIp↩ ReasonCode	IpCode	
Internal↩ ReasonCode	internalCode	
CallManager↩ ReasonCode	cmCode	
SpecReason↩ Code	specCode	
PPPReason↩ Code	pppCode	
EHRPD↩ ReasonCode	ehrpdCode	
Ipv6Reason↩ Code	ipv6Code	
Handoff↩ ReasonCode	handOffCode	

4.47.3 Enumeration Type Documentation

4.47.3.1 enum telux::data::IpFamilyType [strong]

Preferred IP family for the connection

Enumerator

UNKNOWN
IPV4 IPv4 data connection
IPV6 IPv6 data connection
IPV4V6 IPv4 and IPv6 data connection

4.47.3.2 enum telux::data::TechPreference [strong]

Technology Preference

Enumerator

UNKNOWN
TP_3GPP UMTS, LTE
TP_3GPP2 CDMA
TP_ANY ANY (3GPP or 3GPP2)

4.47.3.3 enum telux::data::AuthProtocolType [strong]

Authentication protocol preference type to be used for PDP context.

Enumerator

AUTH_NONE
AUTH_PAP Password Authentication Protocol
AUTH_CHAP Challenge Handshake Authentication Protocol
AUTH_PAP_CHAP

4.47.3.4 enum telux::data::DataRestrictModeType [strong]

Defines the supported filtering mode of the packet data session.

Enumerator

UNKNOWN
DISABLE
ENABLE

4.47.3.5 enum telux::data::ApnMaskType

Specifies APN types that can be set while creating or modifying a profile

Enumerator

APN_MASK_TYPE_DEFAULT APN type for default/internet traffic
APN_MASK_TYPE_IMS APN type for the IP multimedia subsystem
APN_MASK_TYPE_MMS APN type for the multimedia messaging service
APN_MASK_TYPE_DUN APN type for the dial up network
APN_MASK_TYPE_SUPL APN type for secure user plane location
APN_MASK_TYPE_HIPRI APN type for high priority mobile data
APN_MASK_TYPE_FOTA APN type for over the air administration
APN_MASK_TYPE_CBS APN type for carrier branded services
APN_MASK_TYPE_IA APN type for initial attach
APN_MASK_TYPE_EMERGENCY APN type for emergency
APN_MASK_TYPE_UT APN type for UT
APN_MASK_TYPE_MCX APN type for mission critical service

4.47.3.6 enum telux::data::DataCallStatus [strong]

Data call event status

Enumerator

INVALID Invalid
NET_CONNECTED Call is connected
NET_NO_NET Call is disconnected
NET_IDLE Call is in idle state
NET_CONNECTING Call is in connecting state
NET_DISCONNECTING Call is in disconnecting state
NET_RECONFIGURED Interface is reconfigured, IP Address got changed
NET_NEWADDR A new IP address was added on an existing call
NET_DELADDR An IP address was removed from the existing interface

4.47.3.7 enum telux::data::DataBearerTechnology [strong]

Bearer technology types (returned with getCurrentBearerTech).

Enumerator

UNKNOWN Unknown bearer.
CDMA_1X 1X technology.
EVDO_REV0 CDMA Rev 0.
EVDO_REVA CDMA Rev A.
EVDO_REVB CDMA Rev B.
EHRPD EHRPD.
FMC Fixed mobile convergence.
HRPD HRPD
BEARER_TECH_3GPP2_WLAN IWLAN
WCDMA WCDMA.
GPRS GPRS.
HSDPA HSDPA.
HSUPA HSUPA.
EDGE EDGE.
LTE LTE.
HSDPA_PLUS HSDPA+.
DC_HSDPA_PLUS DC HSDPA+.
HSPA HSPA
BEARER_TECH_64_QAM 64 QAM.
TDSCDMA TD-SCDMA.
GSM GSM
BEARER_TECH_3GPP_WLAN IWLAN
BEARER_TECH_5G 5G

4.47.3.8 enum telux::data::EndReasonType [strong]

Data call end/termination due to reason type.

Enumerator

CE_UNKNOWN
CE_MOBILE_IP
CE_INTERNAL
CE_CALL_MANAGER_DEFINED
CE_3GPP_SPEC_DEFINED
CE_PPP
CE_EHRPD
CE_IPV6
CE_HANDOFF

4.47.3.9 enum telux::data::MobileIpReasonCode [strong]

Data call end/termination reason code for EndReasonType::CE_MOBILE_IP

Enumerator

CE_MIP_FA_ERR_REASON_UNSPECIFIED
CE_MIP_FA_ERR_ADMINISTRATIVELY_PROHIBITED
CE_MIP_FA_ERR_INSUFFICIENT_RESOURCES
CE_MIP_FA_ERR_MOBILE_NODE_AUTHENTICATION_FAILURE
CE_MIP_FA_ERR_HA_AUTHENTICATION_FAILURE
CE_MIP_FA_ERR_REQUESTED_LIFETIME_TOO_LONG
CE_MIP_FA_ERR_MALFORMED_REQUEST
CE_MIP_FA_ERR_MALFORMED_REPLY
CE_MIP_FA_ERR_ENCAPSULATION_UNAVAILABLE
CE_MIP_FA_ERR_VJHC_UNAVAILABLE
CE_MIP_FA_ERR_REVERSE_TUNNEL_UNAVAILABLE
CE_MIP_FA_ERR_REVERSE_TUNNEL_IS_MANDATORY_AND_T_BIT_NOT_SET
CE_MIP_FA_ERR_DELIVERY_STYLE_NOT_SUPPORTED
CE_MIP_FA_ERR_MISSING_NAI
CE_MIP_FA_ERR_MISSING_HA
CE_MIP_FA_ERR_MISSING_HOME_ADDR
CE_MIP_FA_ERR_UNKNOWN_CHALLENGE
CE_MIP_FA_ERR_MISSING_CHALLENGE
CE_MIP_FA_ERR_STALE_CHALLENGE
CE_MIP_HA_ERR_REASON_UNSPECIFIED
CE_MIP_HA_ERR_ADMINISTRATIVELY_PROHIBITED
CE_MIP_HA_ERR_INSUFFICIENT_RESOURCES
CE_MIP_HA_ERR_MOBILE_NODE_AUTHENTICATION_FAILURE
CE_MIP_HA_ERR_FA_AUTHENTICATION_FAILURE
CE_MIP_HA_ERR_REGISTRATION_ID_MISMATCH
CE_MIP_HA_ERR_MALFORMED_REQUEST
CE_MIP_HA_ERR_UNKNOWN_HA_ADDR
CE_MIP_HA_ERR_REVERSE_TUNNEL_UNAVAILABLE
CE_MIP_HA_ERR_REVERSE_TUNNEL_IS_MANDATORY_AND_T_BIT_NOT_SET
CE_MIP_HA_ERR_ENCAPSULATION_UNAVAILABLE
CE_MIP_ERR_REASON_UNKNOWN

4.47.3.10 enum telux::data::InternalReasonCode `[strong]`

Data call end/termination reason code for EndReasonType::CE_INTERNAL

Enumerator

CE_RETRY
CE_INTERNAL_ERROR
CE_CALL_ENDED
CE_INTERNAL_UNKNOWN_CAUSE_CODE
CE_UNKNOWN_CAUSE_CODE
CE_CLOSE_IN_PROGRESS
CE_NW_INITIATED_TERMINATION
CE_APP_PREEMPTED
CE_ERR_PDN_IPV4_CALL_DISALLOWED
CE_ERR_PDN_IPV4_CALL_THROTTLED
CE_ERR_PDN_IPV6_CALL_DISALLOWED

CE_ERR_PDN_IPV6_CALL_THROTTLED
CE_MODEM_RESTART
CE_PDP_PPP_NOT_SUPPORTED
CE_UNPREFERRED_RAT
CE_PHYS_LINK_CLOSE_IN_PROGRESS
CE_APN_PENDING_HANDOVER
CE_PROFILE_BEARER_INCOMPATIBLE
CE_MMGSDI_CARD_EVT
CE_LPM_OR_PWR_DOWN
CE_APN_DISABLED
CE_MPIT_EXPIRED
CE_IPV6_ADDR_TRANSFER_FAILED
CE_TRAT_SWAP_FAILED
CE_EHRPD_TO_HRPD_FALLBACK
CE_MANDATORY_APN_DISABLED
CE_MIP_CONFIG_FAILURE
CE_INTERNAL_PDN_INACTIVITY_TIMER_EXPIRED
CE_MAX_V4_CONNECTIONS
CE_MAX_V6_CONNECTIONS
CE_APN_MISMATCH
CE_IP_VERSION_MISMATCH
CE_DUN_CALL_DISALLOWED
CE_INVALID_PROFILE
CE_INTERNAL_EPC_NONEPC_TRANSITION
CE_INVALID_PROFILE_ID
CE_INTERNAL_CALL_ALREADY_PRESENT
CE_IFACE_IN_USE
CE_IP_PDP_MISMATCH
CE_APN_DISALLOWED_ON_ROAMING
CE_APN_PARAM_CHANGE
CE_IFACE_IN_USE_CFG_MATCH
CE_NULL_APN_DISALLOWED
CE_THERMAL_MITIGATION
CE_SUBS_ID_MISMATCH
CE_DATA_SETTINGS_DISABLED
CE_DATA_ROAMING_SETTINGS_DISABLED
CE_APN_FORMAT_INVALID
CE_DDS_CALL_ABORT
CE_VALIDATION_FAILURE
CE_PROFILES_NOT_COMPATIBLE
CE_NULL_RESOLVED_APN_NO_MATCH
CE_INVALID_APN_NAME

4.47.3.11 enum telux::data::CallManagerReasonCode [strong]

Data call end/termination reason code for EndReasonType::CE_CALL_MANAGER_DEFINED

Enumerator

CE_CDMA_LOCK

CE_INTERCEPT
CE_REORDER
CE_REL_SO_REJ
CE_INCOM_CALL
CE_ALERT_STOP
CE_ACTIVATION
CE_MAX_ACCESS_PROBE
CE_CCS_NOT_SUPPORTED_BY_BS
CE_NO_RESPONSE_FROM_BS
CE_REJECTED_BY_BS
CE_INCOMPATIBLE
CE_ALREADY_IN_TC
CE_USER_CALL_ORIG_DURING_GPS
CE_USER_CALL_ORIG_DURING_SMS
CE_NO_CDMA_SRV
CE_MC_ABORT
CE_PSIST_NG
CE_UIM_NOT_PRESENT
CE_RETRY_ORDER
CE_ACCESS_BLOCK
CEACCESS_BLOCK_ALL
CE_IS707B_MAX_ACC
CE_THERMAL_EMERGENCY
CE_CALL_ORIG_THROTTLED
CE_USER_CALL_ORIG_DURING_VOICE_CALL
CE_CONF_FAILED
CE_INCOM_REJ
CE_NEW_NO_GW_SRV
CE_NEW_NO_GPRS_CONTEXT
CE_NEW_ILLEGAL_MS
CE_NEW_ILLEGAL_ME
CE_NEW_GPRS_SERVICES_AND_NON_GPRS_SERVICES_NOT_ALLOWED
CE_NEW_GPRS_SERVICES_NOT_ALLOWED
CE_NEW_MS_IDENTITY_CANNOT_BE_DERIVED_BY_THE_NETWORK
CE_NEW_IMPLICITLY_DETACHED
CE_NEW_PLMN_NOT_ALLOWED
CE_NEW_LA_NOT_ALLOWED
CE_NEW_GPRS_SERVICES_NOT_ALLOWED_IN_THIS_PLMN
CE_NEW_PDP_DUPLICATE
CE_NEW_UE_RAT_CHANGE
CE_NEW_CONGESTION
CE_NEW_NO_PDP_CONTEXT_ACTIVATED
CE_NEW_ACCESS_CLASS_DSAC_REJECTION
CE_PDP_ACTIVATE_MAX_RETRY_FAILED
CE_RAB_FAILURE
CE_ESM_UNKNOWN_EPS_BEARER_CONTEXT
CE_DRB_RELEASED_AT_RRC
CE_NAS_SIG_CONN_RELEASED
CE_REASON_EMM_DETACHED
CE_EMM_ATTACH_FAILED

CE_EMM_ATTACH_STARTED
CE_LTE_NAS_SERVICE_REQ_FAILED
CE_ESM_ACTIVE_DEDICATED_BEARER_REACTIVATED_BY_NW
CE_ESM_LOWER_LAYER_FAILURE
CE_ESM_SYNC_UP_WITH_NW
CE_ESM_NW_ACTIVATED_DED_BEARER_WITH_ID_OF_DEF_BEARER
CE_ESM_BAD_OTA_MESSAGE
CE_ESM_DS_REJECTED_THE_CALL
CE_ESM_CONTEXT_TRANSFERED_DUE_TO_IRAT
CE_DS_EXPLICIT_DEACT
CE_ESM_LOCAL_CAUSE_NONE
CE_LTE_NAS_SERVICE_REQ_FAILED_NO_THROTTLE
CE_ACL_FAILURE
CE_LTE_NAS_SERVICE_REQ_FAILED_DS_DISALLOW
CE_EMM_T3417_EXPIRED
CE_EMM_T3417_EXT_EXPIRED
CE_LRRC_UL_DATA_CNF_FAILURE_TXN
CE_LRRC_UL_DATA_CNF_FAILURE_HO
CE_LRRC_UL_DATA_CNF_FAILURE_CONN_REL
CE_LRRC_UL_DATA_CNF_FAILURE_RLF
CE_LRRC_UL_DATA_CNF_FAILURE_CTRL_NOT_CONN
CE_LRRC_CONN_EST_FAILURE
CE_LRRC_CONN_EST_FAILURE_ABORTED
CE_LRRC_CONN_EST_FAILURE_ACCESS_BARRED
CE_LRRC_CONN_EST_FAILURE_CELL_RESEL
CE_LRRC_CONN_EST_FAILURE_CONFIG_FAILURE
CE_LRRC_CONN_EST_FAILURE_TIMER_EXPIRED
CE_LRRC_CONN_EST_FAILURE_LINK_FAILURE
CE_LRRC_CONN_EST_FAILURE_NOT_CAMPED
CE_LRRC_CONN_EST_FAILURE_SI_FAILURE
CE_LRRC_CONN_EST_FAILURE_CONN_REJECT
CE_LRRC_CONN_REL_NORMAL
CE_LRRC_CONN_REL_RLF
CE_LRRC_CONN_REL_CRE_FAILURE
CE_LRRC_CONN_REL_OOS_DURING_CRE
CE_LRRC_CONN_REL_ABORTED
CE_LRRC_CONN_REL_SIB_READ_ERROR
CE_DETACH_WITH_REATTACH_LTE_NW_DETACH
CE_DETACH_WITH_OUT_REATTACH_LTE_NW_DETACH
CE_ESM_PROC_TIME_OUT
CE_INVALID_CONNECTION_ID
CE_INVALID_NSAPI
CE_INVALID_PRI_NSAPI
CE_INVALID_FIELD
CE_RAB_SETUP_FAILURE
CE_PDP_ESTABLISH_MAX_TIMEOUT
CE_PDP_MODIFY_MAX_TIMEOUT
CE_PDP_INACTIVE_MAX_TIMEOUT
CE_PDP_LOWERLAYER_ERROR
CE_PPD_UNKNOWN_REASON

CE_PDP_MODIFY_COLLISION
CE_PDP_MBMS_REQUEST_COLLISION
CE_MBMS_DUPLICATE
CE_SM_PS_DETACHED
CE_SM_NO_RADIO_AVAILABLE
CE_SM_ABORT_SERVICE_NOT_AVAILABLE
CE_MESSAGE_EXCEED_MAX_L2_LIMIT
CE_SM_NAS_SRV_REQ_FAILURE
CE_RRC_CONN_EST_FAILURE_REQ_ERROR
CE_RRC_CONN_EST_FAILURE_TAI_CHANGE
CE_RRC_CONN_EST_FAILURE_RF_UNAVAILABLE
CE_RRC_CONN_REL_ABORTED_IRAT_SUCCESS
CE_RRC_CONN_REL_RLF_SEC_NOT_ACTIVE
CE_RRC_CONN_REL_IRAT_TO_LTE_ABORTED
CE_RRC_CONN_REL_IRAT_FROM_LTE_TO_G_CCO_SUCCESS
CE_RRC_CONN_REL_IRAT_FROM_LTE_TO_G_CCO_ABORTED
CE_IMSI_UNKNOWN_IN_HSS
CE_IMEI_NOT_ACCEPTED
CE_EPS_SERVICES_AND_NON_EPS_SERVICES_NOT_ALLOWED
CE_EPS_SERVICES_NOT_ALLOWED_IN_PLMN
CE_MSC_TEMPORARILY_NOT_REACHABLE
CE_CS_DOMAIN_NOT_AVAILABLE
CE_ESM_FAILURE
CE_MAC_FAILURE
CE_SYNCH_FAILURE
CE_UE_SECURITY_CAPABILITIES_MISMATCH
CE_SECURITY_MODE_REJ_UNSPECIFIED
CE_NON_EPS_AUTH_UNACCEPTABLE
CE_CS_FALLBACK_CALL_EST_NOT_ALLOWED
CE_NO_EPS_BEARER_CONTEXT_ACTIVATED
CE_EMM_INVALID_STATE
CE_NAS_LAYER_FAILURE
CE_MULTI_PDN_NOT_ALLOWED
CE_EMBMS_NOT_ENABLED
CE_PENDING_REDIAL_CALL_CLEANUP
CE_EMBMS_REGULAR_DEACTIVATION
CE_TLB_REGULAR_DEACTIVATION
CE_LOWER_LAYER_REGISTRATION_FAILURE
CE_DETACH_EPS_SERVICES_NOT_ALLOWED
CE_SM_INTERNAL_PDP_DEACTIVATION
CE_UNSUPPORTED_1X_PREV
CE_CD_GEN_OR_BUSY
CE_CD_BILL_OR_AUTH
CE_CHG_HDR
CE_EXIT_HDR
CE_HDR_NO_SESSION
CE_HDR_ORIG_DURING_GPS_FIX
CE_HDR_CS_TIMEOUT
CE_HDR_RELEASED_BY_CM
CE_COLLOC_ACQ_FAIL

CE_OTASP_COMMIT_IN_PROG
CE_NO_HYBR_HDR_SRV
CE_HDR_NO_LOCK_GRANTED
CE_HOLD_OTHER_IN_PROG
CE_HDR_FADE
CE_HDR_ACC_FAIL
CE_CLIENT_END
CE_NO_SRV
CE_FADE
CE_REL_NORMAL
CE_ACC_IN_PROG
CE_ACC_FAIL
CE_REDIR_OR_HANDOFF
CE_CM_UNKNOWN_ERROR
CE_OFFLINE
CE_EMERGENCY_MODE
CE_PHONE_IN_USE
CE_INVALID_MODE
CE_INVALID_SIM_STATE
CE_NO_COLLOC_HDR
CE_CALL_CONTROL_REJECTED
CE_UNKNOWN

4.47.3.12 enum telux::data::SpecReasonCode `[strong]`

Data call end/termination reason code for EndReasonType::CE_3GPP_SPEC_DEFINED

Enumerator

CE_OPERATOR_DETERMINED_BARRING
CE_NAS_SIGNALLING_ERROR
CE_LLC_SNDCP_FAILURE
CE_INSUFFICIENT_RESOURCES
CE_UNKNOWN_APN
CE_UNKNOWN_PDP
CE_AUTH_FAILED
CE_GGSN_REJECT
CE_ACTIVATION_REJECT
CE_OPTION_NOT_SUPPORTED
CE_OPTION_UNSUBSCRIBED
CE_OPTION_TEMP_OOO
CE_NSAPI_ALREADY_USED
CE_REGULAR_DEACTIVATION
CE_QOS_NOT_ACCEPTED
CE_NETWORK_FAILURE
CE_UMTS_REACTIVATION_REQ
CE_FEATURE_NOT_SUPPORTED
CE_TFT_SEMANTIC_ERROR
CE_TFT_SYNTAX_ERROR
CE_UNKNOWN_PDP_CONTEXT

CE_FILTER_SEMANTIC_ERROR
CE_FILTER_SYNTAX_ERROR
CE_PDP_WITHOUT_ACTIVE_TFT
CE_IP_V4_ONLY_ALLOWED
CE_IP_V6_ONLY_ALLOWED
CE_SINGLE_ADDR_BEARER_ONLY
CE_ESM_INFO_NOT_RECEIVED
CE_PDN_CONN_DOES_NOT_EXIST
CE_MULTI_CONN_TO_SAME_PDN_NOT_ALLOWED
CE_MAX_ACTIVE_PDP_CONTEXT_REACHED
CE_UNSUPPORTED_APN_IN_CURRENT_PLMN
CE_INVALID_TRANSACTION_ID
CE_MESSAGE_INCORRECT_SEMANTIC
CE_INVALID_MANDATORY_INFO
CE_MESSAGE_TYPE_UNSUPPORTED
CE_MSG_TYPE_NONCOMPATIBLE_STATE
CE_UNKNOWN_INFO_ELEMENT
CE_CONDITIONAL_IE_ERROR
CE_MSG_AND_PROTOCOL_STATE_UNCOMPATIBLE
CE_PROTOCOL_ERROR
CE_APN_TYPE_CONFLICT
CE_INVALID_PCSCF_ADDRESS
CE_INTERNAL_CALL_PREEMPT_BY_HIGH_PRIO_APN
CE_EMM_ACCESS_BARRED
CE_EMERGENCY_IFACE_ONLY
CE_IFACE_MISMATCH
CE_COMPANION_IFACE_IN_USE
CE_IP_ADDRESS_MISMATCH
CE_IFACE_AND_POL_FAMILY_MISMATCH
CE_EMM_ACCESS_BARRED_INFINITE_RETRY
CE_AUTH_FAILURE_ON_EMERGENCY_CALL
CE_INVALID_DNS_ADDR
CE_INVALID_PCSCF_DNS_ADDR
CE_TEST_LOOPBACK_MODE_A_OR_B_ENABLED
CE_UNKNOWN

4.47.3.13 enum telux::data::PPPReasonCode `[strong]`

Data call end/termination reason code for EndReasonType::CE_PPP

Enumerator

CE_PPP_TIMEOUT
CE_PPP_AUTH_FAILURE
CE_PPP_OPTION_MISMATCH
CE_PPP_PAP_FAILURE
CE_PPP_CHAP_FAILURE
CE_PPP_CLOSE_IN_PROGRESS
CE_PPP_NV_REFRESH_IN_PROGRESS
CE_PPP_UNKNOWN

4.47.3.14 enum telux::data::EHRPDReasonCode [strong]

Data call end/termination reason code for EndReasonType::CE_EHRPD

Enumerator

CE_EHRPD_SUBS_LIMITED_TO_V4
CE_EHRPD_SUBS_LIMITED_TO_V6
CE_EHRPD_VSNCP_TIMEOUT
CE_EHRPD_VSNCP_FAILURE
CE_EHRPD_VSNCP_3GPP2I_GEN_ERROR
CE_EHRPD_VSNCP_3GPP2I_UNAUTH_APN
CE_EHRPD_VSNCP_3GPP2I_PDN_LIMIT_EXCEED
CE_EHRPD_VSNCP_3GPP2I_NO_PDN_GW
CE_EHRPD_VSNCP_3GPP2I_PDN_GW_UNREACH
CE_EHRPD_VSNCP_3GPP2I_PDN_GW_REJ
CE_EHRPD_VSNCP_3GPP2I_INSUFF_PARAM
CE_EHRPD_VSNCP_3GPP2I_RESOURCE_UNAVAIL
CE_EHRPD_VSNCP_3GPP2I_ADMIN_PROHIBIT
CE_EHRPD_VSNCP_3GPP2I_PDN_ID_IN_USE
CE_EHRPD_VSNCP_3GPP2I_SUBSCR_LIMITATION
CE_EHRPD_VSNCP_3GPP2I_PDN_EXISTS_FOR_THIS_APN
CE_EHRPD_VSNCP_3GPP2I_RECONNECT_NOT_ALLOWED
CE_EHRPD_UNKNOWN

4.47.3.15 enum telux::data::Ipv6ReasonCode [strong]

Data call end/termination reason code for EndReasonType::CE_IPV6

Enumerator

CE_PREFIX_UNAVAILABLE
CE_IPV6_ERR_HRPD_IPV6_DISABLED
CE_IPV6_DISABLED

4.47.3.16 enum telux::data::HandoffReasonCode [strong]

Data call end/termination reason code for EndReasonType::CE_HANDOFF

Enumerator

CE_VCER_HANDOFF_PREF_SYS_BACK_TO_SRAT

4.47.3.17 enum telux::data::ProfileChangeEvent [strong]

Event due to which change in profile happened.

Enumerator

CREATE_PROFILE_EVENT Profile was created
DELETE_PROFILE_EVENT Profile was deleted
MODIFY_PROFILE_EVENT Profile was modified

4.47.3.18 enum telux::data::OperationType [strong]

This applies in architectures where the modem is attached to an External Application Processor(EAP). An API, like start/stop data call, INatManager, IFirewallManager can be invoked from the EAP or from the modems Internal Application Processor (IAP). This type specifies where the operation should be carried out.

Enumerator

DATA_LOCAL Perform the operation on the processor where the API is invoked.
DATA_REMOTE Perform the operation on the application processor other than where the API is invoked.

4.47.3.19 enum telux::data::Direction [strong]

Direction of firewall rule

Enumerator

UPLINK Uplink Direction
DOWNLINK Downlink Direction

4.47.3.20 enum telux::data::InterfaceType [strong]

Peripheral Interface type

Enumerator

UNKNOWN UNKNOWN interface
WLAN Wireless Local Area Network (WLAN)
ETH Ethernet (ETH)
ECM Ethernet Control Model (ECM)
RNDIS Remote Network Driver Interface Specification (RNDIS)
MHI Modem Host Interface (MHI)
VMTAP0 Virtual interface for VM 1
VMTAP1 Virtual interface for VM 2

4.47.3.21 enum telux::data::ServiceState [strong]

State of Service

Enumerator

INACTIVE Service is inactive
ACTIVE Service is Active

4.47.3.22 enum telux::data::QosFlowStateChangeEvent [strong]

QOS flow state change type

Enumerator

UNKNOWN UNKNOWN state
ACTIVATED Flow activated
MODIFIED Flow modified
DELETED Flow deleted

4.47.3.23 enum telux::data::IpTrafficClassType [strong]

QOS flow IP traffic class type

Enumerator

UNKNOWN UNKNOWN type
CONVERSATIONAL Conversational IP Traffic class
STREAMING Streaming IP Traffic class
INTERACTIVE Interactive IP Traffic class
BACKGROUND Background IP Traffic class

4.47.3.24 enum telux::data::QosIPFlowMaskType

Specifies QOS IP Flow parameter mask

Enumerator

MASK_IP_FLOW_NONE
MASK_IP_FLOW_TRF_CLASS No parameters set
MASK_IP_FLOW_DATA_RATE_MIN_MAX Traffic class

4.47.3.25 enum telux::data::QosFlowMaskType

Specifies QOS Flow parameter mask

Enumerator

MASK_FLOW_NONE
MASK_FLOW_TX_GRANTED No parameters set
MASK_FLOW_RX_GRANTED TX Granted flow set
MASK_FLOW_TX_FILTERS RX Granted flow set
MASK_FLOW_RX_FILTERS TX filters set

4.47.3.26 enum telux::data::BackhaulType [strong]

Specifies backhaul types

Enumerator

ETH

USB Ethernet Backhaul
WLAN USB Backhaul
WWAN WLAN Backhaul
BLE WWAN Backhaul with default profile ID set by telux::data::IDataConnectionManager::setDefaultProfile
MAX_SUPPORTED Bluetooth Backhaul

4.47.3.27 enum telux::data::BandPriority [strong]

Set priority between N79 5G and Wlan 5GHz Band

Enumerator

N79
WLAN N79 has higher priority

4.47.3.28 enum telux::data::DdsType [strong]

Possible DDS switch types.

Enumerator

PERMANENT
TEMPORARY Permanently switch the DDS SIM Slot. Intended to be used when the client wants to stop data activities on the current DDS SIM slot and start doing data activities on the other SIM slot, on a Dual SIM Dual Standby (DSDS) device. Permanent switch is persistent across reboots.

4.47.3.29 enum telux::data::DrbStatus [strong]

Dedicated Radio Bearer (DRB) status.

Enumerator

ACTIVE At least one of the physical links across all PDNs is UP
DORMANT All the Physlinks across all PDNs are DOWN
UNKNOWN No PDN is active

4.47.3.30 enum telux::data::RoamingType [strong]

Roaming Type.

Enumerator

UNKNOWN Device roaming mode is unknown
DOMESTIC Device is in Domestic roaming network
INTERNATIONAL Device is in International roaming network

4.47.3.31 enum telux::data::DataServiceState [strong]

Data Service State. Indicates whether data service is ready to setup a data call or not.

Enumerator

UNKNOWN Service State not available
IN_SERVICE Service Available
OUT_OF_SERVICE Service Not Available

4.47.3.32 enum telux::data::NetworkRat [strong]

Data Network RATs.

Enumerator

UNKNOWN UNKNOWN
CDMA_1X CDMA_1X
CDMA_EVDO CDMA_EVDO
GSM GSM
WCDMA WCDMA
LTE LTE
TDSCDMA TDSCDMA
NR5G NR5G

4.47.3.33 enum telux::data::NrIconType [strong]

NR icon type.

Enumerator

NONE Unspecified
BASIC 5G basic
UWB 5G ultrawide band

4.48 net

This section contains APIs related to data network configuration.

4.48.1 Data Structure Documentation

4.48.1.1 struct telux::data::net::BridgeInfo

Structure to configure a software bridge for an interface

Data fields

Type	Field	Description
string	ifaceName	Interface name
BridgeIFace↩ Type	ifaceType	Interface type
uint32_t	bandwidth	Bandwidth(in Mbps) required for software bridge

4.48.1.2 class telux::data::net::IBridgeManager

IBridgeManager provides APIs to enable/disable and set/get/delete software bridges for various WLAN and Ethernet interfaces. It also provides interface to Subsystem Restart events by registering as listener. Notifications will be received when modem is ready/not ready.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual bool isSubsystemReady ()=0
- virtual std::future< bool > onSubsystemReady ()=0
- virtual telux::common::Status enableBridge (bool enable, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status addBridge (BridgeInfo config, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestBridgeInfo (BridgeInfoResponseCb callback)=0
- virtual telux::common::Status removeBridge (std::string ifaceName, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status registerListener (std::weak_ptr< IBridgeListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< IBridgeListener > listener)=0
- virtual ~IBridgeManager ()

4.48.1.2.1 Constructors and Destructors

4.48.1.2.1.1 virtual telux::data::net::IBridgeManager::~IBridgeManager ( ) [virtual]

Destructor for IBridgeManager

4.48.1.2.2 Member Function Documentation

4.48.1.2.2.1 virtual telux::common::ServiceStatus telux::data::net::IBridgeManager::getServiceStatus ( ) [pure virtual]

Checks the status of Bridge manager and returns the result.

Returns

SERVICE_AVAILABLE If Bridge manager object is ready for service. SERVICE_UNAVAILABLE If Bridge manager object is temporarily unavailable. SERVICE_FAILED If Bridge manager object encountered an irrecoverable failure.

4.48.1.2.2.2 virtual bool telux::data::net::IBridgeManager::isSubsystemReady ( ) [pure virtual]

Checks if the Bridge manager subsystem is ready.

Returns

True if the Bridge Manager is ready for service, otherwise returns false.

Deprecated

Use getServiceStatus API.

4.48.1.2.2.3 virtual std::future<bool> telux::data::net::IBridgeManager::onSubsystemReady ( ) [pure virtual]

Wait for Bridge manager subsystem to be ready.

Returns

A future that caller can wait until the Bridge Manager succeed/fail to be ready.

Deprecated

Use InitResponseCb callback in factory API getBridgeManager.

4.48.1.2.2.4 virtual telux::common::Status telux::data::net::IBridgeManager::enableBridge (bool *enable,* telux::common::ResponseCallback *callback* = `nullptr`) [pure virtual]

Enable/Disable the software bridge in the system. It will affect all the configured software bridges for various interfaces.

On platforms with Access control enabled, Caller needs to have TELUX_DATA_NETWORK_CONFIG

permission to invoke this API successfully.

Parameters

in	*enable*	TRUE to enable, FALSE to disable the bridge
in	*callback*	Optional callback to get the response for enableBridge

Returns

Status of enableBridge request i.e. success or suitable status code.

4.48.1.2.2.5 virtual telux::common::Status telux::data::net::IBridgeManager::addBridge (BridgeInfo *config,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

Add software bridge configuration for an interface.

On platforms with Access control enabled, Caller needs to have TELUX_DATA_NETWORK_CONFIG permission to invoke this API successfully.

Parameters

in	*config*	configuration for an interface
in	*callback*	Optional callback to get the response for addBridge

Returns

Status of addBridge request i.e. success or suitable status code.

4.48.1.2.2.6 virtual telux::common::Status telux::data::net::IBridgeManager::requestBridgeInfo (BridgeInfoResponseCb *callback*) [pure virtual]

Request information about all the software bridge configurations in the system

Parameters

in	*callback*	Response callback with list of bridge configurations

Returns

Status of requestBridgeInfo request i.e. success or suitable status code.

4.48.1.2.2.7 virtual telux::common::Status telux::data::net::IBridgeManager::removeBridge (std::string *ifaceName,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

Delete a software bridge configuration for an interface.

On platforms with Access control enabled, Caller needs to have TELUX_DATA_NETWORK_CONFIG permission to invoke this API successfully.

Parameters

in	*ifaceName*	Name of the interface whose configuration needs to be deleted
in	*callback*	Optional callback to get the response for removeBridge

Returns

Status of removeBridge request i.e. success or suitable status code.

4.48.1.2.2.8 virtual telux::common::Status telux::data::net::IBridgeManager::registerListener (std::weak_ptr< IBridgeListener > *listener*) [pure virtual]

Register Bridge Manager as listener for Data Service heath events like data service available or data service not available.

Parameters

in	*listener*	pointer of IBridgeListener object that processes the notification

Returns

Status of registerListener success or suitable status code

4.48.1.2.2.9 virtual telux::common::Status telux::data::net::IBridgeManager::deregisterListener (std::weak_ptr< IBridgeListener > *listener*) [pure virtual]

Removes a previously added listener.

Parameters

in	*listener*	pointer of IBridgeListener object that needs to be removed

Returns

Status of deregisterListener success or suitable status code

4.48.1.3 class telux::data::net::IBridgeListener

Interface for Bridge listener object. Client needs to implement this interface to get access to Bridge services notifications like onServiceStatusChange.

The methods in listener can be invoked from multiple different threads. The implementation should be thread safe.

Public member functions

- virtual void onServiceStatusChange (telux::common::ServiceStatus status)
- virtual ~IBridgeListener ()

4.48.1.3.1 Constructors and Destructors

4.48.1.3.1.1 virtual telux::data::net::IBridgeListener::~IBridgeListener ( ) [virtual]

Destructor for IBridgeListener

4.48.1.3.2 Member Function Documentation

4.48.1.3.2.1 virtual void telux::data::net::IBridgeListener::onServiceStatusChange (telux::common::↩ ServiceStatus *status*) [virtual]

This function is called when service status changes.

Parameters

in	*status*	- ServiceStatus

4.48.1.4 class telux::data::net::IFirewallManager

FirewallManager is a primary interface that filters and controls the network traffic on a pre-configured set of rules. It also provides interface to Subsystem Restart events by registering as listener. Notifications will be received when modem is ready/not ready.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual bool isSubsystemReady ()=0
- virtual std::future< bool > onSubsystemReady ()=0
- virtual telux::common::Status setFirewall (int profileId, bool enable, bool allowPackets, telux::common::ResponseCallback callback=nullptr, SlotId slotId=DEFAULT_SLOT_ID)=0
- virtual telux::common::Status requestFirewallStatus (int profileId, FirewallStatusCb callback, SlotId slotId=DEFAULT_SLOT_ID)=0
- virtual telux::common::Status addFirewallEntry (int profileId, std::shared_ptr< IFirewallEntry > entry, telux::common::ResponseCallback callback=nullptr, SlotId slotId=DEFAULT_SLOT_ID)=0
- virtual telux::common::Status addHwAccelerationFirewallEntry (int profileId, std::shared_ptr< IFirewallEntry > entry, AddFirewallEntryCb callback=nullptr, SlotId slotId=DEFAULT_SLOT_ID)=0
- virtual telux::common::Status requestHwAccelerationFirewallEntries (int profileId, FirewallEntriesCb callback, SlotId slotId=DEFAULT_SLOT_ID)=0
- virtual telux::common::Status requestFirewallEntries (int profileId, FirewallEntriesCb callback, SlotId slotId=DEFAULT_SLOT_ID)=0
- virtual telux::common::Status removeFirewallEntry (int profileId, uint32_t handle, telux::common::ResponseCallback callback=nullptr, SlotId slotId=DEFAULT_SLOT_ID)=0
- virtual telux::common::Status enableDmz (int profileId, const std::string ipAddr,

telux::common::ResponseCallback callback=nullptr, SlotId slotId=DEFAULT_SLOT_ID)=0

- virtual telux::common::Status disableDmz (int profileId, const telux::data::IpFamilyType ipType, telux::common::ResponseCallback callback=nullptr, SlotId slotId=DEFAULT_SLOT_ID)=0
- virtual telux::common::Status requestDmzEntry (int profileId, DmzEntriesCb dmzCb, SlotId slotId=DEFAULT_SLOT_ID)=0
- virtual telux::common::Status registerListener (std::weak_ptr< IFirewallListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< IFirewallListener > listener)=0
- virtual telux::data::OperationType getOperationType ()=0
- virtual ~IFirewallManager ()

4.48.1.4.1 Constructors and Destructors

4.48.1.4.1.1 virtual telux::data::net::IFirewallManager::~IFirewallManager ( ) [virtual]

Destructor for IFirewallManager

4.48.1.4.2 Member Function Documentation

4.48.1.4.2.1 virtual telux::common::ServiceStatus telux::data::net::IFirewallManager::getServiceStatus ( ) [pure virtual]

Checks the status of Firewall manager and returns the result.

Returns

SERVICE_AVAILABLE If Firewall manager object is ready for service.
SERVICE_UNAVAILABLE If Firewall manager object is temporarily unavailable.
SERVICE_FAILED If Firewall manager object encountered an irrecoverable failure.

4.48.1.4.2.2 virtual bool telux::data::net::IFirewallManager::isSubsystemReady ( ) [pure virtual]

Checks if the Firewall manager subsystem is ready.

Returns

True if Firewall Manager is ready for service, otherwise returns false.

Deprecated

Use getServiceStatus API.

4.48.1.4.2.3 virtual std::future<bool> telux::data::net::IFirewallManager::onSubsystemReady ( ) [pure virtual]

Wait for Firewall manager subsystem to be ready.

Returns

A future that caller can wait on to be notified when firewall manager is ready.

Deprecated

Use InitResponseCb callback in factory API getFirewallManager.

4.48.1.4.2.4 virtual telux::common::Status telux::data::net::IFirewallManager::setFirewall (int *profileId,* bool *enable,* bool *allowPackets,* telux::common::ResponseCallback *callback = nullptr,* SlotId *slotId = DEFAULT_SLOT_ID*) [pure virtual]

Sets firewall configuration to enable or disable and update configuration to drop or accept the packets matching the rules.

On platforms with Access control enabled, Caller needs to have TELUX_DATA_NETWORK_CONFIG permission to invoke this API successfully.

Parameters

in	*profileId*	Profile identifier on which firewall will be set.
in	*enable*	Indicates whether the firewall is enabled
in	*allowPackets*	Indicates whether to accept or drop packets matching the rules
in	*callback*	optional callback to get the response setFirewall
in	*slotId*	Specify slot id which has the sim that contains profile id

Returns

Status of setFirewall i.e. success or suitable status code.

4.48.1.4.2.5 virtual telux::common::Status telux::data::net::IFirewallManager::requestFirewallStatus (int *profileId,* FirewallStatusCb *callback,* SlotId *slotId = DEFAULT_SLOT_ID*) [pure virtual]

Request status of firewall

Parameters

in	*profileId*	Profile identifier for which firewall status is requested.
in	*callback*	callback to get the response of requestFirewallStatus
in	*slotId*	Specify slot id which has the sim that contains profile id

Returns

Status of requestFirewallStatus i.e. success or suitable status code.

4.48.1.4.2.6 virtual telux::common::Status telux::data::net::IFirewallManager::addFirewallEntry (int *profileId*, std::shared_ptr< IFirewallEntry > *entry*, telux::common::ResponseCallback *callback = nullptr*, SlotId *slotId = DEFAULT_SLOT_ID*) [pure virtual]

Adds the firewall rule

On platforms with Access control enabled, Caller needs to have TELUX_DATA_NETWORK_CONFIG permission to invoke this API successfully.

Parameters

in	*profileId*	Profile identifier on which firewall rule will be added.
in	*entry*	Firewall entry based on protocol type
in	*callback*	optional callback to get the response addFirewallEntry
in	*slotId*	Specify slot id which has the sim that contains profile id

Returns

Status of addFirewallEntry i.e. success or suitable status code.

4.48.1.4.2.7 virtual telux::common::Status telux::data::net::IFirewallManager::addHwAcceleration↩ FirewallEntry (int *profileId*, std::shared_ptr< IFirewallEntry > *entry*, AddFirewallEntryCb *callback = nullptr*, SlotId *slotId = DEFAULT_SLOT_ID*) [pure virtual]

Add Hardware Acceleration Rule Adds a firewall rule which will direct all traffic that matches the rule to bypass hardware acceleration, and take the software path.

These rules are per PDN. If the same rule applies to more than one PDN then this API needs to be invoked per PDN by specifying the corresponding profile ID of the PDN. setFirewall is not required for hw acceleration firewall rules to have an effect, which means that as soon as the rule is added successfully, packets matching the firewall rule will not be hw accelerated. Irrespective of whether firewall rules are set via addFirewallEntry and the type of firewall set (blacklist/whitelist) via setFirewall, any packet matching rule added by addHwAccelerationFirewallEntry will not be hw accelerated and this packet will be routed by the S/w stack. On successful execution, a firewall handle will be provided in the callback which can be used to remove the firewall entry removeFirewallEntry().

Parameters

in	*profileId*	Profile identifier on which firewall rule will be added.
in	*entry*	Firewall entry based on protocol type
in	*callback*	optional callback to get the response addHwAccelerationFirewallEntry
in	*slotId*	Specify slot id which has the sim that contains profile id

Returns

Status of addHwAccelerationFirewallEntry i.e. success or suitable status code.

4.48.1.4.2.8 virtual telux::common::Status telux::data::net::IFirewallManager::requestHw↩ AccelerationFirewallEntries (int *profileId*, FirewallEntriesCb *callback*, SlotId *slotId* = *DEFAULT_SLOT_ID*) [pure virtual]

Request Hardware Acceleration rules Returns a list of hardware acceleration firewall entries.

Parameters

in	*profileId*	Profile identifier on which firewall entries are retrieved
in	*callback*	callback to get the response requestHwAccelerationFirewallEntries
in	*slotId*	Specify slot id which has the sim that contains profile id

Returns

Status of requestHwAccelerationFirewallEntries i.e. success or suitable status code.

4.48.1.4.2.9 virtual telux::common::Status telux::data::net::IFirewallManager::requestFirewallEntries (int *profileId*, FirewallEntriesCb *callback*, SlotId *slotId* = *DEFAULT_SLOT_ID*) [pure virtual]

Request Firewall rules

Parameters

in	*profileId*	Profile identifier on which firewall entries are retrieved.
in	*callback*	callback to get the response requestFirewallEntries.
in	*slotId*	Specify slot id which has the sim that contains profile id

Returns

Status of requestFirewallEntries i.e. success or suitable status code.

4.48.1.4.2.10 virtual telux::common::Status telux::data::net::IFirewallManager::removeFirewallEntry (int *profileId*, uint32_t *handle*, telux::common::ResponseCallback *callback* = *nullptr*, SlotId *slotId* = *DEFAULT_SLOT_ID*) [pure virtual]

Remove firewall entry

On platforms with Access control enabled, Caller needs to have TELUX_DATA_NETWORK_CONFIG permission to invoke this API successfully.

Parameters

in	*profileId*	Profile identifier on which firewall entry will be removed.
in	*handle*	handle of Firewall entry to be removed. To retrieve the handle, first use requestFirewallEntries() to get the list of entries added in the system. And then use IFirewallEntry::getHandle(). Handle is also returned when hardware acceleration rule is added using addHwAccelerationFirewallEntry

in	*callback*	callback to get the response removeFirewallEntry
in	*slotId*	Specify slot id which has the sim that contains profile id

Returns

Status of removeFirewallEntry i.e. success or suitable status code.

4.48.1.4.2.11 virtual telux::common::Status telux::data::net::IFirewallManager::enableDmz (int *profileId,* const std::string *ipAddr,* telux::common::ResponseCallback *callback = nullptr,* SlotId *slotId = DEFAULT_SLOT_ID*) [pure virtual]

Enable demilitarized zone (DMZ)

On platforms with Access control enabled, Caller needs to have TELUX_DATA_NETWORK_CONFIG permission to invoke this API successfully.

Parameters

in	*profileId*	Profile identifier on which DMZ will be enabled.
in	*ipAddr*	IP address for which DMZ will be enabled
in	*callback*	optional callback to get the response addDmz
in	*slotId*	Specify slot id which has the sim that contains profile id

Returns

Status of enableDmz i.e. success or suitable status code.

4.48.1.4.2.12 virtual telux::common::Status telux::data::net::IFirewallManager::disableDmz (int *profileId,* const telux::data::IpFamilyType *ipType,* telux::common::ResponseCallback *callback = nullptr,* SlotId *slotId = DEFAULT_SLOT_ID*) [pure virtual]

Disable demilitarized zone (DMZ)

On platforms with Access control enabled, Caller needs to have TELUX_DATA_NETWORK_CONFIG permission to invoke this API successfully.

Parameters

in	*profileId*	Profile identifier on which DMZ will be disabled.
in	*ipType*	Specify IP type of the DMZ to be disabled
in	*callback*	optional callback to get the response removeDmz
in	*slotId*	Specify slot id which has the sim that contains profile id

Returns

Status of disableDmz i.e. success or suitable status code.

4.48.1.4.2.13 virtual telux::common::Status telux::data::net::IFirewallManager::requestDmzEntry (int *profileId*, DmzEntriesCb *dmzCb*, SlotId *slotId* = *DEFAULT_SLOT_ID*) [pure virtual]

Request DMZ entry that was previously set using enableDmz API

Parameters

in	*profileId*	Profile identifier on which DMZ entries are requested.
in	*dmzCb*	callback to get the response requestDmzEntry
in	*slotId*	Specify slot id which has the sim that contains profile id

Returns

Status of requestDmzEntry i.e. success or suitable status code.

4.48.1.4.2.14 virtual telux::common::Status telux::data::net::IFirewallManager::registerListener (std::weak_ptr< IFirewallListener > *listener*) [pure virtual]

Register Firewall Manager as listener for Data Service heath events like data service available or data service not available.

Parameters

in	*listener*	pointer of IFirewallListener object that processes the notification

Returns

Status of registerListener success or suitable status code

4.48.1.4.2.15 virtual telux::common::Status telux::data::net::IFirewallManager::deregisterListener (std::weak_ptr< IFirewallListener > *listener*) [pure virtual]

Removes a previously added listener.

Parameters

in	*listener*	pointer of IFirewallListener object that needs to be removed

Returns

Status of deregisterListener success or suitable status code

4.48.1.4.2.16 virtual telux::data::OperationType telux::data::net::IFirewallManager::getOperationType ( ) [pure virtual]

Get the associated operation type for this instance.

Returns

OperationType of getOperationType i.e. LOCAL or REMOTE.

4.48.1.5 class telux::data::net::IFirewallEntry

Firewall entry class is used for configuring firewall rules.

Public member functions

- virtual std::shared_ptr< IIpFilter > getIProtocolFilter ()=0
- virtual telux::data::Direction getDirection ()=0
- virtual telux::data::IpFamilyType getIpFamilyType ()=0
- virtual uint32_t getHandle ()=0
- virtual ~IFirewallEntry ()

Static Public Attributes

- static const uint32_t INVALID_HANDLE = 0

4.48.1.5.1 Constructors and Destructors

4.48.1.5.1.1 virtual telux::data::net::IFirewallEntry::~IFirewallEntry ( ) [virtual]

Destructor for IFirewallEntry

4.48.1.5.2 Member Function Documentation

4.48.1.5.2.1 virtual std::shared_ptr<IIpFilter> telux::data::net::IFirewallEntry::getIProtocolFilter ( ) [pure virtual]

Get IProtocol filter type

Returns

telux::data::IIpFilter.

4.48.1.5.2.2 virtual telux::data::Direction telux::data::net::IFirewallEntry::getDirection ( ) [pure virtual]

Get firewall direction

Returns

telux::data::Direction.

4.48.1.5.2.3 virtual telux::data::IpFamilyType telux::data::net::IFirewallEntry::getIpFamilyType ( ) [pure virtual]

Get Ip FamilyType

Returns

telux::data::IpFamilyType.

4.48.1.5.2.4 virtual uint32_t telux::data::net::IFirewallEntry::getHandle ( ) [pure virtual]

Get the unique handle identifying this Firewall entry in the system

Returns

uint32_t handle if initialized or INVALID_HANDLE otherwise

4.48.1.5.3 Field Documentation

4.48.1.5.3.1 const uint32_t telux::data::net::IFirewallEntry::INVALID_HANDLE = 0 [static]

4.48.1.6 class telux::data::net::IFirewallListener

Interface for Firewall listener object. Client needs to implement this interface to get access to Firewall services notifications like onServiceStatusChange.

The methods in listener can be invoked from multiple different threads. The implementation should be thread safe.

Public member functions

- virtual void onServiceStatusChange (telux::common::ServiceStatus status)
- virtual ~IFirewallListener ()

4.48.1.6.1 Constructors and Destructors

4.48.1.6.1.1 virtual telux::data::net::IFirewallListener::~IFirewallListener ( ) [virtual]

Destructor for IFirewallListener

4.48.1.6.2 Member Function Documentation

4.48.1.6.2.1 virtual void telux::data::net::IFirewallListener::onServiceStatusChange (telux::common↩ ::ServiceStatus *status*) [virtual]

This function is called when service status changes.

Parameters

in	*status*	- ServiceStatus

4.48.1.7 struct telux::data::net::L2tpSessionConfig

L2TP tunnel sessions configuration

Data fields

Type	Field	Description
uint32_t	locId	Local session id
uint32_t	peerId	Peer session id

4.48.1.8 struct telux::data::net::L2tpTunnelConfig

L2TP tunnel configuration

Data fields

Type	Field	Description
L2tpProtocol	prot	Encapsulation protocols
uint32_t	locId	Local tunnel id
uint32_t	peerId	Peer tunnel id
uint32_t	localUdpPort	Local udp port - if UDP encapsulation is used
uint32_t	peerUdpPort	Peer udp port - if IP encapsulation is used
string	peerIpv6Addr	Peer IPv6 Address - for Ipv6 tunnels
string	peerIpv4Addr	Peer IPv4 Address - for Ipv4 tunnels
string	locIface	interface name to create L2TP tunnel on
IpFamilyType	ipType	Ip family type telux::data::IpFamilyType
vector< L2tp↩ SessionConfig >	sessionConfig	List of L2tp tunnel sessions

4.48.1.9 struct telux::data::net::L2tpSysConfig

L2TP Configuration

Data fields

Type	Field	Description
vector< L2tp↩ TunnelConfig >	configList	List of L2tp tunnel configurations
bool	enableMtu	Enable MTU size setting on underlying interfaces to avoid segmentation
bool	enableTcpMss	Enable TCP MSS clampping on L2TP interfaces to avoid segmentation
uint32_t	mtuSize	Current MTU size in bytes

4.48.1.10 class telux::data::net::IL2tpManager

L2tpManager is a primary interface for configuring L2TP Service. It also provides interface to Subsystem Restart events by registering as listener. Notifications will be received when modem is ready/not ready.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual bool isSubsystemReady ()=0
- virtual std::future< bool > onSubsystemReady ()=0
- virtual telux::common::Status setConfig (bool enable, bool enableMss, bool enableMtu, telux::common::ResponseCallback callback=nullptr, uint32_t mtuSize=0)=0
- virtual telux::common::Status addTunnel (const L2tpTunnelConfig &l2tpTunnelConfig, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status requestConfig (L2tpConfigCb l2tpConfigCb)=0
- virtual telux::common::Status removeTunnel (uint32_t tunnelId, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status registerListener (std::weak_ptr< IL2tpListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< IL2tpListener > listener)=0
- virtual ~IL2tpManager ()

4.48.1.10.1 Constructors and Destructors

4.48.1.10.1.1 virtual telux::data::net::IL2tpManager::~IL2tpManager ( ) [virtual]

Destructor for IL2tpManager

4.48.1.10.2 Member Function Documentation

4.48.1.10.2.1 virtual telux::common::ServiceStatus telux::data::net::IL2tpManager::getServiceStatus ( ) [pure virtual]

Checks the status of L2tp manager and returns the result.

Returns

SERVICE_AVAILABLE If L2tp manager is ready for service. SERVICE_UNAVAILABLE If L2tp manager is temporarily unavailable. SERVICE_FAILED - If L2tp manager encountered an irrecoverable failure.

4.48.1.10.2.2 virtual bool telux::data::net::IL2tpManager::isSubsystemReady ( ) [pure virtual]

Checks if the L2tp manager subsystem is ready.

Returns

True if L2tp Manager is ready for service, otherwise returns false.

Note

This API will be deprecated. getServiceStatus API is recommended as an alternative

4.48.1.10.2.3 virtual std::future<bool> telux::data::net::IL2tpManager::onSubsystemReady ( ) [pure virtual]

Wait for L2tp manager subsystem to be ready.

Returns

A future that caller can wait on to be notified when L2tp manager is ready.

Note

This API will be deprecated. Callback of type InitResponseCb argument in data factory API getL2tpManager is recommended as an alternative.

4.48.1.10.2.4 virtual telux::common::Status telux::data::net::IL2tpManager::setConfig (bool *enable,* bool *enableMss,* bool *enableMtu,* telux::common::ResponseCallback *callback = nullptr,* uint32_t *mtuSize = 0*) [pure virtual]

Enable L2TP for unmanaged Tunnel State

On platforms with Access control enabled, Caller needs to have TELUX_DATA_NETWORK_CONFIG permission to invoke this API successfully.

Parameters

in	*enable*	Enable/Disable L2TP for unmanaged tunnels.
in	*enableMss*	Enable/Disable TCP MSS to be clamped on L2TP interfaces to avoid Segmentation
in	*enableMtu*	Enable/Disable MTU size to be set on underlying interfaces to avoid fragmentation
in	*callback*	optional callback to get the response setConfig
in	*mtuSize*	optional MTU size in bytes. If not set, MTU size will be set to default 1422 bytes

Returns

Status of setConfig i.e. success or suitable status code.

4.48.1.10.2.5 virtual telux::common::Status telux::data::net::IL2tpManager::addTunnel (const L2tpTunnelConfig & *l2tpTunnelConfig,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

Set L2TP Configuration for one tunnel

On platforms with Access control enabled, Caller needs to have TELUX_DATA_NETWORK_CONFIG permission to invoke this API successfully.

Parameters

in	*l2tpTunnelConfig*	Configuration to be set telux::data::net::L2tpTunnelConfig
in	*callback*	Optional callback to get the response addTunnel

Returns

Status of addTunnel i.e. success or suitable status code.

4.48.1.10.2.6 virtual telux::common::Status telux::data::net::IL2tpManager::requestConfig (L2tpConfigCb *l2tpConfigCb*) [pure virtual]

Get Current L2TP Configuration

Parameters

in	*l2tpConfigCb*	Asynchronous callback to get current L2TP configurations

Returns

Status of requestConfig i.e. success or suitable status code.

4.48.1.10.2.7 virtual telux::common::Status telux::data::net::IL2tpManager::removeTunnel (uint32_t *tunnelId,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

Remove L2TP Tunnel

On platforms with Access control enabled, Caller needs to have TELUX_DATA_NETWORK_CONFIG permission to invoke this API successfully.

Parameters

in	*tunnelId*	Tunnel ID to be removed
in	*callback*	optional callback to get the response removeConfig

Returns

Status of removeTunnel i.e. success or suitable status code.

4.48.1.10.2.8 virtual telux::common::Status telux::data::net::IL2tpManager::registerListener (std::weak_ptr< IL2tpListener > *listener*) [pure virtual]

Register L2TP Manager as listener for Data Service heath events like data service available or data service not available.

Parameters

in	*listener*	pointer of IL2tpListener object that processes the notification

Returns

Status of registerListener success or suitable status code

4.48.1.10.2.9 virtual telux::common::Status telux::data::net::IL2tpManager::deregisterListener (std::weak_ptr< IL2tpListener > *listener*) [pure virtual]

Removes a previously added listener.

Parameters

in	*listener*	pointer of IL2tpListener object that needs to be removed

Returns

Status of deregisterListener success or suitable status code

4.48.1.11 class telux::data::net::IL2tpListener

Interface for L2TP listener object. Client needs to implement this interface to get access to L2TP services notifications like onServiceStatusChange.

The methods in listener can be invoked from multiple different threads. The implementation should be thread safe.

Public member functions

- virtual void onServiceStatusChange (telux::common::ServiceStatus status)
- virtual ~IL2tpListener ()

4.48.1.11.1 Constructors and Destructors

4.48.1.11.1.1 virtual telux::data::net::IL2tpListener::~IL2tpListener ( ) [virtual]

Destructor for IL2tpListener

4.48.1.11.2 Member Function Documentation

4.48.1.11.2.1 virtual void telux::data::net::IL2tpListener::onServiceStatusChange (telux::common::ServiceStatus *status*) [virtual]

This function is called when service status changes.

Parameters

in	*status*	- ServiceStatus

4.48.1.12 struct telux::data::net::NatConfig

Structure represents Network Address Translation (NAT) configuration

Data fields

Type	Field	Description
string	addr	Private IP address
uint16_t	port	Private port
uint16_t	globalPort	Global port
IpProtocol	proto	IP protocol telux::data::IpProtocol

4.48.1.13 class telux::data::net::INatManager

NatManager is a primary interface for configuring static network address translation(SNAT) and DMZ (demilitarized zone). It also provides interface to Subsystem Restart events by registering as listener. Notifications will be received when modem is ready/not ready.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual bool isSubsystemReady ()=0
- virtual std::future< bool > onSubsystemReady ()=0
- virtual telux::common::Status addStaticNatEntry (int profileId, const NatConfig &snatConfig, telux::common::ResponseCallback callback=nullptr, SlotId slotId=DEFAULT_SLOT_ID)=0
- virtual telux::common::Status removeStaticNatEntry (int profileId, const NatConfig &snatConfig, telux::common::ResponseCallback callback=nullptr, SlotId slotId=DEFAULT_SLOT_ID)=0
- virtual telux::common::Status requestStaticNatEntries (int profileId, StaticNatEntriesCb snatEntriesCb, SlotId slotId=DEFAULT_SLOT_ID)=0
- virtual telux::common::Status registerListener (std::weak_ptr< INatListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< INatListener > listener)=0
- virtual telux::data::OperationType getOperationType ()=0
- virtual ~INatManager ()

4.48.1.13.1 Constructors and Destructors

4.48.1.13.1.1 virtual telux::data::net::INatManager::~INatManager ( ) [virtual]

Destructor for INatManager

4.48.1.13.2 Member Function Documentation

4.48.1.13.2.1 virtual telux::common::ServiceStatus telux::data::net::INatManager::getServiceStatus ( ) [pure virtual]

Checks the status of NAT manager and returns the result.

Returns

SERVICE_AVAILABLE If Nat manager object is ready for service. SERVICE_UNAVAILABLE If Nat manager object is temporarily unavailable. SERVICE_FAILED - If Nat manager object encountered an irrecoverable failure.

4.48.1.13.2.2 virtual bool telux::data::net::INatManager::isSubsystemReady ( ) [pure virtual]

Checks if the NAT manager subsystem is ready.

Returns

True if NAT Manager is ready for service, otherwise returns false.

Deprecated

Use getServiceStatus API.

4.48.1.13.2.3 virtual std::future<bool> telux::data::net::INatManager::onSubsystemReady ( ) [pure virtual]

Wait for NAT manager subsystem to be ready.

Returns

A future that caller can wait on to be notified when NAT manager is ready.

Deprecated

Use InitResponseCb callback in factory API getNatManager.

4.48.1.13.2.4 virtual telux::common::Status telux::data::net::INatManager::addStaticNatEntry (int *profileId,* const NatConfig & *snatConfig,* telux::common::ResponseCallback *callback = nullptr,* SlotId *slotId = DEFAULT_SLOT_ID*) [pure virtual]

Adds a static Network Address Translation (NAT) entry in the NAT table, these entries are persistent across object, connection and reboot lifetimes. To remove an entry it needs a explicit call to removeStaticNatEntry() API, it supports both IPv4 and IPv6

On platforms with Access control enabled, Caller needs to have TELUX_DATA_NETWORK_CONFIG permission to invoke this API successfully.

Parameters

in	*profileId*	Profile identifier to which static entry will be mapped to.
in	*snatConfig*	snatConfiguration telux::data::net::NatConfig
in	*callback*	optional callback to get the response addStaticNatEntry
in	*slotId*	Specify slot id which has the sim that contains profile id

Returns

Status of addStaticNatEntry i.e. success or suitable status code.

4.48.1.13.2.5 virtual telux::common::Status telux::data::net::INatManager::removeStaticNatEntry (int *profileId*, const NatConfig & *snatConfig*, telux::common::ResponseCallback *callback* = *nullptr*, SlotId *slotId* = *DEFAULT_SLOT_ID*) [pure virtual]

Removes a static Network Address Translation (NAT) entry in the NAT table, it supports both IPv4 and IPv6

On platforms with Access control enabled, Caller needs to have TELUX_DATA_NETWORK_CONFIG permission to invoke this API successfully.

Parameters

in	*profileId*	Profile identifier to which static entry will be removed from.
in	*snatConfig*	snatConfiguration telux::data::net::NatConfig
in	*callback*	optional callback to get the response removeStaticNatEntry
in	*slotId*	Specify slot id which has the sim that contains profile id

Returns

Status of removeStaticNatEntry i.e. success or suitable status code.

4.48.1.13.2.6 virtual telux::common::Status telux::data::net::INatManager::requestStaticNatEntries (int *profileId*, StaticNatEntriesCb *snatEntriesCb*, SlotId *slotId* = *DEFAULT_SLOT_ID*) [pure virtual]

Request list of static nat entries available in the NAT table

Parameters

in	*profileId*	Profile identifier to which static entries will be retrieved.
in	*snatEntriesCb*	Asynchronous callback to get the list of static Network Address Translation (NAT) entries
in	*slotId*	Specify slot id which has the sim that contains profile id

Returns

Status of requestStaticNatEntries i.e. success or suitable status code.

4.48.1.13.2.7 virtual telux::common::Status telux::data::net::INatManager::registerListener (std::weak_ptr< INatListener > *listener*) [pure virtual]

Register Nat Manager as listener for Data Service heath events like data service available or data service not available.

Parameters

in	*listener*	pointer of INatListener object that processes the notification

Returns

Status of registerListener success or suitable status code

4.48.1.13.2.8 virtual telux::common::Status telux::data::net::INatManager::deregisterListener (std::weak_ptr< INatListener > *listener*) [pure virtual]

Removes a previously added listener.

Parameters

in	*listener*	pointer of INatListener object that needs to be removed

Returns

Status of deregisterListener success or suitable status code

4.48.1.13.2.9 virtual telux::data::OperationType telux::data::net::INatManager::getOperationType ( ) [pure virtual]

Get the associated operation type for this instance.

Returns

OperationType of getOperationType i.e. LOCAL or REMOTE.

4.48.1.14 class telux::data::net::INatListener

Interface for Nat listener object. Client needs to implement this interface to get access to Nat services notifications like onServiceStatusChange.

The methods in listener can be invoked from multiple different threads. The implementation should be thread safe.

Public member functions

- virtual void onServiceStatusChange (telux::common::ServiceStatus status)
- virtual ~INatListener ()

4.48.1.14.1 Constructors and Destructors

4.48.1.14.1.1 virtual telux::data::net::INatListener::~INatListener ( ) [virtual]

Destructor for INatListener

4.48.1.14.2 Member Function Documentation

4.48.1.14.2.1 virtual void telux::data::net::INatListener::onServiceStatusChange (telux::common::↩ ServiceStatus *status*) [virtual]

This function is called when service status changes.

Parameters

in	*status*	- ServiceStatus

4.48.1.15 class telux::data::net::ISocksManager

SocksManager is a primary interface for configuring legacy Socks proxy server. It also provides interface to Subsystem Restart events by registering as listener. Notifications will be received when modem is ready/not ready.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual bool isSubsystemReady ()=0
- virtual std::future< bool > onSubsystemReady ()=0
- virtual telux::common::Status enableSocks (bool enable, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status registerListener (std::weak_ptr< ISocksListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< ISocksListener > listener)=0
- virtual telux::data::OperationType getOperationType ()=0
- virtual ~ISocksManager ()

4.48.1.15.1 Constructors and Destructors

4.48.1.15.1.1 virtual telux::data::net::ISocksManager::~ISocksManager ( ) [virtual]

Destructor for Socks Manager

4.48.1.15.2 Member Function Documentation

4.48.1.15.2.1 virtual telux::common::ServiceStatus telux::data::net::ISocksManager::getServiceStatus ( ) [pure virtual]

Checks the status of SocksManager and returns the result.

Returns

SERVICE_AVAILABLE If Socks manager object is ready for service. SERVICE_UNAVAILABLE

If Socks manager object is temporarily unavailable. SERVICE_FAILED - If Socks manager object encountered an irrecoverable failure.

4.48.1.15.2.2 virtual bool telux::data::net::ISocksManager::isSubsystemReady ( ) [pure virtual]

Checks if the SocksManager subsystem is ready.

Returns

True if SocksManager is ready for service, otherwise returns false.

Deprecated

Use getServiceStatus API..

4.48.1.15.2.3 virtual std::future<bool> telux::data::net::ISocksManager::onSubsystemReady ( ) [pure virtual]

Wait for SocksManager subsystem to be ready.

Returns

A future that caller can wait on to be notified when Socksanager is ready.

Deprecated

Use InitResponseCb callback in factory API getSocksManager.

4.48.1.15.2.4 virtual telux::common::Status telux::data::net::ISocksManager::enableSocks (bool *enable,* telux::common::ResponseCallback *callback = nullptr*) [pure virtual]

Enable or Disable Socks proxy service.

On platforms with Access control enabled, Caller needs to have TELUX_DATA_NETWORK_CONFIG permission to invoke this API successfully.

Parameters

in	*enable*	true: enable proxy, false: disable proxy
in	*callback*	optional callback to get the operation error code if any

Returns

Status of proxy enablement i.e. success or suitable status code.

4.48.1.15.2.5 virtual telux::common::Status telux::data::net::ISocksManager::registerListener (std::weak_ptr< ISocksListener > *listener*) [pure virtual]

Register Socks Manager as listener for Data Service heath events like data service available or data service not available.

Parameters

in	*listener*	pointer of ISocksListener object that processes the notification

Returns

Status of registerListener success or suitable status code

4.48.1.15.2.6 virtual telux::common::Status telux::data::net::ISocksManager::deregisterListener (std::weak_ptr< ISocksListener > *listener*) [pure virtual]

Removes a previously added listener.

Parameters

in	*listener*	pointer of ISocksListener object that needs to be removed

Returns

Status of deregisterListener success or suitable status code

4.48.1.15.2.7 virtual telux::data::OperationType telux::data::net::ISocksManager::getOperationType ( ) [pure virtual]

Get the associated operation type for this instance.

Returns

OperationType of getOperationType i.e. LOCAL or REMOTE.

4.48.1.16 class telux::data::net::ISocksListener

Interface for Socks listener object. Client needs to implement this interface to get access to Socks services notifications like onServiceStatusChange.

The methods in listener can be invoked from multiple different threads. The implementation should be thread safe.

Public member functions

- virtual void onServiceStatusChange (telux::common::ServiceStatus status)
- virtual ~ISocksListener ()

4.48.1.16.1 Constructors and Destructors

4.48.1.16.1.1 virtual telux::data::net::ISocksListener::~ISocksListener ( ) [virtual]

Destructor for ISocksListener

4.48.1.16.2 Member Function Documentation

4.48.1.16.2.1 virtual void telux::data::net::ISocksListener::onServiceStatusChange (telux::common::ServiceStatus *status*) [virtual]

This function is called when service status changes.

Parameters

in	*status*	- ServiceStatus

4.48.1.17 class telux::data::net::IVlanManager

VlanManager is a primary interface for configuring VLAN (Virtual Local Area Network). it provide APIs for create, query, remove VLAN interfaces and associate or disassociate with profile IDs. It also provides interface to Subsystem Restart events by registering as listener. Notifications will be received when modem is ready/not ready.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual bool isSubsystemReady ()=0
- virtual std::future< bool > onSubsystemReady ()=0
- virtual telux::common::Status createVlan (const VlanConfig &vlanConfig, CreateVlanCb callback=nullptr)=0
- virtual telux::common::Status removeVlan (int16_t vlanId, InterfaceType ifaceType, telux::common::ResponseCallback callback=nullptr)=0
- virtual telux::common::Status queryVlanInfo (QueryVlanResponseCb callback)=0
- virtual telux::common::Status bindWithProfile (int profileId, int vlanId, telux::common::ResponseCallback callback=nullptr, SlotId slotId=DEFAULT_SLOT_ID)=0
- virtual telux::common::Status unbindFromProfile (int profileId, int vlanId, telux::common::ResponseCallback callback=nullptr, SlotId slotId=DEFAULT_SLOT_ID)=0
- virtual telux::common::Status queryVlanMappingList (VlanMappingResponseCb callback, SlotId slotId=DEFAULT_SLOT_ID)=0
- virtual telux::common::Status registerListener (std::weak_ptr< IVlanListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< IVlanListener > listener)=0
- virtual telux::data::OperationType getOperationType ()=0

- virtual ~IVlanManager ()

4.48.1.17.1 Constructors and Destructors

4.48.1.17.1.1 virtual telux::data::net::IVlanManager::~IVlanManager ( ) [virtual]

Destructor for IVlanManager

4.48.1.17.2 Member Function Documentation

4.48.1.17.2.1 virtual telux::common::ServiceStatus telux::data::net::IVlanManager::getServiceStatus ( ) [pure virtual]

Checks the status of VLAN manager and returns the result.

Returns

SERVICE_AVAILABLE If VLAN manager object is ready for service. SERVICE_UNAVAILABLE If VLAN manager object is temporarily unavailable. SERVICE_FAILED - If VLAN manager object encountered an irrecoverable failure.

4.48.1.17.2.2 virtual bool telux::data::net::IVlanManager::isSubsystemReady ( ) [pure virtual]

Checks if the VLAN manager subsystem is ready.

Returns

True if VLAN Manager is ready for service, otherwise returns false.

Deprecated

Use getServiceStatus API.

4.48.1.17.2.3 virtual std::future<bool> telux::data::net::IVlanManager::onSubsystemReady ( ) [pure virtual]

Wait for VLAN manager subsystem to be ready.

Returns

A future that caller can wait on to be notified when VLAN manager is ready.

Deprecated

Use InitResponseCb callback in factory API getVlanManager.

4.48.1.17.2.4 virtual telux::common::Status telux::data::net::IVlanManager::createVlan (const VlanConfig & *vlanConfig,* CreateVlanCb *callback* = `nullptr`) [pure virtual]

Create a VLAN associated with multiple interfaces Creates VLAN on hardware interface telux::data::InterfaceType, assigns VLAN id, assigns VLAN priority level (according to IEEE 802.1p priority code point-PCP), and sets whether traffic on this VLAN needs to be accelerated. If platform does not support assigning priorities to VLANs and priority is set to value other than 0, telux::common::Status::NOTSUPPORTED is returned. If platform supports Vlan priority, all traffic coming from WWAN or LAN are stamped with priority before sending traffic to tethered client.

On platforms with Access control enabled, Caller needs to have TELUX_DATA_NETWORK_CONFIG permission to invoke this API successfully.

Note

if interface configured as VLAN for the first time, it may trigger auto reboot.

Parameters

in	*vlanConfig*	VLAN configuration
out	*callback*	optional callback to get the response createVlan

Returns

Immediate status of createVlan() request sent i.e. success or suitable status code.

4.48.1.17.2.5 virtual telux::common::Status telux::data::net::IVlanManager::removeVlan (int16_t *vlanId,* InterfaceType *ifaceType,* telux::common::ResponseCallback *callback* = `nullptr`) [pure virtual]

Remove VLAN configuration

On platforms with Access control enabled, Caller needs to have TELUX_DATA_NETWORK_CONFIG permission to invoke this API successfully.

Note

This will delete all clients associated with interface

Parameters

in	*vlanId*	VLAN ID
in	*ifaceType*	telux::data::InterfaceType
out	*callback*	optional callback to get the response removeVlan

Returns

Immediate status of removeVlan() request sent i.e. success or suitable status code.

4.48.1.17.2.6 virtual telux::common::Status telux::data::net::IVlanManager::queryVlanInfo (QueryVlanResponseCb *callback*) [pure virtual]

Query information about all the VLANs in the system

Parameters

out	*callback*	Response callback with list of configured VLANs

Returns

Immediate status of queryVlanInfo() request sent i.e. success or suitable status code.

4.48.1.17.2.7 virtual telux::common::Status telux::data::net::IVlanManager::bindWithProfile (int *profileId*, int *vlanId*, telux::common::ResponseCallback *callback* = *nullptr*, SlotId *slotId* = *DEFAULT_SLOT_ID*) [pure virtual]

Bind a VLAN with a particular profile id and slot id. When a WWAN network interface is brought up using IDataConnectionManager::startDataCall on that profile id and slot id, that interface will be accessible from this VLAN The behavior of this API is dependent on platform/system configuration. If the platform is configured to allow multiple VLANs to be bound to the same profile id - slot id pair then:

- Binding multiple VLANs to any profile id - slot id pair can be achieved by calling this API with each VLAN id. Each VLAN will be associated with it's own bridge.
- Reboot is not triggered with any bind operation. If the platform is not configured to allow multiple VLANs to be bound to the same profile id - slot id pair then:
- Binding VLAN to default profile id and slot id will associate it with bridge0 and trigger automatic reboot.
- Binding VLAN to any other profile id and slot id will associate it with own bridge.
- Multiple VLAN binding attempt to any profile id or slot id will result in error telux::common::ErrorCode::INVALID_OPERATION This setting will be persistant across multiple boots.

Parameters

in	*profileId*	profile id for VLAN association
in	*vlanId*	VLAN ID to be bound to the data call brought up on the profile id
out	*callback*	callback to get the response of associateWithProfileId API
in	*slotId*	Specify slot id which has the sim that contains profile id.

Returns

Immediate status of associateWithProfileId() request sent i.e. success or suitable status code.

4.48.1.17.2.8 virtual telux::common::Status telux::data::net::IVlanManager::unbindFromProfile (int *profileId,* int *vlanId,* telux::common::ResponseCallback *callback = nullptr,* SlotId *slotId = DEFAULT_SLOT_ID*) [pure virtual]

Unbind VLAN id from given slot id and profile id This setting will be persistant across multiple boots.

Parameters

in	*profileId*	profile id for VLAN association
in	*vlanId*	VLAN ID to be unbound to the data call brought up on the profile id
in	*callback*	callback to get the response of associateWithProfileId API
in	*slotId*	Specify slot id which has the sim that contains profile id .

Returns

Immediate status of disassociateFromProfileId() request sent i.e. success or suitable status code

4.48.1.17.2.9 virtual telux::common::Status telux::data::net::IVlanManager::queryVlanMappingList (VlanMappingResponseCb *callback,* SlotId *slotId = DEFAULT_SLOT_ID*) [pure virtual]

Query VLAN mapping of profile id and VLAN id on specified sim

Parameters

in	*callback*	callback to get the response of queryVlanMappingList API
in	*slotId*	Specify slot id which has the sim that contains profile id mapping to VLAN id.

Returns

Immediate status of queryVlanMappingList() request sent i.e. success or suitable status code

4.48.1.17.2.10 virtual telux::common::Status telux::data::net::IVlanManager::registerListener (std::weak_ptr< IVlanListener > *listener*) [pure virtual]

Register VLAN Manager as a listener for Data Service health events like data service available or data service not available.

Parameters

in	*listener*	pointer of IVlanListener object that processes the notification

Returns

Status of registerListener success or suitable status code

4.48.1.17.2.11 virtual telux::common::Status telux::data::net::IVlanManager::deregisterListener (std::weak_ptr< IVlanListener > *listener*) [pure virtual]

Removes a previously added listener.

Parameters

in	*listener*	pointer of IVlanListener object that needs to be removed

Returns

Status of deregisterListener success or suitable status code

4.48.1.17.2.12 virtual telux::data::OperationType telux::data::net::IVlanManager::getOperationType ( ) [pure virtual]

Get the associated operation type for this instance.

Returns

OperationType of getOperationType i.e. LOCAL or REMOTE.

4.48.1.18 class telux::data::net::IVlanListener

Interface for VLAN listener object. Client needs to implement this interface to get access to Socks services notifications like onServiceStatusChange.

The methods in listener can be invoked from multiple different threads. The implementation should be thread safe.

Public member functions

- virtual void onServiceStatusChange (telux::common::ServiceStatus status)
- virtual void onHwAccelerationChanged (const ServiceState state)
- virtual ~IVlanListener ()

4.48.1.18.1 Constructors and Destructors

4.48.1.18.1.1 virtual telux::data::net::IVlanListener::~IVlanListener ( ) [virtual]

Destructor for IVlanListener

4.48.1.18.2 Member Function Documentation

4.48.1.18.2.1 virtual void telux::data::net::IVlanListener::onServiceStatusChange (telux::common::↩ ServiceStatus *status*) [virtual]

This function is called when service status changes.

Parameters

in	*status*	- ServiceStatus

4.48.1.18.2.2 virtual void telux::data::net::IVlanListener::onHwAccelerationChanged (const ServiceState *state*) [virtual]

This function is called when there is a change in IPA Connection Manager daemon state.

Parameters

in	*state*	New state of IPA connection Manager daemon Active/Inactive

Note

This is global state

4.48.2 Enumeration Type Documentation

4.48.2.1 enum telux::data::net::BridgeIFaceType [strong]

Interface types supported for bridge configuration

Enumerator

UNKNOWN
WLAN_AP Wireless Local Area Network (WLAN) in AP mode
WLAN_STA Wireless Local Area Network (WLAN) in STA mode
ETH Ethernet (ETH)

4.48.2.2 enum telux::data::net::L2tpProtocol [strong]

L2TP encapsulation protocols

Enumerator

NONE
IP IP Protocol used for encapsulation
UDP UDP Protocol used for encapsulation

4.49 Telematics_platform_deviceinfo

4.49.1 Data Structure Documentation

4.49.1.1 class telux::platform::IDeviceInfoListener

Listener class for getting device info related notifications . The client needs to implement these methods as briefly as possible and avoid blocking calls in it. The methods in this class can be invoked from multiple different threads. Client needs to make sure that the implementation is thread-safe.

Public member functions

- virtual ~IDeviceInfoListener ()

4.49.1.1.1 Constructors and Destructors

4.49.1.1.1.1 virtual telux::platform::IDeviceInfoListener::~IDeviceInfoListener ( ) [virtual]

Destructor of IDeviceInfoListener

4.49.1.2 struct telux::platform::PlatformVersion

Structure contains the version of the platform software

Data fields

Type	Field	Description
string	meta	Meta Version, for example: SA2150P_SA515M.LE_LE.1-3_2-1-00297-STD.INT-1
string	modem	Modem Version, for example: MPSS.HI.3.1.c3-00114-SDX55_GENAUTO_TEST-1
string	externalApp	External App Version, for example: LE.UM.3.2.3-72102-SA2150p.Int-1
string	integratedApp	Integrated App MDM Version, for example: LE.UM.4.1.1-71802-sa515m.Int-1

4.49.1.3 class telux::platform::IDeviceInfoManager

IDeviceInfoManager provides interface to to retrieve IMEI and platform version operations.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual telux::common::Status registerListener (std::weak_ptr< IDeviceInfoListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< IDeviceInfoListener > listener)=0
- virtual telux::common::Status getPlatformVersion (PlatformVersion &pv)=0
- virtual telux::common::Status getIMEI (std::string &imei)=0

- virtual ~IDeviceInfoManager ()

4.49.1.3.1 Constructors and Destructors

4.49.1.3.1.1 virtual telux::platform::IDeviceInfoManager::~IDeviceInfoManager ( ) [virtual]

Destructor of IDeviceInfoManager

4.49.1.3.2 Member Function Documentation

4.49.1.3.2.1 virtual telux::common::ServiceStatus telux::platform::IDeviceInfoManager::getService↩ Status ( ) [pure virtual]

This status indicates whether the object is in a usable state.

Returns

telux::common::ServiceStatus indicating the current status of the device info service.

4.49.1.3.2.2 virtual telux::common::Status telux::platform::IDeviceInfoManager::registerListener (std::weak_ptr< IDeviceInfoListener > *listener*) [pure virtual]

Registers the listener for FileSystem Manager indications.

Parameters

in	*listener*	- pointer to implemented listener.

Returns

status of the registration request.

4.49.1.3.2.3 virtual telux::common::Status telux::platform::IDeviceInfoManager::deregisterListener (std::weak_ptr< IDeviceInfoListener > *listener*) [pure virtual]

Deregisters the previously registered listener.

Parameters

in	*listener*	- pointer to registered listener that needs to be removed.

Returns

status of the deregistration request.

4.49.1.3.2.4 virtual telux::common::Status telux::platform::IDeviceInfoManager::getPlatformVersion (PlatformVersion & *pv*) [pure virtual]

Get the platform version. Need obtain required permissions from telux_allow_version.

Parameters

out	*pv*	- telux::platform::PlatformVersion

Returns

- telux::common::Status

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.49.1.3.2.5 virtual telux::common::Status telux::platform::IDeviceInfoManager::getIMEI (std::string & *imei*) [pure virtual]

Get the international mobile equipment identity.

Parameters

out	*imei*	- std::string

Returns

- telux::common::Status

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.50 Telematics_sec_mgmt

4.50.1 Data Structure Documentation

4.50.1.1 struct telux::sec::ECCPoint

Represents a point on an elliptic curve.

Data fields

Type	Field	Description
uint8_t *	x	
size_t	xLength	
uint8_t *	y	
size_t	yLength	

4.50.1.2 struct telux::sec::DataDigest

Represents digest of the data whose signature is to be verified.

Data fields

Type	Field	Description
uint8_t *	digest	
size_t	digestLength	

4.50.1.3 struct telux::sec::Signature

Represents signature of the digest to be verified.

Data fields

Type	Field	Description
uint8_t *	rSignature	
uint8_t *	sSignature	
size_t	rsLength	

4.50.1.4 struct telux::sec::Scalar

Represents scalar value to be used with an ECQV operation.

Data fields

Type	Field	Description
uint8_t *	scalar	
size_t	scalarLength	

4.50.1.5 struct telux::sec::OperationResult

Represents a result obtained from the crypto accelerator. The value of an individual field must only be interpreted through helper methods in ResultParser.

Data fields

Type	Field	Description
uint32_t	reserved:4	
uint32_t	id:12	
uint32_t	operationType:3	
uint32_t	result:4	
uint32_t	errCode:9	
uint8_t	data[CA_RESULT_DATA_LENGTH]	

4.50.1.6 class telux::sec::ICryptoAcceleratorListener

Receives ECC signature verification and ECQV calculation result.

Public member functions

- virtual void onVerificationResult (uint32_t uniqueId, telux::common::ErrorCode errorCode, std::vector< uint8_t > resultData)
- virtual void onCalculationResult (uint32_t uniqueId, telux::common::ErrorCode errorCode, std::vector< uint8_t > resultData)
- virtual ~ICryptoAcceleratorListener ()

4.50.1.6.1 Constructors and Destructors

4.50.1.6.1.1 virtual telux::sec::ICryptoAcceleratorListener::~ICryptoAcceleratorListener ( ) [virtual]

Destructor for ICryptoAcceleratorListener.

4.50.1.6.2 Member Function Documentation

4.50.1.6.2.1 virtual void telux::sec::ICryptoAcceleratorListener::onVerificationResult (uint32_t *uniqueId,* telux::common::ErrorCode *errorCode,* std::vector< uint8_t > *resultData*) [virtual]

Invoked to provide an ECC signature verification result.

Parameters

in	*uniqueId*	Unique request identifier. This is the same as what was passed to ICryptoAcceleratorManager::eccPostDigestForVerification()
in	*errorCode*	telux::common::ErrorCode::SUCCESS, if signature passed validation, telux::common::ErrorCode::VERIFICATION_FAILED if all inputs were correct, verification completed and signature was invalid, an appropriate error code in all other cases
in	*resultData*	Contains the r' (computed r-component of the signature)

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.50.1.6.2.2 virtual void telux::sec::ICryptoAcceleratorListener::onCalculationResult (uint32_t *uniqueId*, telux::common::ErrorCode *errorCode*, std::vector< uint8_t > *resultData*) [virtual]

Invoked to provide an ECQV calculation result.

Parameters

in	*uniqueId*	Unique request identifier. This is the same as what was passed to ICryptoAcceleratorManager::ecqvPostDataForMultiply↩ AndAdd()
in	*errorCode*	telux::common::ErrorCode::SUCCESS, if calculation succeeded, otherwise, an appropriate error code
in	*resultData*	Output point Q (Q=kP+A). For CURVE_SM2, CURVE_NISTP256 and CURVE_BRAINPOOLP256R1, byte from 0 to 31 contains x-coordinate, and byte from 32 to 63 contains y-coordinate. For CURVE_NISTP384 and CURVE_BRAINPOOLP384R1, byte from 0 to 47 contains x-coordinate, and byte from 48 to 95 contains y-coordinate.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.50.1.7 class telux::sec::ICryptoAcceleratorManager

Provides support for ECC based signature verification and calculation related crypto operations.

APIs with asynchronous and synchronous semantics are provided for the same operation, providing flexibility to optimally support multiple client solutions.

Clients that prefer to invoke verifications from a thread and consume the results on a different thread should use the asynchronous APIs. Clients that prefer to invoke verification APIs and block until the result is ready, should use the synchronous APIs.

Public member functions

- virtual telux::common::ErrorCode eccPostDigestForVerification (const DataDigest &digest, const ECCPoint &publicKey, const Signature &signature, telux::sec::ECCCurve curve, uint32_t uniqueId, telux::sec::RequestPriority priority)=0
- virtual telux::common::ErrorCode ecqvPostDataForMultiplyAndAdd (const ECCPoint &multiplicandPoint, const ECCPoint &addendPoint, const Scalar &scalar, telux::sec::ECCCurve curve, uint32_t uniqueId, telux::sec::RequestPriority priority)=0
- virtual telux::common::ErrorCode getAsyncResults (std::vector< OperationResult > &results, uint32_t numResultsToRead, int32_t timeout, uint32_t &numResultsRead)=0
- virtual telux::common::ErrorCode eccVerifyDigest (const DataDigest &digest, const ECCPoint &publicKey, const Signature &signature, telux::sec::ECCCurve curve, uint32_t uniqueId, telux::sec::RequestPriority priority, std::vector< uint8_t > &resultData)=0
- virtual telux::common::ErrorCode ecqvPointMultiplyAndAdd (const ECCPoint &multiplicandPoint, const ECCPoint &addendPoint, const Scalar &scalar, telux::sec::ECCCurve curve, uint32_t uniqueId, telux::sec::RequestPriority priority, std::vector< uint8_t > &resultData)=0
- virtual ~ICryptoAcceleratorManager ()

4.50.1.7.1 Constructors and Destructors

4.50.1.7.1.1 virtual telux::sec::ICryptoAcceleratorManager::~ICryptoAcceleratorManager ( ) [virtual]

Destructor of ICryptoAcceleratorManager. Cleans up as applicable.

4.50.1.7.2 Member Function Documentation

4.50.1.7.2.1 virtual telux::common::ErrorCode telux::sec::ICryptoAcceleratorManager::eccPostDigestForVerification (const DataDigest & *digest,* const ECCPoint & *publicKey,* const Signature & *signature,* telux::sec::ECCCurve *curve,* uint32_t *uniqueId,* telux::sec::RequestPriority *priority*) [pure virtual]

Sends hashed ECC data to the crypto accelerator for integrity verification using the given public key and signature.

Verification result is received by the ICryptoAcceleratorListener::onVerificationResult() method for MODE_ASYNC_LISTENER. For MODE_ASYNC_POLL, getAsyncResults() is used to obtain the results.

Parameters

in	*digest*	Digest of data
in	*publicKey*	Uncompressed public key used to verify the signature
in	*signature*	Signature of the digest
in	*curve*	ECC curve on which given public key lies
in	*uniqueId*	Unique identifier for each request. This number must be unique across all requests for which results are pending. Once the result for a request is received, the same number can be reused. Valid value range is 0 <= uniqueId <= 4095.
in	*priority*	Relative priority indicating this digest should be verified before any other low priority digest

Returns

telux::common::ErrorCode::SUCCESS, if the data is sent to the accelerator, otherwise an appropriate error code

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.50.1.7.2.2 virtual telux::common::ErrorCode telux::sec::ICryptoAcceleratorManager::ecqvPostDataForMultiplyAndAdd (const ECCPoint & *multiplicandPoint,* const ECCPoint & *addendPoint,* const Scalar & *scalar,* telux::sec::ECCCurve *curve,* uint32_t *uniqueId,* telux::sec::RequestPriority *priority*) [pure virtual]

Sends data to the crypto accelerator to perform a point multiplication and addition for 'Short Weierstrass' curves; Q=kP+A.

Calculation result is received by the ICryptoAcceleratorListener::onCalculationResult() method for MODE_ASYNC_LISTENER. For MODE_ASYNC_POLL, getAsyncResults() is used to obtain the results.

Parameters

in	*multiplicandPoint*	Point to multiply (P). In context of public key reconstruction, it represents the reconstruction value
in	*addendPoint*	Point to add (A). In context of public key reconstruction, it represents the CA public key
in	*scalar*	Scalar for the scalar multiplication (k). In context of public key reconstruction, it represents the hash construct
in	*curve*	ECC curve associated with point P and A
in	*uniqueId*	Unique identifier for each request. This number must be unique across all requests for which results are pending. Once the result for a request is received, the the same number can be reused. Valid value range is 0 <= uniqueId <= 4095.
in	*priority*	Relative priority indicating this calculation should be performed before any other low priority operation

Returns

telux::common::ErrorCode::SUCCESS, if the data is sent to the accelerator, otherwise an appropriate error code

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.50.1.7.2.3 virtual telux::common::ErrorCode telux::sec::ICryptoAcceleratorManager::getAsync↩ Results (std::vector< OperationResult > & *results*, uint32_t *numResultsToRead*, int32_t *timeout*, uint32_t & *numResultsRead*) [pure virtual]

When using Mode::MODE_ASYNC_POLL, ICryptoAcceleratorManager::eccPostDigestForVerification() and ICryptoAcceleratorManager::ecqvPostDataForMultiplyAndAdd() APIs are used to send request.

The result of these request is obtained asynchronously using this method. It blocks until result(s) is available or timeout occurs.

Caller should allocate sufficient memory pointed by 'results'.

Parameters

in,out	*results*	Buffer that will contain the results
in	*numResultsToRead*	Number of the results to read
in	*timeout*	Time to wait (in milliseconds) for the result(s). Specifying a negative value means an infinite timeout. Zero value means return immediately (there may or may not be any results read).
out	*numResultsRead*	Number of results actually read

Returns

telux::common::ErrorCode::SUCCESS, if the result(s) are obtained successfully, otherwise an appropriate error code

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.50.1.7.2.4 virtual telux::common::ErrorCode telux::sec::ICryptoAcceleratorManager::eccVerifyDigest (const DataDigest & *digest*, const ECCPoint & *publicKey*, const Signature & *signature*, telux::sec::ECCCurve *curve*, uint32_t *uniqueId*, telux::sec::RequestPriority *priority*, std::vector< uint8_t > & *resultData*) [pure virtual]

Verifies the signature of the digest using given public key.

Parameters

in	*digest*	Digest of data
in	*publicKey*	Uncompressed public key used to verify the signature
in	*signature*	Signature of the digest
in	*curve*	ECC curve on which given public key lies
in	*uniqueId*	Unique identifier for each request. This number must be unique across all requests for which results are pending. Once the result for a request is received, the same number can be reused. Valid value range is 0 <= uniqueId <= 4095.
in	*priority*	Relative priority indicating this digest should be verified before any other low priority digest
out	*resultData*	Contains the r' prime (computed r-component of the signature)

Returns

telux::common::ErrorCode::SUCCESS, if signature passed validation, telux::common::ErrorCode::VERIFICATION_FAILED if all inputs were correct, verification completed and signature was invalid, an appropriate error code in all other cases

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.50.1.7.2.5 virtual telux::common::ErrorCode telux::sec::ICryptoAcceleratorManager::ecqvPointMultiplyAndAdd (const ECCPoint & *multiplicandPoint,* const ECCPoint & *addendPoint,* const Scalar & *scalar,* telux::sec::ECCCurve *curve,* uint32_t *uniqueId,* telux::sec::RequestPriority *priority,* std::vector< uint8_t > & *resultData*) [pure virtual]

Performs a point multiplication and addition for 'Short Weierstrass' curves; Q=kP+A with the help of accelerator. This can be used, for example; to reconstruct a public key, using 'Elliptic Curve Qu-Vanstone (ECQV)' implicit certificate scheme.

Parameters

in	*multiplicandPoint*	Point to multiply (P). In context of public key reconstruction, it represents the reconstruction value
in	*addendPoint*	Point to add (A). In context of public key reconstruction, it represents the CA public key
in	*scalar*	Scalar for the scalar multiplication (k). In context of public key reconstruction, it represents the hash construct
in	*curve*	ECC curve associated with point P and A
in	*uniqueId*	Unique identifier for each request. This number must be unique across all requests for which results are pending. Once the result for a request is received, the the same number can be reused. Valid value range is 0 <= uniqueId <= 4095.

in	*priority*	Relative priority indicating this calculation should be performed before any other low priority operation
out	*resultData*	Output point Q (Q=kP+A). For CURVE_SM2, CURVE_NISTP256 and CURVE_BRAINPOOLP256R1, byte from 0 to 31 contains x-coordinate, and byte from 32 to 63 contains y-coordinate. For CURVE_NISTP384 and CURVE_BRAINPOOLP384R1, byte from 0 to 47 contains x-coordinate, and byte from 48 to 95 contains y-coordinate.

Returns

telux::common::ErrorCode::SUCCESS, if the calculation succeeded, otherwise an appropriate error code

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.50.1.8 class telux::sec::ResultParser

Provides helpers to parse fields in the OperationResult.

Static Public Member Functions

- static uint32_t getId (const OperationResult &result)
- static OperationType getOperationType (const OperationResult &result)
- static telux::common::ErrorCode getErrorCode (const OperationResult &result)
- static telux::common::ErrorCode getCAErrorCode (const OperationResult &result)
- static uint8_t * getData (OperationResult &result)

4.50.1.8.1 Member Function Documentation

4.50.1.8.1.1 static uint32_t telux::sec::ResultParser::getId (const OperationResult & *result*) [static]

Gets the unique identifier associated with the result.

Parameters

in	*result*	Result obtained from ICryptoAcceleratorManager::getAsyncResults()

Returns

Unique identifier associated with the result. This is the same as what was passed in request

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.50.1.8.1.2 static OperationType telux::sec::ResultParser::getOperationType (const OperationResult & *result*) [static]

Gets the type of operation corresponding to this result; values are and OperationType::OP_TYPE_VERIFY and OperationType::OP_TYPE_CALCULATE.

Parameters

in	*result*	Result obtained from ICryptoAcceleratorManager::getAsyncResults()

Returns

Operation type - OperationType::OP_TYPE_VERIFY for signature verification, OperationType::OP_TYPE_CALCULATE for point calculation.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.50.1.8.1.3 static telux::common::ErrorCode telux::sec::ResultParser::getErrorCode (const OperationResult & *result*) [static]

Indicates if the operation passed.

Parameters

in	*result*	Result obtained from ICryptoAcceleratorManager::getAsyncResults()

Returns

For ECC verification, telux::common::ErrorCode::SUCCESS, if signature passed validation, telux::common::ErrorCode::VERIFICATION_FAILED if all inputs were correct, verification completed and signature was invalid, an appropriate error code in all other cases. For ECQV calculation, telux::common::ErrorCode::SUCCESS, if the calculation succeeded, an appropriate error code in all other cases

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.50.1.8.1.4 static telux::common::ErrorCode telux::sec::ResultParser::getCAErrorCode (const OperationResult & *result*) [static]

Provides a crypto accelerator hardware specific error code to further identify the actual error. Should be used only if getErrorCode() indicates an error occurred.

Parameters

in	*result*	Result obtained from ICryptoAcceleratorManager::getAsyncResults()

Returns

Error code - telux::common::ErrorCode::∗ as obtained from the accelerator

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.50.1.8.1.5 static uint8_t∗ telux::sec::ResultParser::getData (OperationResult & *result*) [static]

Gets the actual result data. For ECC verification, it contains r-prime and for ECQV it contains coordinates.

Parameters

in	*result*	Result obtained from ICryptoAcceleratorManager::getAsyncResults()

Returns

Pointer to the data, For ECC verification contains r-prime, For ECQV calculatio contains coordinates

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.50.1.9 class telux::sec::ICryptoParam

Specifies how a crypto operation should be performed. An instance of this must be created only thorough CryptoParamBuilder.

Public member functions

- virtual ∼ICryptoParam ()

4.50.1.9.1 Constructors and Destructors

4.50.1.9.1.1 virtual telux::sec::ICryptoParam::~ICryptoParam ( ) [virtual]

4.50.1.10 struct telux::sec::EncryptedData

Represents encrypted data and optional nonce.

Data fields

Type	Field	Description
vector< uint8_t >	encryptedText	Encrypted text.
vector< uint8_t >	nonce	Generated nonce.

4.50.1.11 class telux::sec::ICryptoManager

ICryptoManager provides key management and crypto operation support. It uses trusted hardware bound cryptography. All keys generated are bound to the device cryptographically.

Public member functions

- virtual telux::common::ErrorCode generateKey (std::shared_ptr< ICryptoParam > cryptoParam, std::vector< uint8_t > &keyBlob)=0
- virtual telux::common::ErrorCode importKey (std::shared_ptr< ICryptoParam > cryptoParam, telux::sec::KeyFormat keyFmt, std::vector< uint8_t > const &keyData, std::vector< uint8_t > &keyBlob)=0
- virtual telux::common::ErrorCode exportKey (telux::sec::KeyFormat keyFmt, std::vector< uint8_t > const &keyBlob, std::vector< uint8_t > &keyData)=0
- virtual telux::common::ErrorCode upgradeKey (std::shared_ptr< ICryptoParam > cryptoParam, std::vector< uint8_t > const &oldKeyBlob, std::vector< uint8_t > &newKeyBlob)=0
- virtual telux::common::ErrorCode signData (std::shared_ptr< ICryptoParam > cryptoParam, std::vector< uint8_t > const &keyBlob, std::vector< uint8_t > const &plainText, std::vector< uint8_t > &signature)=0
- virtual telux::common::ErrorCode verifyData (std::shared_ptr< ICryptoParam > cryptoParam, std::vector< uint8_t > const &keyBlob, std::vector< uint8_t > const &plainText, std::vector< uint8_t > const &signature)=0
- virtual telux::common::ErrorCode encryptData (std::shared_ptr< ICryptoParam > cryptoParam, std::vector< uint8_t > const &keyBlob, std::vector< uint8_t > const &plainText, std::shared_ptr< EncryptedData > &encryptedData)=0
- virtual telux::common::ErrorCode decryptData (std::shared_ptr< ICryptoParam > cryptoParam, std::vector< uint8_t > const &keyBlob, std::vector< uint8_t > const &encryptedText, std::vector< uint8_t > &decryptedText)=0
- virtual ~ICryptoManager ()

4.50.1.11.1 Constructors and Destructors

4.50.1.11.1.1 virtual telux::sec::ICryptoManager::~ICryptoManager ( ) [virtual]

Destroys the ICryptoManager instance. Performs cleanup as applicable.

4.50.1.11.2 Member Function Documentation

4.50.1.11.2.1 virtual telux::common::ErrorCode telux::sec::ICryptoManager::generateKey (std::shared_ptr< ICryptoParam > *cryptoParam,* std::vector< uint8_t > & *keyBlob*) [pure virtual]

Generates key and provides it in the form of a corresponding key blob. The key's secret is encrypted in this key blob.

On platforms with access control enabled, the caller needs to have TELUX_SEC_KEY_OPS permission to successfully invoke this API.

Parameters

in	*cryptoParam*	Specifications of the key.
out	*keyBlob*	Key blob representing the key.

Returns

telux::common::ErrorCode as appropriate.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.50.1.11.2.2 virtual telux::common::ErrorCode telux::sec::ICryptoManager::importKey (std::shared_ptr< ICryptoParam > *cryptoParam,* telux::sec::KeyFormat *keyFmt,* std::vector< uint8_t > const & *keyData,* std::vector< uint8_t > & *keyBlob*) [pure virtual]

Creates a key blob from the given key data.

On platforms with access control enabled, the caller needs to have TELUX_SEC_KEY_OPS permission to successfully invoke this API.

Parameters

in	*cryptoParam*	Specifications of the key
in	*keyFmt*	Format in which the key should be imported (KeyFormat)
in	*keyData*	Key's data, in the specified format, to be imported.
out	*keyBlob*	Key blob created from the given key data.

Returns

telux::common::ErrorCode as appropriate.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.50.1.11.2.3 virtual telux::common::ErrorCode telux::sec::ICryptoManager::exportKey (telux::sec::↩ KeyFormat *keyFmt,* std::vector< uint8_t > const & *keyBlob,* std::vector< uint8_t > & *keyData*) [pure virtual]

Generates equivalent key data from the given key blob.

On platforms with access control enabled, the caller needs to have TELUX_SEC_KEY_OPS permission to successfully invoke this API.

Parameters

in	*keyFmt*	KeyFormat Format in which key should be exported.
in	*keyBlob*	Key blob representing the key to be exported.
out	*keyData*	Key's data generated from the given key blob.

Returns

telux::common::ErrorCode as appropriate.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.50.1.11.2.4 virtual telux::common::ErrorCode telux::sec::ICryptoManager::upgradeKey (std↩ ::shared_ptr< ICryptoParam > *cryptoParam,* std::vector< uint8_t > const & *oldKeyBlob,* std::vector< uint8_t > & *newKeyBlob*) [pure virtual]

Upgrades the given key if it has expired. For example, This API can be used when a key has expired due to a system software upgrade.

On platforms with access control enabled, the caller needs to have TELUX_SEC_KEY_OPS permission to successfully invoke this API.

Parameters

in	*cryptoParam*	Input parameters passed to the upgrade algorithm. Specifically, unique data should be set if it was used when the key was originally created.
in	*oldKeyBlob*	Key blob representing the key to be upgraded.
out	*newKeyBlob*	Key blob representing the upgraded key.

Returns

telux::common::ErrorCode as appropriate.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.50.1.11.2.5 virtual telux::common::ErrorCode telux::sec::ICryptoManager::signData (std::shared_ptr< ICryptoParam > *cryptoParam*, std::vector< uint8_t > const & *keyBlob*, std::vector< uint8_t > const & *plainText*, std::vector< uint8_t > & *signature*) [pure virtual]

Generates a signature to verify the integrity of the given data.

On platforms with access control enabled, the caller needs to have TELUX_SEC_SIGN_OPS permission to successfully invoke this API.

Parameters

in	*cryptoParam*	Input parameters passed to the signature generation algorithm.
in	*keyBlob*	Key blob to sign given data.
in	*plainText*	Data to be signed.
out	*signature*	Signature generated for the given data.

Returns

telux::common::ErrorCode as appropriate.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.50.1.11.2.6 virtual telux::common::ErrorCode telux::sec::ICryptoManager::verifyData (std::shared_ptr< ICryptoParam > *cryptoParam*, std::vector< uint8_t > const & *keyBlob*, std::vector< uint8_t > const & *plainText*, std::vector< uint8_t > const & *signature*) [pure virtual]

Verifies integrity of the given data through its signature.

On platforms with access control enabled, the caller needs to have TELUX_SEC_SIGN_OPS permission to successfully invoke this API.

Parameters

in	*cryptoParam*	Input parameters passed to the signature validation algorithm.
in	*keyBlob*	Key blob to verify the given data.
in	*plainText*	Data to be verified.
in	*signature*	Signature of the data.

Returns

telux::common::ErrorCode::SUCCESS if verification is passed otherwise telux::common::ErrorCode as appropriate.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.50.1.11.2.7 virtual telux::common::ErrorCode telux::sec::ICryptoManager::encryptData (std::shared_ptr< ICryptoParam > *cryptoParam*, std::vector< uint8_t > const & *keyBlob*, std::vector< uint8_t > const & *plainText*, std::shared_ptr< EncryptedData > & *encryptedData*) [pure virtual]

Encrypts data per the given inputs to the encryption algorithm.

On platforms with access control enabled, the caller needs to have TELUX_SEC_ENCRYPTION_OPS permission to successfully invoke this API.

Parameters

in	*cryptoParam*	Input parameters passed to the encryption algorithm.
in	*keyBlob*	Key blob to be used for encryption.
in	*plainText*	Data to be encrypted.
out	*encryptedData*	Encrypted data and nonce, if CryptoParamBuilder::setCallerNonce() was not set when creating keys for encryption/decryption).

Returns

telux::common::ErrorCode as appropriate.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.50.1.11.2.8 virtual telux::common::ErrorCode telux::sec::ICryptoManager::decryptData (std::shared_ptr< ICryptoParam > *cryptoParam*, std::vector< uint8_t > const & *keyBlob*, std::vector< uint8_t > const & *encryptedText*, std::vector< uint8_t > & *decryptedText*) [pure virtual]

Decrypts data per the given inputs to the decryption algorithm.

On platforms with access control enabled, the caller needs to have TELUX_SEC_ENCRYPTION_OPS permission to successfully invoke this API.

Parameters

in	*cryptoParam*	Input parameters passed to the decryption algorithm.
in	*keyBlob*	Key blob to be used for decryption.
in	*encryptedText*	Encrypted data to be decrypted.
out	*decryptedText*	Decrypted data.

Returns

telux::common::ErrorCode as appropriate.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.50.1.12 class telux::sec::CryptoParamBuilder

CryptoParamBuilder helps setup input parameters for a given crypto operation.

Public member functions

- CryptoParamBuilder ()
- CryptoParamBuilder setAlgorithm (AlgorithmTypes algorithm)
- CryptoParamBuilder setCryptoOperation (CryptoOperationTypes operation)
- CryptoParamBuilder setDigest (DigestTypes digest)
- CryptoParamBuilder setPadding (PaddingTypes padding)
- CryptoParamBuilder setKeySize (int32_t keySize)
- CryptoParamBuilder setMinimumMacLength (int32_t minMacLength)
- CryptoParamBuilder setMacLength (int32_t macLength)
- CryptoParamBuilder setBlockMode (BlockModeTypes blockMode)
- CryptoParamBuilder setCurve (int32_t curve)
- CryptoParamBuilder setCallerNonce (bool callerNonce)
- CryptoParamBuilder setPublicExponent (uint64_t publicExponent)
- CryptoParamBuilder setInitVector (std::vector< uint8_t > initVector)
- CryptoParamBuilder setUniqueData (std::vector< uint8_t > uniqueData)
- CryptoParamBuilder setAssociatedData (std::vector< uint8_t > associatedData)
- std::shared_ptr< ICryptoParam > build (void)

4.50.1.12.1 Constructors and Destructors

4.50.1.12.1.1 telux::sec::CryptoParamBuilder::CryptoParamBuilder ( )

Allocates an instance of CryptoParamBuilder.

4.50.1.12.2 Member Function Documentation

4.50.1.12.2.1 CryptoParamBuilder telux::sec::CryptoParamBuilder::setAlgorithm (AlgorithmTypes *algorithm*)

When generating keys, specifies with which algorithm the keys will be used. For crypto operations, specifies the algorithm to use. Use telux::sec::Algorithm enumeration to define this.

4.50.1.12.2.2 CryptoParamBuilder telux::sec::CryptoParamBuilder::setCryptoOperation (CryptoOperationTypes *operation*)

When generating keys, specifies the crypto operation(s) for which the key will be used. For crypto operations, specifies the operation itself (encrypting/decrypting/ signing/verifying). Use telux::sec::CryptoOperation enumeration to define this. Multiple operation values can be OR'ed (|).

4.50.1.12.2.3 CryptoParamBuilder telux::sec::CryptoParamBuilder::setDigest (DigestTypes *digest*)

When generating keys, specifies the digest algorithm(s) that may be used with the key to perform signing and verifying operations using RSA, ECDSA, and HMAC keys. For crypto operations, specifies exact digest algorithm to be used. Use telux::sec::Digest enumeration to define this. Multiple values can be OR'ed (|).

4.50.1.12.2.4 CryptoParamBuilder telux::sec::CryptoParamBuilder::setPadding (PaddingTypes *padding*)

When generating keys, specifies the padding modes that may be used with the RSA and AES key. For crypto operations, specifies the exact padding to be used. Use telux::sec::Padding enumeration to define this. Multiple padding values can be OR'ed (|).

4.50.1.12.2.5 CryptoParamBuilder telux::sec::CryptoParamBuilder::setKeySize (int32_t *keySize*)

When generating keys, specifies the size in bits, of the key, measured in the regular way for the key's algorithm.

- For RSA keys, specifies the size of the public modulus.
- For AES keys, specifies length of the secret key material.
- For HMAC keys, specifies the key size in bits.
- For EC keys, selects the EC group.

4.50.1.12.2.6 CryptoParamBuilder telux::sec::CryptoParamBuilder::setMinimumMacLength (int32_t *minMacLength*)

When generating keys, specifies minimum length of the MAC in bits that can be requested or verified with this key for HMAC keys and AES keys that support GCM mode.

4.50.1.12.2.7 CryptoParamBuilder telux::sec::CryptoParamBuilder::setMacLength (int32_t *macLength*)

For crypto operations, specifies requested length of a MAC or GCM (which is guaranteed to be no less then minimum length of the MAC/GCM used when generating the key).

4.50.1.12.2.8 CryptoParamBuilder telux::sec::CryptoParamBuilder::setBlockMode (BlockModeTypes *blockMode*)

When generating keys, specifies the block cipher mode(s) with which this key can be used. For crypto operations, specifies the exact block mode to be used. Use telux::sec::BlockMode enumeration to define this. Multiple block mode values can be OR'ed (|).

4.50.1.12.2.9 CryptoParamBuilder telux::sec::CryptoParamBuilder::setCurve (int32_t *curve*)

When generating the keys using an EC algorithm, only key size, only curve, or both key size and curve can be specified. If only key size is specified, the appropriate NIST curve is selected automatically. If only curve is specified, the given curve is used. If both are specified, the given curve is used and key size is validated.

4.50.1.12.2.10 CryptoParamBuilder telux::sec::CryptoParamBuilder::setCallerNonce (bool *callerNonce*)

When generating AES key, if callerNonce is set to true, it specifies that an explicit nonce will be supplied by the caller during encryption and decryption using setInitVector(). If the callerNonce is set to false (or not set), platform will generate the nonce during encryption. This nonce should be passed during decryption.

4.50.1.12.2.11 CryptoParamBuilder telux::sec::CryptoParamBuilder::setPublicExponent (uint64_t *publicExponent*)

When generating an RSA key, specifies the value of the public exponent for an RSA key pair (necessary for all RSA keys).

4.50.1.12.2.12 CryptoParamBuilder telux::sec::CryptoParamBuilder::setInitVector (std::vector< uint8_t > *initVector*)

When performing AES crypto operations, specifies the initialization vector to be used.

4.50.1.12.2.13 CryptoParamBuilder telux::sec::CryptoParamBuilder::setUniqueData (std::vector< uint8_t > *uniqueData*)

When generating or importing a key, an optional arbitrary value can be supplied through this method. In all subsequent use of the key, this value must be supplied again. The data given is bound to the key cryptographically. This data ties the key to the caller.

4.50.1.12.2.14 CryptoParamBuilder telux::sec::CryptoParamBuilder::setAssociatedData (std::vector< uint8_t > *associatedData*)

When encrypting/decrypting data, this specifies optional associated data to be used. This is applicable only for AES-GCM algorithm.

4.50.1.12.2.15 std::shared_ptr<ICryptoParam> telux::sec::CryptoParamBuilder::build (void)

Creates an instance of ICryptoParam based on the setter methods invoked on the builder. After building the builder's state is reset.

4.50.1.13 class telux::sec::SecurityFactory

SecurityFactory allows creation of ICryptoManager and ICryptoAcceleratorManager.

Public member functions

- virtual std::shared_ptr< ICryptoManager > getCryptoManager (telux::common::ErrorCode &ec)=0
- virtual std::shared_ptr< ICryptoAcceleratorManager > getCryptoAcceleratorManager (telux::common::ErrorCode &ec, Mode mode, std::weak_ptr< ICryptoAcceleratorListener > cryptoAccelListener=std::weak_ptr< ICryptoAcceleratorListener >())=0

Static Public Member Functions

- static SecurityFactory & getInstance ()

4.50.1.13.1 Member Function Documentation

4.50.1.13.1.1 static SecurityFactory& telux::sec::SecurityFactory::getInstance ( ) [static]

Gets the SecurityFactory instance.

4.50.1.13.1.2 virtual std::shared_ptr<ICryptoManager> telux::sec::SecurityFactory::getCryptoManager (telux::common::ErrorCode & *ec*) [pure virtual]

Instantiates a CryptoManager instance that can be used to perform key management and cryptographic operations.

Parameters

out	*ec*	telux::common::ErrorCode::SUCCESS if ICryptoManager is created successfully, otherwise, an appropriate error code

Returns

ICryptoManager instance

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.50.1.13.1.3 virtual std::shared_ptr<ICryptoAcceleratorManager> telux::sec::Security↩ Factory::getCryptoAcceleratorManager (telux::common::ErrorCode & *ec,* Mode *mode,* std::weak_ptr< ICryptoAcceleratorListener > *cryptoAccelListener =* `std::weak_ptr<` ICryptoAcceleratorListener `>()`) `[pure virtual]`

Provides a CryptoAcceleratorManager instance that can be used to perform cryptographic operations requiring elliptic-curve cryptography (ECC) verifications and calculations.

Parameters

out	*ec*	telux::common::ErrorCode::SUCCESS if ICryptoAcceleratorManager is created successfully, otherwise, an appropriate error code

Providing ICryptoAcceleratorListener instance is mandatory when using Mode::MODE_ASYNC_LISTENER. It is not required with modes, Mode::MODE_SYNC and Mode::MODE_ASYNC_POLL for cryptographic operations.

To receive subsystem-restart (SSR) updates, application must provide ICryptoAcceleratorListener instance (irrespective of Mode::∗) and implement method telux::common::IServiceStatusListener::onServiceStatusChange().

Specifying mode (Mode::∗) defines how an application will send request and receive cryptographic results.

Passing listener determines whether an application is also interested in SSR updates in addition to cryptographic results or not.

Parameters

in	*mode*	Defines how users obtain verification and calculation results
in	*cryptoAccelListener*	Optional, listener for ECC signature verification and ECQV calculation results

Returns

ICryptoAcceleratorManager instance

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.50.2 Enumeration Type Documentation

4.50.2.1 enum telux::sec::Mode [strong]

Defines how the user gets verification and calculation results.

Enumerator

MODE_SYNC ICryptoAcceleratorManager::eccVerifyDigest() and ICryptoAcceleratorManager::ecqvPointMultiplyAndAdd() APIs are used to send verification and calculation data and obtain results synchronously.

MODE_ASYNC_POLL ICryptoAcceleratorManager::eccPostDigestForVerification() and ICrypto↩ AcceleratorManager::ecqvPostDataForMultiplyAndAdd() APIs are used to send verification and calculation data. Results are obtained via ICryptoAcceleratorManager::getAsyncResults() API.

MODE_ASYNC_LISTENER ICryptoAcceleratorManager::eccPostDigestForVerification() and ICryptoAcceleratorManager::ecqvPostDataForMultiplyAndAdd() APIs are used to send verification and calculation data. Results are obtained asynchronously in ICryptoAcceleratorListener::onVerificationResult() and ICryptoAcceleratorListener::onCalculationResult() callbacks.

4.50.2.2 enum telux::sec::RequestPriority [strong]

Relative priority of the request.

Enumerator

REQ_PRIORITY_HIGH High priority

REQ_PRIORITY_NORMAL Lower priority (compared to high priority data)

4.50.2.3 enum telux::sec::ECCCurve [strong]

Elliptic curve used by ECC algorithm.

Enumerator

CURVE_SM2

CURVE_NISTP256

CURVE_NISTP384

CURVE_BRAINPOOLP256R1

CURVE_BRAINPOOLP384R1

4.50.2.4 enum telux::sec::OperationType [strong]

Type of operation carried by crypto accelerator.

Enumerator

OP_TYPE_VERIFY
OP_TYPE_CALCULATE

4.50.2.5 enum telux::sec::CryptoOperation

Specifies the operation for which the key can be used. A key can be used for multiple operation types.

Enumerator

CRYPTO_OP_ENCRYPT Key will be used for encryption.
CRYPTO_OP_DECRYPT Key will be used for decryption.
CRYPTO_OP_SIGN Key will be used for signing.
CRYPTO_OP_VERIFY Key will be used for verification.

4.50.2.6 enum telux::sec::BlockMode

Specifies the block cipher mode(s) with which the AES key may be used.

Enumerator

BLOCK_MODE_ECB Electronic code block mode
BLOCK_MODE_CBC Cipher block chain mode
BLOCK_MODE_CTR Counter-based mode
BLOCK_MODE_GCM Galois/counter mode

4.50.2.7 enum telux::sec::Padding

Padding modes that may be applied to plain text for encryption operations. Only cryptographically-appropriate pairs are specified here.

Enumerator

PADDING_NONE No padding.
PADDING_RSA_OAEP RSA optimal asymmetric encryption padding.
PADDING_RSA_PSS RSA probabilistic signature scheme.
PADDING_RSA_PKCS1_1_5_ENC RSA PKCS#1 v1.5 padding for encryption.
PADDING_RSA_PKCS1_1_5_SIGN RSA PKCS#1 v1.5 padding for signing.
PADDING_PKCS7 Public-key cryptography standard.

4.50.2.8 enum telux::sec::Digest

Specifies the digest algorithms that may be used with the key to perform signing and verification operations using RSA, ECDSA, and HMAC keys. The digest used during signing or verification must match the digest associated with the key when the key was generated.

Enumerator

DIGEST_NONE No digest.
DIGEST_MD5 Message-digest algorithm.
DIGEST_SHA1 Secure hash algorithm 1

DIGEST_SHA_2_224 Secure hash algorithm 2, digest 224.
DIGEST_SHA_2_256 Secure hash algorithm 2, digest 256.
DIGEST_SHA_2_384 Secure hash algorithm 2, digest 384.
DIGEST_SHA_2_512 Secure hash algorithm 2, digest 512.

4.50.2.9 enum telux::sec::Algorithm

Algorithm for signing, verification, encryption, and decryption operations.

Enumerator

ALGORITHM_UNKNOWN Unspecified algorithm.
ALGORITHM_RSA RSA (Rivest–Shamir–Adleman) algorithm.
ALGORITHM_EC Elliptic-curve algorithm.
ALGORITHM_AES Advanced encryption standard algorithm.
ALGORITHM_HMAC Hash-based message authentication code algorithm.

4.50.2.10 enum telux::sec::Curve

NIST curves used with ECDSA.

Enumerator

CURVE_P_224 NIST curve P-224.
CURVE_P_256 NIST curve P-256.
CURVE_P_384 NIST curve P-384.
CURVE_P_521 NIST curve P-521.

4.50.2.11 enum telux::sec::KeyFormat

Formats for key import and export.

Enumerator

KEY_FORMAT_X509 Public key export.
KEY_FORMAT_PKCS8 Asymmetric key pair import.
KEY_FORMAT_RAW Symmetric key import and export.

4.50.3 Variable Documentation

4.50.3.1 const uint32_t telux::sec::CA_RESULT_DATA_LENGTH = 96 `[static]`

Length of the unparsed raw result from the crypto accelerator.

4.51 Telematics_ecall

4.51.1 Data Structure Documentation

4.51.1.1 class telux::tel::IEcallManager

IEcallManager allows operations related to automotive emergency call management and its related configurations.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual telux::common::Status setConfig (EcallConfig config)=0
- virtual telux::common::Status getConfig (EcallConfig &config)=0
- virtual telux::common::Status registerListener (std::weak_ptr< IEcallListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< IEcallListener > listener)=0
- virtual ~IEcallManager ()

4.51.1.1.1 Constructors and Destructors

4.51.1.1.1.1 virtual telux::tel::IEcallManager::~IEcallManager ( ) [virtual]

4.51.1.1.2 Member Function Documentation

4.51.1.1.2.1 virtual telux::common::ServiceStatus telux::tel::IEcallManager::getServiceStatus ( ) [pure virtual]

Checks the status of IEcallManager sub-system and returns the result.

Returns

the status of IEcallManager sub-system status telux::common::ServiceStatus

Deprecated

This API is not being supported

4.51.1.1.2.2 virtual telux::common::Status telux::tel::IEcallManager::setConfig (EcallConfig *config*) [pure virtual]

Set the configuration related to emergency call. The configuration is persistent and takes effect when the next emergency call is dialed.

Parameters

in	*config*	eCall configuration to be set EcallConfig

Returns

Status of setConfig i.e. success or suitable error code.

Deprecated

This API is not being supported. Use ICallManager::setECallConfig() API instead.

4.51.1.1.2.3 virtual telux::common::Status telux::tel::IEcallManager::getConfig (EcallConfig & *config*) [pure virtual]

Get the configuration related to emergency call.

Parameters

out	*config*	Parameter to hold the fetched eCall configuration EcallConfig

Returns

Status of getConfig i.e. success or suitable error code.

Deprecated

This API is not being supported. Use ICallManager::getECallConfig() API instead.

4.51.1.1.2.4 virtual telux::common::Status telux::tel::IEcallManager::registerListener (std::weak_ptr< IEcallListener > *listener*) [pure virtual]

Register a listener for notifications from the EcallManager.

Parameters

in	*listener*	Pointer to IEcallListener object that processes the notification

Returns

Status of registerListener i.e. success or suitable error code.

Deprecated

This API is not being supported

4.51.1.1.2.5 virtual telux::common::Status telux::tel::IEcallManager::deregisterListener (std::weak_ptr< IEcallListener > *listener*) [pure virtual]

Deregister a previously registered listener.

Parameters

in	*listener*	Pointer to IEcallListener object that needs to be deregistered.

Returns

Status of deregisterListener i.e. success or suitable error code.

Deprecated

This API is not being supported

4.51.1.2 class telux::tel::IEcallListener

Listener class to notify service status change notifications. The listener method can be invoked from multiple different threads. Client needs to make sure that implementation is thread-safe.

Public member functions

- virtual ~IEcallListener ()

4.51.1.2.1 Constructors and Destructors

4.51.1.2.1.1 virtual telux::tel::IEcallListener::~IEcallListener ( ) [virtual]

Destructor of IEcallListener

4.52 Telematics_supp_services

4.52.1 Data Structure Documentation

4.52.1.1 class telux::tel::ISuppServicesListener

A listener class for receiving supplementary services notifications. The methods in listener can be invoked from multiple different threads. The implementation should be thread safe.

Public member functions

- virtual ~ISuppServicesListener ()

  Destroy the ISuppServicesListener object.

4.52.1.1.1 Constructors and Destructors

4.52.1.1.1.1 virtual telux::tel::ISuppServicesListener::~ISuppServicesListener ( ) [virtual]

Destroy the ISuppServicesListener object.

4.52.1.2 struct telux::tel::ForwardInfo

Represents parameters for forwarding.

Data fields

Type	Field	Description
SuppServices↩ Status	status	Status of the supplemetary service
ServiceClass	serviceClass	Service class
string	number	Phone number to which the call to be forwarded
uint8_t	noReplyTimer	No reply timer

4.52.1.3 struct telux::tel::ForwardReq

Represents parameters required for forwarding request.

Data fields

Type	Field	Description
Forward↩ Operation	operation	Type of operation for forwarding
ForwardReason	reason	Reason for call forwarding telux::tel::ForwardReason
ServiceClass	serviceClass	Service Class for operation telux::tel::ServiceClass
string	number	Number to which call has to be forwarded. This parameter is required only for registration purpose only. telux::tel::ForwardOperation::REGISTER
uint8_t	noReplyTimer	Timer for no reply operation. Required only for no reply forward reason. telux::tel::ForwardReason::NOREPLY.

4.52.1.4 class telux::tel::ISuppServicesManager

ISuppServicesManager is the interface to provide supplementary services like call forwarding and call waiting.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual telux::common::Status setCallWaitingPref (SuppServicesStatus suppSvcStatus, SetSuppSvcPrefCallback callback=nullptr)=0
- virtual telux::common::Status requestCallWaitingPref (GetCallWaitingPrefExCb callback)=0
- virtual telux::common::Status setForwardingPref (ForwardReq forwardReq, SetSuppSvcPrefCallback callback=nullptr)=0
- virtual telux::common::Status requestForwardingPref (ServiceClass serviceClass, ForwardReason reason, GetForwardingPrefExCb callback)=0
- virtual telux::common::Status setOirPref (ServiceClass serviceClass, SuppServicesStatus suppSvcStatus, SetSuppSvcPrefCallback callback=nullptr)=0
- virtual telux::common::Status requestOirPref (ServiceClass serviceClass, GetOirPrefCb callback)=0
- virtual telux::common::Status registerListener (std::weak_ptr< ISuppServicesListener > listener)=0
- virtual telux::common::Status removeListener (std::weak_ptr< ISuppServicesListener > listener)=0
- virtual ~ISuppServicesManager ()
- virtual telux::common::Status requestCallWaitingPref (GetCallWaitingPrefCb callback)=0
- virtual telux::common::Status requestForwardingPref (ServiceClass serviceClass, ForwardReason reason, GetForwardingPrefCb callback)=0

4.52.1.4.1 Constructors and Destructors

4.52.1.4.1.1 virtual telux::tel::ISuppServicesManager::~ISuppServicesManager ( ) [virtual]

Destructor for ISupplementaryServicesManager

4.52.1.4.2 Member Function Documentation

4.52.1.4.2.1 virtual telux::common::ServiceStatus telux::tel::ISuppServicesManager::getServiceStatus ( ) [pure virtual]

This status indicates whether the ISuppServicesManager object is in a usable state.

Returns

SERVICE_AVAILABLE - If ISuppServicesManager manager is ready for service.
SERVICE_UNAVAILABLE - If ISuppServicesManager manager is temporarily unavailable.
SERVICE_FAILED - If ISuppServicesManager manager encountered an irrecoverable failure.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.52.1.4.2.2 virtual telux::common::Status telux::tel::ISuppServicesManager::setCallWaitingPref (SuppServicesStatus *suppSvcStatus*, SetSuppSvcPrefCallback *callback* = `nullptr`) `[pure virtual]`

Enable/disable call waiting on device.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SUPP_SERVICES permissions to invoke this API successfully.

Parameters

`in`	*suppSvcStatus*	- Call waiting preference telux::tel::SuppServicesStatus.
`in`	*callback*	- Callback function to get the response of setCallWaitingPref

Returns

Status of setCallWaitingPref i.e. success or suitable error code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.52.1.4.2.3 virtual telux::common::Status telux::tel::ISuppServicesManager::requestCallWaitingPref (GetCallWaitingPrefExCb *callback*) `[pure virtual]`

This API queries the preference for call waiting.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SUPP_SERVICES permissions to invoke this API successfully.

Parameters

`in`	*callback*	- Callback function to get the response of call waiting preference.

Returns

Status of requestCallWaitingPref i.e. success or suitable error code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.52.1.4.2.4 virtual telux::common::Status telux::tel::ISuppServicesManager::setForwardingPref (ForwardReq *forwardReq,* SetSuppSvcPrefCallback *callback* = `nullptr`) [pure virtual]

To set call forwarding preference.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SUPP_SERVICES permissions to invoke this API successfully.

Parameters

in	*forwardReq*	- Parameters for call forwarding operation. telux::tel::ForwardReq
in	*callback*	- Callback function to get response of setForwardingPref API.

Returns

Status of setForwardingPref i.e. success or suitable error code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.52.1.4.2.5 virtual telux::common::Status telux::tel::ISuppServicesManager::requestForwardingPref (ServiceClass *serviceClass,* ForwardReason *reason,* GetForwardingPrefExCb *callback*) [pure virtual]

This API queries preference for call forwarding supplementary service. If active, returns for which service classes and call forwarding number it is active. There is an option to configure for which service class the request is made, if the option is not configured it assumes that the request is made for all service classes.

On platforms with Access control enabled, Caller needs to have TELUX_TEL_SUPP_SERVICES permissions to invoke this API successfully.

Parameters

in	*serviceClass*	- Service class telux::tel::ServiceClass.
in	*callback*	- Callback function to get the response of request call forwarding preference.

Returns

Status of requestForwardingPref i.e. success or suitable error code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.52.1.4.2.6 virtual telux::common::Status telux::tel::ISuppServicesManager::setOirPref (ServiceClass *serviceClass,* SuppServicesStatus *suppSvcStatus,* SetSuppSvcPrefCallback *callback* = *nullptr*) [pure virtual]

Activate/Deactivate originating identification restriction preference on the device. If the OIR service was activated, the original call number will be restricted to the target when a call is dialed to a subscriber.

On platforms with access control enabled, the caller must have TELUX_TEL_SUPP_SERVICES permissions to invoke this API successfully.

Parameters

in	*serviceClass*	- Service class telux::tel::ServiceClass.
in	*suppSvcStatus*	- OIR Status telux::tel::SuppServicesStatus.
in	*callback*	- Callback function to get the response of setOIRPref

Returns

Status of setOirPref i.e. success or suitable error code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.52.1.4.2.7 virtual telux::common::Status telux::tel::ISuppServicesManager::requestOirPref (ServiceClass *serviceClass,* GetOirPrefCb *callback*) [pure virtual]

This API queries the originating identification restriction preference.

On platforms with access control enabled, the caller must have TELUX_TEL_SUPP_SERVICES permissions to invoke this API successfully.

Parameters

in	*serviceClass*	- Service class telux::tel::ServiceClass.
in	*callback*	- Callback function to get the response of requestOIRPref

Returns

Status of requestOirPref i.e. success or suitable error code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

4.52.1.4.2.8 virtual telux::common::Status telux::tel::ISuppServicesManager::registerListener (std::weak_ptr< ISuppServicesListener > *listener*) [pure virtual]

Register a listener for supplementary services events.

Parameters

in	*listener*	Pointer to ISuppServicesListener object that processes the notification.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

Returns

Status of registerListener i.e. success or suitable status code.

4.52.1.4.2.9 virtual telux::common::Status telux::tel::ISuppServicesManager::removeListener (std::weak_ptr< ISuppServicesListener > *listener*) [pure virtual]

Remove a previously added listener.

Parameters

in	*listener*	Pointer to ISuppServicesListener object that needs to be removed.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

Returns

Status of removeListener i.e. success or suitable status code.

4.52.1.4.2.10 virtual telux::common::Status telux::tel::ISuppServicesManager::requestCallWaitingPref (GetCallWaitingPrefCb *callback*) [pure virtual]

This API queries the preference for call waiting.

Parameters

in	*callback*	- Callback function to get the response of requestCallWaitingPref.

Returns

Status of requestCallWaitingPref i.e. success or suitable error code.

Deprecated

This API is not being supported instead use requestCallWaitingPref(GetCallWaitingPrefExCb) API.

4.52.1.4.2.11 virtual telux::common::Status telux::tel::ISuppServicesManager::requestForwardingPref (ServiceClass *serviceClass,* ForwardReason *reason,* GetForwardingPrefCb *callback*) [pure virtual]

This API queries preference for call forwarding supplementary service. If active, returns for which service classes and call forwarding number it is active. It also returns the provision status of the supplemetary service. There is an option to configure for which service class the request is made, if the option is not configured it assumes that the request is made for all service classes.

Parameters

in	*serviceClass*	- Service class telux::tel::ServiceClass.
in	*callback*	- Callback function to get the response of request call forwarding preference.

Returns

Status of requestForwardingPref i.e. success or suitable error code.

Deprecated

This API is not being supported instead use requestForwardingPref(ServiceClass serviceClass, ForwardReason reason, GetForwardingPrefExCb callback) API.

4.52.2 Enumeration Type Documentation

4.52.2.1 enum telux::tel::SuppServicesStatus [strong]

Defines supplementary services status.

Enumerator

UNKNOWN Supplementary service status unknown
ENABLED Supplementary service is enabled
DISABLED Supplementary service is disabled

4.52.2.2 enum telux::tel::SuppSvcProvisionStatus [strong]

Defines supplementary services provision status.

Enumerator

UNKNOWN Supplementary service provision status unknown
NOT_PROVISIONED Supplementary service is not provisioned
PROVISIONED Supplementary service is provisioned
PRESENTATION_RESTRICTED Supplementary service is presentation restricted
PRESENTATION_ALLOWED Supplementary service is presentation allowed

4.52.2.3 enum telux::tel::ForwardOperation [strong]

Defines call forwarding operation.

Enumerator

UNKNOWN Status unknown
ACTIVATE To activate call forwarding
DEACTIVATE To deactivate call forwarding
REGISTER To register for call forwarding
ERASE To erase the previous registration

4.52.2.4 enum telux::tel::ForwardReason [strong]

Defines reasons for call forwarding.

Enumerator

UNCONDITIONAL Unconditional call forwarding
BUSY Forward when the device is busy on another call
NOREPLY Forward when there is no reply
NOT_REACHABLE Forward when the device is unreachable
NOT_LOGGED_IN Forward when the device is not logged in

4.52.2.5 enum telux::tel::ServiceClassType [strong]

Defines service class for telephony

Enumerator

NONE Service class not provided
VOICE Service class voice

4.52.2.6 enum telux::tel::FailureCause [strong]

Represents the cause for supplementary services failure.

Enumerator

UNAVAILABLE
OFFLINE
CDMA_LOCK
NO_SRV
FADE

INTERCEPT
REORDER
REL_NORMAL
REL_SO_REJ
INCOM_CALL
ALERT_STOP
CLIENT_END
ACTIVATION
MC_ABORT
MAX_ACCESS_PROBE
PSIST_N
UIM_NOT_PRESENT
ACC_IN_PROG
ACC_FAIL
RETRY_ORDER
CCS_NOT_SUPPORTED_BY_BS
NO_RESPONSE_FROM_BS
REJECTED_BY_BS
INCOMPATIBLE
ACCESS_BLOCK
ALREADY_IN_TC
EMERGENCY_FLASHED
USER_CALL_ORIG_DURING_GPS
USER_CALL_ORIG_DURING_SMS
USER_CALL_ORIG_DURING_DATA
REDIR_OR_HANDOFF
ACCESS_BLOCK_ALL
OTASP_SPC_ERR
IS707B_MAX_ACC
ACC_FAIL_REJ_ORD
ACC_FAIL_RETRY_ORD
TIMEOUT_T42
TIMEOUT_T40
SRV_INIT_FAIL
T50_EXP
T51_EXP
RL_ACK_TIMEOUT
BAD_FL
TRM_REQ_FAIL
TIMEOUT_T41
INCOM_REJ
SETUP_REJ
NETWORK_END
NO_FUNDS
NO_GW_SRV
NO_CDMA_SRV
NO_FULL_SRV
MAX_PS_CALLS
UNKNOWN_SUBSCRIBER
ILLEGAL_SUBSCRIBER

BEARER_SERVICE_NOT_PROVISIONED
TELE_SERVICE_NOT_PROVISIONED
ILLEGAL_EQUIPMENT
CALL_BARRED
ILLEGAL_SS_OPERATION
SS_ERROR_STATUS
SS_NOT_AVAILABLE
SS_SUBSCRIPTION_VIOLATION
SS_INCOMPATIBILITY
FACILITY_NOT_SUPPORTED
ABSENT_SUBSCRIBER
SHORT_TERM_DENIAL
LONG_TERM_DENIAL
SYSTEM_FAILURE
DATA_MISSING
UNEXPECTED_DATA_VALUE
PWD_REGISTRATION_FAILURE
NEGATIVE_PWD_CHECK
NUM_OF_PWD_ATTEMPTS_VIOLATION
POSITION_METHOD_FAILURE
UNKNOWN_ALPHABET
USSD_BUSY
REJECTED_BY_USER
REJECTED_BY_NETWORK
DEFLECTION_TO_SERVED_SUBSCRIBER
SPECIAL_SERVICE_CODE
INVALID_DEFLECTED_TO_NUMBER
MPTY_PARTICIPANTS_EXCEEDED
RESOURCES_NOT_AVAILABLE
UNASSIGNED_NUMBER
NO_ROUTE_TO_DESTINATION
CHANNEL_UNACCEPTABLE
OPERATOR_DETERMINED_BARRING
NORMAL_CALL_CLEARING
USER_BUSY
NO_USER_RESPONDING
USER_ALERTING_NO_ANSWER
CALL_REJECTED
NUMBER_CHANGED
PREEMPTION
DESTINATION_OUT_OF_ORDER
INVALID_NUMBER_FORMAT
FACILITY_REJECTED
RESP_TO_STATUS_ENQUIRY
NORMAL_UNSPECIFIED
NO_CIRCUIT_OR_CHANNEL_AVAILABLE
NETWORK_OUT_OF_ORDER
TEMPORARY_FAILURE
SWITCHING_EQUIPMENT_CONGESTION
ACCESS_INFORMATION_DISCARDED

REQUESTED_CIRCUIT_OR_CHANNEL_NOT_AVAILABLE
RESOURCES_UNAVAILABLE_OR_UNSPECIFIED
QOS_UNAVAILABLE
REQUESTED_FACILITY_NOT_SUBSCRIBED
INCOMING_CALLS_BARRED_WITHIN_CUG
BEARER_CAPABILITY_NOT_AUTH
BEARER_CAPABILITY_UNAVAILABLE
SERVICE_OPTION_NOT_AVAILABLE
ACM_LIMIT_EXCEEDED
BEARER_SERVICE_NOT_IMPLEMENTED
REQUESTED_FACILITY_NOT_IMPLEMENTED
ONLY_DIGITAL_INFORMATION_BEARER_AVAILABLE
SERVICE_OR_OPTION_NOT_IMPLEMENTED
INVALID_TRANSACTION_IDENTIFIER
USER_NOT_MEMBER_OF_CUG
INCOMPATIBLE_DESTINATION
INVALID_TRANSIT_NW_SELECTION
SEMANTICALLY_INCORRECT_MESSAGE
INVALID_MANDATORY_INFORMATION
MESSAGE_TYPE_NON_IMPLEMENTED
MESSAGE_TYPE_NOT_COMPATIBLE_WITH_PROTOCOL_STATE
INFORMATION_ELEMENT_NON_EXISTENT
CONDITONAL_IE_ERROR
MESSAGE_NOT_COMPATIBLE_WITH_PROTOCOL_STATE
RECOVERY_ON_TIMER_EXPIRED
PROTOCOL_ERROR_UNSPECIFIED
INTERWORKING_UNSPECIFIED
OUTGOING_CALLS_BARRED_WITHIN_CUG
NO_CUG_SELECTION
UNKNOWN_CUG_INDEX
CUG_INDEX_INCOMPATIBLE
CUG_CALL_FAILURE_UNSPECIFIED
CLIR_NOT_SUBSCRIBED
CCBS_POSSIBLE
CCBS_NOT_POSSIBLE
IMSI_UNKNOWN_IN_HLR
ILLEGAL_MS
IMSI_UNKNOWN_IN_VLR
IMEI_NOT_ACCEPTED
ILLEGAL_ME
PLMN_NOT_ALLOWED
LOCATION_AREA_NOT_ALLOWED
ROAMING_NOT_ALLOWED_IN_THIS_LOCATION_AREA
NO_SUITABLE_CELLS_IN_LOCATION_AREA
NETWORK_FAILURE
MAC_FAILURE
SYNCH_FAILURE
NETWORK_CONGESTION
GSM_AUTHENTICATION_UNACCEPTABLE
SERVICE_NOT_SUBSCRIBED

SERVICE_TEMPORARILY_OUT_OF_ORDER
CALL_CANNOT_BE_IDENTIFIED
INCORRECT_SEMANTICS_IN_MESSAGE
MANDATORY_INFORMATION_INVALID
ACCESS_STRATUM_FAILURE
INVALID_SIM
WRONG_STATE
ACCESS_CLASS_BLOCKED
NO_RESOURCES
INVALID_USER_DATA
TIMER_T3230_EXPIRED
NO_CELL_AVAILABLE
ABORT_MSG_RECEIVED
RADIO_LINK_LOST
TIMER_T303_EXPIRED
CNM_MM_REL_PENDING
ACCESS_STRATUM_REJ_RR_REL_IND
ACCESS_STRATUM_REJ_RR_RANDOM_ACCESS_FAILURE
ACCESS_STRATUM_REJ_RRC_REL_IND
ACCESS_STRATUM_REJ_RRC_CLOSE_SESSION_IND
ACCESS_STRATUM_REJ_RRC_OPEN_SESSION_FAILURE
ACCESS_STRATUM_REJ_LOW_LEVEL_FAIL
ACCESS_STRATUM_REJ_LOW_LEVEL_FAIL_REDIAL_NOT_ALLOWED
ACCESS_STRATUM_REJ_LOW_LEVEL_IMMED_RETRY
ACCESS_STRATUM_REJ_ABORT_RADIO_UNAVAILABLE
SERVICE_OPTION_NOT_SUPPORTED
ACCESS_STRATUM_REJ_CONN_EST_FAILURE_ACCESS_BARRED
ACCESS_STRATUM_REJ_CONN_REL_NORMAL
ACCESS_STRATUM_REJ_UL_DATA_CNF_FAILURE_CONN_REL
BAD_REQ_WAIT_INVITE
BAD_REQ_WAIT_REINVITE
INVALID_REMOTE_URI
REMOTE_UNSUPP_MEDIA_TYPE
PEER_NOT_REACHABLE
NETWORK_NO_RESP_TIME_OUT
NETWORK_NO_RESP_HOLD_FAIL
DATA_CONNECTION_LOST
UPGRADE_DOWNGRADE_REJ
SIP_403_FORBIDDEN
NO_NETWORK_RESP
UPGRADE_DOWNGRADE_FAILED
UPGRADE_DOWNGRADE_CANCELLED
SSAC_REJECT
THERMAL_EMERGENCY
FAILURE_1XCSFB_SOFT
FAILURE_1XCSFB_HARD
CONNECTION_EST_FAILURE
CONNECTION_FAILURE
RRC_CONN_REL_NO_MT_SETUP
ESR_FAILURE

MT_CSFB_NO_RESPONSE_FROM_NW
BUSY_EVERYWHERE
ANSWERED_ELSEWHERE
RLF_DURING_CC_DISCONNECT
TEMP_REDIAL_ALLOWED
PERM_REDIAL_NOT_NEEDED
MERGED_TO_CONFERENCE
LOW_BATTERY
CALL_DEFLECTED
RTP_RTCP_TIMEOUT
RINGING_RINGBACK_TIMEOUT
REG_RESTORATION
CODEC_ERROR
UNSUPPORTED_SDP
RTP_FAILURE
QoS_FAILURE
MULTIPLE_CHOICES
MOVED_PERMANENTLY
MOVED_TEMPORARILY
USE_PROXY
ALTERNATE_SERVICE
ALTERNATE_EMERGENCY_CALL
UNAUTHORIZED
PAYMENT_REQUIRED
METHOD_NOT_ALLOWED
NOT_ACCEPTABLE
PROXY_AUTHENTICATION_REQUIRED
GONE
REQUEST_ENTITY_TOO_LARGE
REQUEST_URI_TOO_LARGE
UNSUPPORTED_URI_SCHEME
BAD_EXTENSION
EXTENSION_REQUIRED
INTERVAL_TOO_BRIEF
CALL_OR_TRANS_DOES_NOT_EXIST
LOOP_DETECTED
TOO_MANY_HOPS
ADDRESS_INCOMPLETE
AMBIGUOUS
REQUEST_TERMINATED
NOT_ACCEPTABLE_HERE
REQUEST_PENDING
UNDECIPHERABLE
SERVER_INTERNAL_ERROR
NOT_IMPLEMENTED
BAD_GATEWAY
SERVER_TIME_OUT
VERSION_NOT_SUPPORTED
MESSAGE_TOO_LARGE
DOES_NOT_EXIST_ANYWHERE

SESS_DESCR_NOT_ACCEPTABLE
SRVCC_END_CALL
INTERNAL_ERROR
SERVER_UNAVAILABLE
PRECONDITION_FAILURE
DRVCC_IN_PROG
DRVCC_END_CALL
CS_HARD_FAILURE
CS_ACQ_FAILURE
REJECTED_ELSEWHERE
CALL_PULLED
CALL_PULL_OUT_OF_SYNC
HOLD_RESUME_FAILED
HOLD_RESUME_CANCELED
REINVITE_COLLISION
REDIAL_SECONDARY_LINE_CS
REDIAL_SECONDARY_LINE_PS
REDIAL_SECONDARY_LINE_CS_AUTO
REDIAL_SECONDARY_LINE_PS_AUTO

4.53 Cellular V2X

This section contains APIs related to Cellular-V2X operation. The SDK has C and C++ APIs. The C APIs are legacy and are being maintained for backwards compatibility purposes. These APIs are mostly wrappers over the C++ APIs. New features and functionality will be added to C++ APIs. For anyone writing new Cv2x software, it is recommended to use the C++ APIs.

5 Namespace Documentation

5.1 telux Namespace Reference

DeviceConfig provides utility functions to get device configuration details such as multi SIM support.

Namespaces

- audio
- common
- config
- cv2x
- data
- loc
- platform
- power
- sec
- sensor
- tel
- therm
- wlan

DeviceConfig provides utility functions to get device configuration details such as multi SIM support.

Cv2x Rx packets helper class - Help to parse the received packets' meta data at the beginning of the payload, any applications that have enabled the received packets' meta data should use this helper class before using the real payload.

5.2 telux::audio Namespace Reference

Data Structures

- struct AmrwbpParams
- class AudioFactory
- struct ChannelVolume

- struct DtmfTone
- struct FormatInfo
- struct FormatParams
- class IAudioBuffer
- class IAudioCaptureStream
- class IAudioDevice
- class IAudioListener
- class IAudioLoopbackStream
- class IAudioManager
- class IAudioPlayStream
- class IAudioStream
- class IAudioToneGeneratorStream
- class IAudioVoiceStream
- class IPlayListener
- class IStreamBuffer
- class ITranscodeListener
- class ITranscoder
- class IVoiceListener
- struct StreamConfig
- struct StreamMute
- struct StreamVolume

5.2.1 Variable Documentation

5.2.1.1 const uint16_t telux::audio::INFINITE_DTMF_DURATION = 0xFFFF

Specifies that the DTMF tone should be played indefinitely

5.2.1.2 const uint16_t telux::audio::INFINITE_TONE_DURATION = 0xFFFF

Specifies that the audio tone should be played indefinitely

5.3 telux::common Namespace Reference

Data Structures

- class DeviceConfig
- class ICommandCallback

- class ICommandResponseCallback

  General command response callback for most of the requests, client needs to implement this interface to get single shot response. More...

- class IServiceStatusListener
- class Log
- struct SdkVersion
- class Version

  Provides version of SDK. More...

5.3.1 Typedef Documentation

5.3.1.1 using telux::common::ResponseCallback = typedef std::function<void(telux::common::ErrorCode errorCode)>

General response callback for most of the requests, client needs to implement this function to get the asynchronous response.

The methods in callback can be invoked from multiple different threads. The implementation should be thread safe.

Parameters

in	*errorCode*	ErrorCode

5.3.1.2 using telux::common::InitResponseCb = typedef std::function<void(telux::common::ServiceStatus status)>

This API is invoked when the initialization of an object completes.

Parameters

in	*status*	- ServiceStatus

5.4 telux::config Namespace Reference

Data Structures

- class ConfigFactory

  ConfigFactory allows creation of config related classes. More...

- struct ConfigInfo
- class IConfigListener

  IConfigListener interface is used to receive notifications related to any updates in the configurations dynamically. More...

- class IConfigManager

IConfigManager provides APIs to retrieve an instance of the manager, APIs for processes to update and retrieve configurations dynamically. More...

- class IModemConfigListener

  Listener class for getting notifications related to configuration change detection. The client needs to implement these methods as briefly as possible and avoid blocking calls in it. The methods in this class can be invoked from multiple different threads. Client needs to make sure that the implementation is thread-safe. More...

- class IModemConfigManager

  IModemConfigManager provides interface to list config files present in modem's storage. load a new config file in modem, activate a config file, get active config file information, deactivate a config file, delete config file from the modem's storage, get and set mode of config auto selection, register and deregister listener for config update in modem. The config files are also referred to as MBNs. More...

5.5 telux::cv2x Namespace Reference

Data Structures

- struct ConfigEventInfo
- class Cv2xFactory

  Cv2xFactory is the factory that creates the Cv2x Radio. More...

- struct Cv2xPoolStatus
- struct Cv2xRadioCapabilities
- class Cv2xRxMetaDataHelper
- struct Cv2xStatus
- struct Cv2xStatusEx
- class Cv2xUtil
- struct DataSessionSettings
- struct EventFlowInfo
- struct GlobalIPUnicastRoutingInfo
- class ICv2xConfig

  Cv2xConfig provide operations to update or request cv2x configuration. More...

- class ICv2xConfigListener

  Listeners for ICv2xConfig must implement this interface. More...

- class ICv2xListener

  Cv2x Radio Manager listeners implement this interface.

- class ICv2xRadio
- class ICv2xRadioListener

  Listeners for Cv2xRadio must implement this interface. More...

- class ICv2xRadioManager

  Cv2xRadioManager manages instances of Cv2xRadio. More...

- class ICv2xRxSubscription
- class ICv2xThrottleManager

  ThrottleManager provides throttle manager client interface. More...

- class ICv2xThrottleManagerListener

  Listener class for getting filter rate update notification. More...

- class ICv2xTxFlow
- class ICv2xTxRxSocket
- class ICv2xTxStatusReportListener

  Listeners for CV2X Tx status report must implement this interface. More...

- struct IPv6Address
- struct IPv6AddrType
- struct L2FilterInfo
- struct MacDetails
- struct RFTxInfo
- struct RxPacketMetaDataReport
- struct SlssRxInfo
- struct SocketInfo
- struct SpsFlowInfo
- struct SpsSchedulingInfo
- struct SyncRefUeInfo
- struct TrustedUEInfo
- struct TrustedUEInfoList
- struct TxPoolIdInfo
- struct TxStatusReport

5.5.1 Typedef Documentation

5.5.1.1 using telux::cv2x::CreateRxSubscriptionCallback = typedef std::function<void (std::shared_ptr<ICv2xRxSubscription> rxSub, telux::common::ErrorCode error)>

This function is called as a response to ICv2xRadio::createRxSubscription.

Parameters

in	*rxSub*	- Rx Subscription
in	*error*	- Indicates whether socket creation succeeded • SUCCESS • GENERIC_FAILURE

5.5.1.2 using telux::cv2x::CreateTxSpsFlowCallback = typedef std::function<void (std::shared_ptr<ICv2xTxFlow> txSpsFlow, std::shared_ptr<ICv2xTxFlow> txEventFlow, telux::common::ErrorCode spsError, telux::common::ErrorCode eventError)>

This function is called as a response to ICv2xRadio::createTxSpsFlow

Parameters

in	*txSpsFlow*	- Sps flow
in	*txEventFlow*	- Optional event flow. Will be nullptr if event flow was not specified in the request
in	*spsError*	- Indicates whether Tx SPS flow creation succeeded • SUCCESS • GENERIC_FAILURE
in	*eventError*	- Indicates whether optional Tx Event flow creation succeeded • SUCCESS • GENERIC_FAILURE

5.5.1.3 using telux::cv2x::CreateTxEventFlowCallback = typedef std::function<void (std::shared_ptr<ICv2xTxFlow> txEventFlow, telux::common::ErrorCode error)>

This function is called with the response to ICv2xRadio::createTxEventFlow

Parameters

in	*txEventFlow*	- Event flow
in	*error*	- Indicates whether Tx event flow creation succeeded • SUCCESS • GENERIC_FAILURE

5.5.1.4 using telux::cv2x::CloseTxFlowCallback = typedef std::function<void (std::shared_ptr<ICv2xTxFlow> txFlow, telux::common::ErrorCode error)>

This function is called with the response to ICv2xRadio::closeTxFlow.

Parameters

in	*txFlow*	- Closed tx flow
in	*error*	- Indicates whether close operation succeeded • SUCCESS • GENERIC_FAILURE

5.5.1.5 using telux::cv2x::CloseRxSubscriptionCallback = typedef std::function<void (std::shared_ptr<ICv2xRxSubscription> rxSub, telux::common::ErrorCode error)>

This function is called with the response to ICv2xRadio::closeRxSubscription.

Parameters

in	*rxSub*	- Closed rx subscription
in	*error*	- Indicates whether Rx subscription close succeeded • SUCCESS • GENERIC_FAILURE

5.5.1.6 using telux::cv2x::ChangeSpsFlowInfoCallback = typedef std::function<void (std::shared_ptr<ICv2xTxFlow> txFlow, telux::common::ErrorCode error)>

This function is called with the response to ICv2xRadio::changeSpsFlowInfo.

Parameters

in	*txFlow*	- Sps flow that requested reservation change
in	*error*	- SUCCESS if Tx reservation change succeeded • SUCCESS • GENERIC_FAILURE

5.5.1.7 using telux::cv2x::RequestSpsFlowInfoCallback = typedef std::function<void (std::shared_ptr<ICv2xTxFlow> txFlow, const SpsFlowInfo & spsInfo, telux::common::ErrorCode error)>

This function is called with the response to ICv2xRadio::requestSpsFlowInfo.

Parameters

in	*txFlow*	- SPS flow that requested info
in	*spsInfo*	- SPS flow reservation info
in	*error*	- SUCCESS if Tx reservation change succeeded • SUCCESS • GENERIC_FAILURE

5.5.1.8 using telux::cv2x::ChangeEventFlowInfoCallback = typedef std::function<void (std::shared_ptr<ICv2xTxFlow> txFlow, telux::common::ErrorCode error)>

This function is called with the response to ICv2xRadio::changeEventFlowInfo.

Parameters

in	*txFlow*	- Event flow that requested reservation change
in	*error*	- SUCCESS if Tx parameter change succeeded • SUCCESS • GENERIC_FAILURE

5.5.1.9 using telux::cv2x::RequestCapabilitiesCallback = typedef std::function<void(const Cv2xRadioCapabilities & capabilities, telux::common::ErrorCode error)>

This function is called with the response to ICv2xRadio::requestCapabilities.

Parameters

in	*capabilities*	- Capability info
in	*error*	- SUCCESS if capabilities request succeeded • SUCCESS • GENERIC_FAILURE

5.5.1.10 using telux::cv2x::RequestDataSessionSettingsCallback = typedef std::function<void (const DataSessionSettings & settings, telux::common::↩ ErrorCode error)>

This function is called with the response to ICv2xRadio::requestDataSessionSettings.

Parameters

in	*settings*	- Data session settings
in	*error*	- SUCCESS if data session settings request succeeded • SUCCESS • GENERIC_FAILURE

5.5.1.11 using telux::cv2x::UpdateTrustedUEListCallback = typedef std::function<void(telux::common::ErrorCode error)>

This function is called with the response to ICv2xRadio::updateTrustedUEList.

Parameters

in	*error*	- SUCCESS if update succeeded • INVALID_ARGUMENTS if trustedUEs or maliciousIds length greater than maximum value • SUCCESS • GENERIC_FAILURE • INVALID_ARGUMENTS

5.5.1.12 using telux::cv2x::UpdateSrcL2InfoCallback = typedef std::function<void (telux::common::ErrorCode error)>

This function is called with the response to ICv2xRadio::updateSrcL2Info.

Parameters

in	*error*	- SUCCESS if Tx reservation change succeeded • SUCCESS • GENERIC_FAILURE

5.5.1.13 using telux::cv2x::CreateTcpSocketCallback = typedef std::function<void (std::shared_ptr<ICv2xTxRxSocket> sock, telux::common::ErrorCode error)>

This function is called with the response to ICv2xRadio::createCv2xTcpSocket.

Parameters

in	*sock*	- TCP socket
in	*error*	- Indicates whether TCP socket creation succeeded • SUCCESS • GENERIC_FAILUREs

5.5.1.14 using telux::cv2x::CloseTcpSocketCallback = typedef std::function<void (std::shared_ptr<ICv2xTxRxSocket> sock, telux::common::ErrorCode error)>

This function is called with the response to ICv2xRadio::closeCv2xTcpSocket.

Parameters

in	*sock*	- Closed TCP socket
in	*error*	- Indicates whether close operation succeeded • SUCCESS • GENERIC_FAILURE

5.5.1.15 using telux::cv2x::StartCv2xCallback = typedef std::function<void (telux::common::ErrorCode error)>

This function is called as a response to ICv2xRadioManager::startCv2x

Parameters

in	*error*	- SUCCESS if Cv2x mode successfully started • SUCCESS • GENERIC_FAILURE

5.5.1.16 using telux::cv2x::StopCv2xCallback = typedef std::function<void (telux::common::ErrorCode error)>

This function is called as a response to ICv2xRadioManager::stopCv2x

Parameters

in	*error*	- SUCCESS if Cv2x mode successfully stopped • SUCCESS • GENERIC_FAILURE

5.5.1.17 using telux::cv2x::RequestCv2xStatusCallback = typedef std::function<void (Cv2xStatus status, telux::common::ErrorCode error)>

This function is called as a response to ICv2xRadioManager::requestCv2xStatus

Parameters

in	*status*	- Cv2x status
in	*error*	- SUCCESS if Cv2x status was successully retrieved • SUCCESS • GENERIC_FAILURE

Deprecated

use RequestCv2xStatusCallbackEx

5.5.1.18 using telux::cv2x::RequestCv2xStatusCallbackEx = typedef std::function<void (Cv2xStatusEx status, telux::common::ErrorCode error)>

This function is called as a response to ICv2xRadioManager::requestCv2xStatus

Parameters

in	*status*	- Cv2x status
in	*error*	- SUCCESS if Cv2x status was successully retrieved • SUCCESS • GENERIC_FAILURE

5.5.1.19 using telux::cv2x::UpdateConfigurationCallback = typedef std::function<void (telux::common::ErrorCode error)>

This function is called with the response to ICv2xRadioManager::updateConfiguration

Parameters

in	*error*	- SUCCESS if configuration was updated successfully • SUCCESS • GENERIC_FAILURE

5.5.1.20 using telux::cv2x::GetSlssRxInfoCallback = typedef std::function<void (const SlssRxInfo& info, telux::common::ErrorCode error)>

This function is called as a response to ICv2xRadioManager::getCv2xSlssRxInfo

Parameters

out	*info*	- Cv2x SLSS Rx Information
out	*error*	- SUCCESS if Cv2x SLSS Rx Information was successully retrieved • SUCCESS • GENERIC_FAILURE

5.6 telux::data Namespace Reference

Namespaces

- net

Data Structures

- struct BandInterferenceConfig
- struct BitRateInfo
- struct DataCallEndReason
- union DataCallEndReason.__unnamed__
- struct DataCallStats
- class DataFactory

  DataFactory is the central factory to create all data classes. More...
- class DataProfile

  DataProfile class represents single data profile on the modem. More...
- struct DataRestrictMode
- struct DdsInfo
- struct EspInfo
- struct FlowDataRate
- struct IcmpInfo
- class IDataCall

  Represents single established data call on the device. More...
- class IDataConnectionListener
- class IDataConnectionManager

  IDataConnectionManager is a primary interface for cellular connectivity This interface provides APIs for start and stop data call connections, get data call information and listener for monitoring data calls. It also provides interface to Subsystem Restart events by registering as listener. Notifications will be received when modem is ready/not ready. More...
- class IDataCreateProfileCallback
- class IDataFilterListener

  Listener class for listening to filtering mode notifications, like Data filtering mode change. Client need to implement these methods. The methods in listener can be invoked from multiple threads. So the client needs to make sure that the implementation is thread-safe. More...
- class IDataFilterManager

  IDataFilterManager class provides interface to enable/disable the data restrict filters and register for data restrict filter. The filtering can be done at any time. One such use case is to do it when we want the AP to suspend so that we are not waking up the AP due to spurious incoming messages. Also to make sure the DataRestrict mode is enabled. More...
- class IDataProfileCallback
- class IDataProfileListCallback

  Interface for getting list of DataProfile using callback. Client needs to implement this interface to get single shot responses for commands like get profile list and query profile. More...

- class IDataProfileListener

  Listener class for getting profile change notification. More...

- class IDataProfileManager
- class IDataSettingsListener
- class IDataSettingsManager

  Data Settings Manager class provides APIs related to the data subsystem settings. For example, ability to reset current network settings to factory settings, setting backhaul priority, and enabling roaming per PDN. More...

- class IEspFilter

  This class represents a IP Filter for the ESP, get the new instance from telux::data::DataFactory. More...

- class IIcmpFilter

  This class represents a IP Filter for the ICMP, get the new instance from telux::data::DataFactory. More...

- class IIpFilter

  A IP filter class to add specific filters like what data will be allowed from the modem to the application processor. Only data packets that match the filter will be sent to the apps processor. Also used to configure Firewall rules. More...

- struct IpAddrInfo
- struct IpFamilyInfo
- struct IPv4Info
- struct IPv6Info
- class IServingSystemListener

  Listener class for data serving system change notification. More...

- class IServingSystemManager

  Serving System Manager class provides APIs related to the serving system for data functionality. For example, ability to query or be notified about the state of the platform's WWAN PS data serving information. More...

- class ITcpFilter

  This class represents a IP Filter for the TCP, get the new instance from telux::data::DataFactory. More...

- class IUdpFilter

  This class represents a IP Filter for the UDP, get the new instance from telux::data::DataFactory. More...

- struct PortInfo
- struct ProfileParams
- struct QosFilterRule
- struct QosIPFlowInfo
- struct RoamingStatus

  Roaming Status. More...

- struct ServiceStatus

  Data Service Status Info. More...

- struct TcpInfo
- struct TftChangeInfo
- struct TrafficFlowTemplate
- struct UdpInfo
- struct VlanConfig

5.7 telux::data::net Namespace Reference

Data Structures

- struct BridgeInfo
- class IBridgeListener
- class IBridgeManager

  IBridgeManager provides APIs to enable/disable and set/get/delete software bridges for various WLAN and Ethernet interfaces. It also provides interface to Subsystem Restart events by registering as listener. Notifications will be received when modem is ready/not ready. More...

- class IFirewallEntry

  Firewall entry class is used for configuring firewall rules. More...

- class IFirewallListener
- class IFirewallManager

  FirewallManager is a primary interface that filters and controls the network traffic on a pre-configured set of rules. It also provides interface to Subsystem Restart events by registering as listener. Notifications will be received when modem is ready/not ready. More...

- class IL2tpListener
- class IL2tpManager

  L2tpManager is a primary interface for configuring L2TP Service. It also provides interface to Subsystem Restart events by registering as listener. Notifications will be received when modem is ready/not ready. More...

- class INatListener
- class INatManager

  NatManager is a primary interface for configuring static network address translation(SNAT) and DMZ (demilitarized zone). It also provides interface to Subsystem Restart events by registering as listener. Notifications will be received when modem is ready/not ready. More...

- class ISocksListener
- class ISocksManager

  SocksManager is a primary interface for configuring legacy Socks proxy server. It also provides interface to Subsystem Restart events by registering as listener. Notifications will be received when modem is ready/not

ready. More...

- class IVlanListener
- class IVlanManager

  VlanManager is a primary interface for configuring VLAN (Virtual Local Area Network). it provide APIs for create, query, remove VLAN interfaces and associate or disassociate with profile IDs. It also provides interface to Subsystem Restart events by registering as listener. Notifications will be received when modem is ready/not ready. More...

- struct L2tpSessionConfig
- struct L2tpSysConfig
- struct L2tpTunnelConfig
- struct NatConfig

5.8 telux::loc Namespace Reference

Data Structures

- struct BodyToSensorMountParams
- struct DREngineConfiguration
- struct GlonassTimeInfo
- struct GnssData
- struct GnssDisasterCrisisReport
- struct GnssEnergyConsumedInfo
- struct GnssKinematicsData
- struct GnssMeasurementInfo
- struct GnssMeasurements
- struct GnssMeasurementsClock
- struct GnssMeasurementsData
- class IDgnssManager

  IRtcmManager provides interface to inject RTCM data into modem, register event listener reported by cdfw(correction data framework). More...

- class IDgnssStatusListener

  Listener class for getting RTCM injection event notification information. More...

- class IGnssSignalInfo

  IGnssSignalInfo provides interface to retrieve GNSS data information like jammer metrics and automatic gain control for satellite signal type. More...

- class IGnssSVInfo

  IGnssSVInfo provides interface to retrieve the list of SV info available and whether altitude is assumed or

calculated. More...

- class ILocationConfigListener

  ILocationConfigListener interface is used to receive notifications related to configuration events. More...

- class ILocationConfigurator

  ILocationConfigurator allows general engine configurations (example: TUNC, PACE etc), configuration of specific engines like SPE (example: minSVElevation, minGPSWeek etc) or DRE, deletion of warm and cold aiding data, NMEA configuration and support for XTRA feature. ILocationConfigurator APIs strictly adheres to the principle of single client per process. More...

- class ILocationInfoBase

  ILocationInfoBase provides interface to get basic position related information like latitude, longitude, altitude, timestamp. More...

- class ILocationInfoEx

  ILocationInfoEx provides interface to get richer position related information like latitude, longitude, altitude and other information like time stamp, session status, dop, reliabilities, uncertainities etc. More...

- class ILocationListener

  Listener class for getting location updates and satellite vehicle information. More...

- class ILocationManager

  ILocationManager provides interface to register and remove listeners. It also allows to set and get configuration/ criteria for position reports. The new APIs(registerListenerEx, deRegisterListenerEx, startDetailedReports, startBasicReports) and old/deprecated APIs(registerListener, removeListener, setPositionReportTimeout, setHorizontalAccuracyLevel, setMinIntervalForReports) should not be used interchangebly, either the new APIs should be used or the old APIs should be used. More...

- class ILocationSystemInfoListener
- class ISVInfo

  ISVInfo provides interface to retrieve information about Satellite Vehicles, their position and health status. More...

- struct LeapSecondChangeInfo
- struct LeapSecondInfo
- struct LeverArmParams
- struct LLAInfo
- class LocationFactory

  LocationFactory allows creation of location manager. More...

- struct LocationSystemInfo
- struct NmeaConfig
- struct RobustLocationConfiguration
- struct RobustLocationVersion
- struct SvBlackListInfo
- struct SvUsedInPosition

- struct SystemTime
- union SystemTimeInfo
- struct TimeInfo
- struct XtraConfig
- struct XtraStatus

5.9 telux::platform Namespace Reference

Data Structures

- struct EfsEventInfo
- class IDeviceInfoListener

  Listener class for getting device info related notifications . The client needs to implement these methods as briefly as possible and avoid blocking calls in it. The methods in this class can be invoked from multiple different threads. Client needs to make sure that the implementation is thread-safe. More...

- class IDeviceInfoManager

  IDeviceInfoManager provides interface to to retrieve IMEI and platform version operations. More...

- class IFsListener

  Listener class for getting notifications related to EFS backup/restore operations. The client needs to implement these methods as briefly as possible and avoid blocking calls in it. The methods in this class can be invoked from multiple different threads. Client needs to make sure that the implementation is thread-safe. More...

- class IFsManager

  IFsManager provides interface to to control and get notified about file system operations. This includes Embedded file system (EFS) operations. More...

- class PlatformFactory

  PlatformFactory allows creation of Platform services related classes. More...

- struct PlatformVersion

5.10 telux::power Namespace Reference

Data Structures

- struct ClientInstanceConfig
- class ITcuActivityListener

  Listener class for getting notifications related to TCU-activity state and also the updates related to TCU-activity service status. The client needs to implement these methods as briefly as possible and avoid blocking calls in it. The methods in this class can be invoked from multiple different threads. Client needs to make sure that the implementation is thread-safe. More...

- class ITcuActivityManager

ITcuActivityManager provides interface to register and de-register listeners to get TCU-activity state updates. And also API to initiate TCU-activity state transition. More...

- class PowerFactory

  PowerFactory allows creation of TCU-activity manager instance. More...

5.11 telux::sec Namespace Reference

Data Structures

- class CryptoParamBuilder
- struct DataDigest
- struct ECCPoint
- struct EncryptedData
- class ICryptoAcceleratorListener
- class ICryptoAcceleratorManager
- class ICryptoManager

  ICryptoManager provides key management and crypto operation support. It uses trusted hardware bound cryptography. All keys generated are bound to the device cryptographically. More...

- class ICryptoParam
- struct OperationResult
- class ResultParser
- struct Scalar
- class SecurityFactory

  SecurityFactory allows creation of ICryptoManager and ICryptoAcceleratorManager. More...

- struct Signature

5.12 telux::sensor Namespace Reference

Data Structures

- class ISensorClient

  ISensorClient interface is used to access the different services provided by the sensor framework to configure, activate and acquire sensor data. More...

- class ISensorEventListener

  ISensorEventListener interface is used to receive notifications related to sensor events and configuration updates. More...

- class ISensorFeatureEventListener

  ISensorFeatureEventListener interface is used to receive notifications related to sensor feature events. More...

- class ISensorFeatureManager

  Sensor Feature Manager class provides APIs to interact with the sensor framework to list the available features, enable them or disable them. The availability of sensor features depends on the capabilities of the underlying hardware. More...

- class ISensorManager

  Sensor Manager class provides APIs to interact with the sensor sub-system and get access to other sensor objects which can be used to configure, activate or get data from the individual sensors available - Gyro, Accelero, etc. More...

- struct MotionSensorData

  Structure of a single sample from a motion sensor. More...

- struct SensorConfiguration

  Configurable parameters of a sensor. More...

- struct SensorEvent

  Structure of a single sensor event. More...

- union SensorEvent.__unnamed__

- class SensorFactory

  SensorFactory is the central factory to create instances of sensor objects. More...

- struct SensorFeature

  Feature offered by sensor hardware and/or software framework. More...

- struct SensorFeatureEvent

  Structure of an event that is generated from a sensor feature. More...

- struct SensorInfo

  Information related to sensor. More...

- struct UncalibratedMotionSensorData

  Structure of a single sample from uncalibrated motion sensor. More...

5.13 telux::tel Namespace Reference

Data Structures

- struct CardReaderStatus
- class CdmaCellIdentity
- class CdmaCellInfo
- class CdmaSignalStrengthInfo
- struct CellBroadcastFilter
- class CellBroadcastMessage

  Cell Broadcast message. More...

- class CellInfo
- struct CellularCapabilityInfo
- class Circle
- class CmasInfo
- struct CustomHeader
- struct CustomSipHeader
- struct DcStatus
- struct DeleteInfo

  Specify delete information used for deleting message on storage. More...
- struct EcallConfig
- struct ECallHlapTimerEvents
- struct ECallHlapTimerStatus
- struct ECallModeInfo
- struct ECallMsdControlBits
- struct ECallMsdData
- struct ECallMsdOptionals
- struct ECallOptionalPdu
- struct ECallVehicleIdentificationNumber
- struct ECallVehicleLocation
- struct ECallVehicleLocationDelta
- struct ECallVehiclePropulsionStorageType
- class EtwsInfo
- struct FileAttributes
- struct ForwardInfo
- struct ForwardReq
- class Geometry
- class GsmCellIdentity
- class GsmCellInfo
- class GsmSignalStrengthInfo
- class IAtrResponseCallback
- class ICall

  ICall represents a call in progress. An ICall cannot be directly created by the client, rather it is returned as a result of instantiating a call or from the PhoneListener when receiving an incoming call. More...
- class ICallListener

A listener class for monitoring changes in call, including call state change and ECall state change. Override the methods for the state that you wish to receive updates for. More...

- class ICallManager

  Call Manager is the primary interface for call related operations Allows to conference calls, swap calls, make normal voice call and emergency call, send and update MSD pdu. More...

- class ICard

  ICard represents currently inserted UICC or eUICC. More...

- class ICardApp

  Represents a single card application. More...

- class ICardChannelCallback
- class ICardCommandCallback
- class ICardFileHandler

  ICardFileHandler provides APIs for reading from an elementary file(EF) on SIM and writing to EF on SIM. Provide API to get EF attributes like file size, record size, and the number of records in EF. More...

- class ICardListener
- class ICardManager
- class ICardReaderCallback
- struct IccResult
- class ICellBroadcastListener

  A listener class which monitors cell broadcast messages. More...

- class ICellBroadcastManager

  CellBroadcastManager class is primary interface to configure and activate emergency broadcast messages and receive broadcast messages. More...

- class ICellularCapabilityCallback
- class IEcallListener

  Listener class to notify service status change notifications. The listener method can be invoked from multiple different threads. Client needs to make sure that implementation is thread-safe. More...

- class IEcallManager

  IEcallManager allows operations related to automotive emergency call management and its related configurations. More...

- class IHttpTransactionListener

  The interface listens for indication to perform HTTP request and send back the response for HTTP request to modem. More...

- class IHttpTransactionManager

  IHttpTransactionManager is the interface to service HTTP related requests from the modem for SIM profile update related operations. More...

- class IImsServingSystemListener

A listener class for monitoring changes in IMS Serving System manager, including IMS registration status change. Override the methods for the state that you wish to receive updates for.

- class IImsServingSystemManager

  IMS Serving System Manager is the primary interface for IMS related operations Allows to query IMS registration status.

- class IImsSettingsListener

  Listener class for getting IMS service configuration change notifications. The listener method can be invoked from multiple different threads. Client needs to make sure that implementation is thread-safe. More...

- class IImsSettingsManager

  ImsSettingsManager allows IMS settings. For example enabling or disabling IMS service, VOIMS service. More...

- class IMakeCallCallback

  Interface for Make Call callback object. Client needs to implement this interface to get single shot responses for commands like make call. More...

- struct ImsRegistrationInfo
- struct ImsServiceConfig
- struct ImsServiceInfo
- class IMultiSimListener

  Listener class for getting high capability change notification. The listener method can be invoked from multiple different threads. Client needs to make sure that implementation is thread-safe. More...

- class IMultiSimManager

  MultiSimManager allows to perform operation pertaining to devices which have more than one SIM/UICC card. Clients should check if the subsystem is ready before invoking any of the APIs as follows. More...

- class INetworkSelectionListener

  Listener class for getting network selection mode change notification. More...

- class INetworkSelectionManager

  Network Selection Manager class provides the interface to get and set network selection mode, preferred network list and scan available networks. More...

- class IOperatingModeCallback
- class IPhone

  This class allows getting system information and registering for system events. Each Phone instance is associated with a single SIM. So on a dual SIM device you would have 2 Phone instances. More...

- class IPhoneListener

  A listener class for monitoring changes in specific telephony states on the device, including service state and signal strength. Override the methods for the state that you wish to receive updates for. More...

- class IPhoneManager

  Phone Manager creates one or more phones based on SIM slot count, it allows clients to register for notification of system events. Clients should check if the subsystem is ready before invoking any of the APIs.

More...

- class IRemoteSimListener

  A listener class for getting remote SIM notifications. More...

- class IRemoteSimManager

  IRemoteSimManager provides APIs for remote SIM related operations. This allows a device to use a SIM card on another device for its WWAN modem functionality. The SIM provider service is the endpoint that interfaces with the SIM card (e.g. over bluetooth) and sends/receives data to the other endpoint, the modem. The modem sends requests to the SIM provider service to interact with the SIM card (e.g. power up, transmit APDU, etc.), and is notified of events (e.g. card errors, resets, etc.). This API is used by the SIM provider endpoint to provide a SIM card to the modem. More...

- class ISapCardCommandCallback
- class ISapCardListener
- class ISapCardManager

  ISapCardManager provide APIs for SAP related operations. More...

- class IServingSystemListener

  Listener class for getting notifications related to updates in radio access technology mode preference, service domain preference, serving system information, etc. Some notifications in this listener could be frequent in nature. When the system is in a suspended/low power state, those indications will wake the system up. This could result in increased power consumption by the system. If those notifications are not required in the suspended/low power state, it is recommended for the client to de-register specific notifications using the deregisterListener API. More...

- class IServingSystemManager
- class ISignalStrengthCallback

  Interface for Signal strength callback object. Client needs to implement this interface to get single shot responses for commands like get signal strength. More...

- class ISimProfileListener

  The interface listens for profile download indication and keep track of download and install progress of profile. More...

- class ISimProfileManager

  ISimProfileManager is a primary interface for remote eUICCs (eSIMs or embedded SIMs) provisioning.This interface provides APIs to add, delete, set profile, update nickname, provide user consent, get Eid on the eUICC. More...

- class ISmscAddressCallback
- class ISmsListener

  A listener class receives notification for the incoming message(s) and delivery report for sent message(s). More...

- class ISmsManager

  SmsManager class is the primary interface to manage SMS operations such as send and receive an SMS text and raw encoded PDU(s). This class handles single part and multi-part messages. More...

- class ISubscription

Subscription returns information about network operator subscription details pertaining to a SIM card. More...

- class ISubscriptionListener

  A listener class for receiving device subscription information. The methods in listener can be invoked from multiple different threads. The implementation should be thread safe. More...

- class ISubscriptionManager
- class ISuppServicesListener

  A listener class for receiving supplementary services notifications. The methods in listener can be invoked from multiple different threads. The implementation should be thread safe. More...

- class ISuppServicesManager

  ISuppServicesManager is the interface to provide supplementary services like call forwarding and call waiting. More...

- class IVoiceServiceStateCallback

  Interface for voice service state callback object. Client needs to implement this interface to get single shot responses for commands like request voice radio technology. More...

- class LteCellIdentity
- class LteCellInfo
- class LteSignalStrengthInfo
- struct MessageAttributes

  Contains structure of message attributes like encoding type, number of segments, characters left in last segment. More...

- struct MessagePartInfo

  Structure containing information about the part of multi-part SMS such as concatenated message reference number, number of segments and segment number. During concatenation this information along with originating address helps in associating each part of the multi-part message to the corresponding multi-part message. More...

- struct NetworkScanInfo
- struct NetworkTimeInfo
- class Nr5gCellIdentity
- class Nr5gCellInfo
- class Nr5gSignalStrengthInfo
- class OperatorInfo
- struct OperatorStatus
- class PhoneFactory

  PhoneFactory is the central factory to create all Telephony SDK Classes and services. More...

- struct Point
- class Polygon

- struct PreferredNetworkInfo
- struct RFBandInfo
- struct ServingSystemInfo
- class SignalStrength
- class SimProfile

  SimProfile class represents single eUICC profile on the card. More...
- struct SimRatCapability
- struct SlotStatus
- class SmsMessage

  Data structure represents an incoming SMS. This is applicable for single part message or part of the multipart message. More...
- struct SmsMetaInfo

  Provides certain attributes of an SMS message. More...
- class TdscdmaCellIdentity
- class TdscdmaCellInfo
- class TdscdmaSignalStrengthInfo
- class VoiceServiceInfo
- class WarningAreaInfo
- class WcdmaCellIdentity
- class WcdmaCellInfo
- class WcdmaSignalStrengthInfo

5.13.1 Data Structure Documentation

Type	Field	Description

5.13.1.1 struct telux::tel::ImsRegistrationInfo

Defines the IMS registration status parameters and the error code value

Data fields

Type	Field	Description
RegistrationStatus	imsRegStatus	The status of the IMS registration with the network
RadioTechnology	rat	The RAT is returned when IMS registration is being attempted or is successful
int	errorCode	An error code is returned when the IMS registration status is RegistrationStatus::NOT_REGISTERED. Values(Defined in SIP-RFC3261 section 13.2.2.2 and section 13.2.2.3): • 3xx – Redirection responses • 4xx – Client failure responses • 5xx – Server failure responses • 6xx – Global failure responses
string	errorString	Registration failure error string when the IMS is not registered.

5.13.1.2 struct telux::tel::ImsServiceInfo

Represents the status for supporting various services over IMS.

Data fields

Type	Field	Description
CellularServiceStatus	sms	SMS service status over IMS
CellularServiceStatus	voice	Voice service status over IMS

5.13.2 Typedef Documentation

5.13.2.1 using telux::tel::ImsRegistrationInfoCb = typedef std::function<void(ImsRegistrationInfo status, telux::common::ErrorCode error)>

This function is called in the response to requestRegistrationInfo API.

The callback can be invoked from multiple different threads. The implementation should be thread safe.

Parameters

in	*status*	Indicates the IMS registration status and the error code telux::tel::ImsRegistrationInfo.

in	*error*	Return code which indicates whether the operation succeeded or not telux::common::ErrorCode.

5.13.2.2 using telux::tel::ImsServiceInfoCb = typedef std::function<void(ImsServiceInfo service, telux::common::ErrorCode error)>

This function is called in response to the requestServiceInfo API.

The callback can be invoked from multiple different threads. The implementation should be thread safe.

Parameters

in	*service*	Indicates the IMS service information telux::tel::ImsServiceInfo.
in	*error*	Return code which indicates whether the operation succeeded or not telux::common::ErrorCode.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

5.13.2.3 using telux::tel::RatMask = typedef std::bitset<16>

16 bit mask that denotes which of the radio access technologies defined in RatType enum are used for preferred networks.

5.13.2.4 using telux::tel::SelectionModeResponseCallback = typedef std::function<void(NetworkSelectionMode mode, telux::common::ErrorCode error)>

This function is called with the response to requestNetworkSelectionMode API.

The callback can be invoked from multiple different threads. The implementation should be thread safe.

Parameters

in	*mode*	NetworkSelectionMode
in	*error*	Return code which indicates whether the operation succeeded or not telux::common::ErrorCode

5.13.2.5 using telux::tel::PreferredNetworksCallback = typedef std::function<void(std::vector<PreferredNetworkInfo> info, std::vector<PreferredNetworkInfo> staticInfo, telux::common::ErrorCode error)>

This function is called with the response to requestPreferredNetworks API.

The callback can be invoked from multiple different threads. The implementation should be thread safe.

Parameters

in	*info*	3GPP preferred networks list i.e PLMN list.
in	*staticInfo*	Static 3GPP preferred networks list i.e OPLMN list.
in	*error*	Return code which indicates whether the operation succeeded or not. telux::common::ErrorCode

5.13.2.6 using telux::tel::NetworkScanCallback = typedef std::function<void(std::vector<OperatorInfo> operatorInfos, telux::common::ErrorCode error)>

This function is called with the response to performNetworkScan API.

The callback can be invoked from multiple different threads. The implementation should be thread safe.

Parameters

in	*operatorInfos*	Operators info with details of network operator name, MCC, MNC and status.
in	*error*	Return code which indicates whether the operation succeeded or not. telux::common::ErrorCode

5.13.3 Enumeration Type Documentation

5.13.3.1 enum telux::tel::RegistrationStatus [strong]

Defines the IMS registration status parameters

Enumerator

UNKOWN_STATE Unknown status for IMS
NOT_REGISTERED Not registered status for IMS
REGISTERING Registering status for IMS
REGISTERED Registered status for IMS
LIMITED_REGISTERED Limited registration status for IMS

5.13.3.2 enum telux::tel::CellularServiceStatus [strong]

Defines the cellular service status parameters.

Enumerator

UNKNOWN Unknown service status
NO_SERVICE Unavailable service status
LIMITED_SERVICE Emergency service status
FULL_SERVICE Available service status

5.14 telux::therm Namespace Reference

Data Structures

- struct BoundCoolingDevice
- class ICoolingDevice

  ICoolingDevice provides interface to get type of the cooling device, the maximum throttle state and the currently requested throttle state of the cooling device. More...

- class IThermalListener

  Listener class for getting notifications when thermal service status changes. The client needs to implement these methods as briefly as possible and avoid blocking calls in it. The methods in this class can be invoked from multiple different threads. Client needs to make sure that the implementation is thread-safe. More...

- class IThermalManager

  IThermalManager provides interface to get thermal zone and cooling device information. More...

- class IThermalShutdownListener

  Listener class for getting notifications when automatic thermal shutdown mode is enabled/ disabled or will be enabled imminently. The client needs to implement these methods as briefly as possible and avoid blocking calls in it. The methods in this class can be invoked from multiple different threads. Client needs to make sure that the implementation is thread-safe. More...

- class IThermalShutdownManager

  IThermalShutdownManager class provides interface to enable/disable automatic thermal shutdown. Additionally it facilitates to register for notifications when the automatic shutdown mode changes. More...

- class IThermalZone

  IThermalZone provides interface to get type of the sensor, the current temperature reading, trip points and the cooling devices binded etc. More...

- class ITripPoint

  ITripPoint provides interface to get trip point type, trip point temperature and hysteresis value for that trip point. More...

- class ThermalFactory

  ThermalFactory allows creation of thermal manager. More...

5.15 telux::wlan Namespace Reference

Data Structures

- struct ApConfig
- struct ApInfo
- struct ApNetConfig
- struct ApNetInfo
- struct ApStatus
- struct DeviceIndInfo
- struct DeviceInfo

- class IApInterfaceManager

  Manager class for configuring Wlan Access Points. *More...*

- class IApListener
- struct InterfaceStatus
- class IStaInterfaceManager

  Manager class for configuring Wlan Station Mode. *More...*

- class IStaListener
- class IWlanDeviceManager

  WlanDeviceManager is a primary interface for configuring Wireless LAN. it provide APIs to enable, configure, activate, and modify modes. *More...*

- class IWlanListener
- struct StaConfig
- struct StaStaticIpConfig
- struct StaStatus
- class WlanFactory

  WlanFactory is the central factory to create all wlan classes. *More...*

6 Data Structure Documentation

6.1 telux::cv2x::ICv2xListener Class Reference

Cv2x Radio Manager listeners implement this interface.

Public member functions

- virtual void onStatusChanged (Cv2xStatus status)
- virtual void onStatusChanged (Cv2xStatusEx status)
- virtual void onSlssRxInfoChanged (const SlssRxInfo &slssInfo)
- virtual ~ICv2xListener ()

Cv2x Radio Manager listeners implement this interface.

6.1.1 Constructors and Destructors

6.1.1.1 virtual telux::cv2x::ICv2xListener::~ICv2xListener ( ) [virtual]

Destructor for ICv2xListener

6.1.2 Member Function Documentation

6.1.2.1 virtual void telux::cv2x::ICv2xListener::onStatusChanged (Cv2xStatus *status*) [virtual]

Called when the status of the CV2X radio has changed.

Parameters

in	*status*	- CV2X radio status.

Deprecated

use onStatusChanged(Cv2xStatusEx status)

6.1.2.2 virtual void telux::cv2x::ICv2xListener::onStatusChanged (Cv2xStatusEx *status*) [virtual]

Called when the status of the CV2X radio has changed.

Parameters

in	*status*	- CV2X radio status.

6.1.2.3 virtual void telux::cv2x::ICv2xListener::onSlssRxInfoChanged (const SlssRxInfo & *slssInfo*) [virtual]

Called when CV2X SLSS Rx is enabled and any of below events has occurred:

- A new SLSS synce reference UE is detected, lost, or selected as the timing source, report the present sync reference UEs.
- UE timing source switches from SLSS to GNSS, report 0 sync reference UE.
- SLSS Rx is disabled, report 0 sync reference UE.
- Cv2x is stopped, report 0 sync reference UE.

Parameters

in	*info*	- CV2X SLSS Rx information.

6.2 telux::tel::IImsServingSystemListener Class Reference

A listener class for monitoring changes in IMS Serving System manager, including IMS registration status change. Override the methods for the state that you wish to receive updates for.

Public member functions

- virtual void onServiceStatusChange (telux::common::ServiceStatus status)
- virtual void onImsRegStatusChange (ImsRegistrationInfo status)
- virtual void onImsServiceInfoChange (ImsServiceInfo service)
- virtual ~IImsServingSystemListener ()

A listener class for monitoring changes in IMS Serving System manager, including IMS registration status change. Override the methods for the state that you wish to receive updates for.

The methods in listener can be invoked from multiple different threads. The implementation should be thread safe.

6.2.1 Constructors and Destructors

6.2.1.1 virtual telux::tel::IImsServingSystemListener::~IImsServingSystemListener ( ) [virtual]

6.2.2 Member Function Documentation

6.2.2.1 virtual void telux::tel::IImsServingSystemListener::onServiceStatusChange (telux::common::ServiceStatus *status*) [virtual]

This function is called when service status changes.

Parameters

in	*status*	- telux::common::ServiceStatus

Reimplemented from telux::common::IServiceStatusListener.

6.2.2.2 virtual void telux::tel::IImsServingSystemListener::onImsRegStatusChange (ImsRegistrationInfo *status*) [virtual]

This function is called whenever any IMS service configuration is changed.

Parameters

in	*status*	Indicates which registration status is the IMS service changed to. telux::tel::ImsRegistrationInfo.

6.2.2.3 virtual void telux::tel::IImsServingSystemListener::onImsServiceInfoChange (ImsServiceInfo *service*) [virtual]

This function is called whenever any IMS service information is changed.

Parameters

in	*service*	Indicates which IMS service information has changed. telux::tel::ImsServiceInfo.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

6.3 telux::tel::IImsServingSystemManager Class Reference

IMS Serving System Manager is the primary interface for IMS related operations Allows to query IMS registration status.

Public member functions

- virtual telux::common::ServiceStatus getServiceStatus ()=0
- virtual telux::common::Status requestRegistrationInfo (ImsRegistrationInfoCb callback)=0
- virtual telux::common::Status requestServiceInfo (ImsServiceInfoCb callback)=0
- virtual telux::common::Status registerListener (std::weak_ptr< telux::tel::IImsServingSystemListener > listener)=0
- virtual telux::common::Status deregisterListener (std::weak_ptr< telux::tel::IImsServingSystemListener > listener)=0
- virtual ~IImsServingSystemManager ()

IMS Serving System Manager is the primary interface for IMS related operations Allows to query IMS registration status.

6.3.1 Constructors and Destructors

6.3.1.1 virtual telux::tel::IImsServingSystemManager::~IImsServingSystemManager ( ) [virtual]

6.3.2 Member Function Documentation

6.3.2.1 virtual telux::common::ServiceStatus telux::tel::IImsServingSystemManager::getServiceStatus ( ) [pure virtual]

This status indicates whether the IImsServingSystemManager object is in a usable state.

Returns

SERVICE_AVAILABLE - If IMS Serving System manager is ready for service.
SERVICE_UNAVAILABLE - If IMS Serving System manager is temporarily unavailable.
SERVICE_FAILED - If IMS Serving System manager encountered an irrecoverable failure.

6.3.2.2 virtual telux::common::Status telux::tel::IImsServingSystemManager::requestRegistrationInfo (ImsRegistrationInfoCb *callback*) [pure virtual]

Request IMS registration information.

Parameters

in	*callback*	Callback pointer to get the response of requestRegistrationInfo.

Returns

Status of requestRegistrationInfo i.e. success or suitable status code.

6.3.2.3 virtual telux::common::Status telux::tel::IImsServingSystemManager↩ ::requestServiceInfo (ImsServiceInfoCb *callback*) [pure virtual]

Request IMS service information, such as SMS and voice service status over IMS.

Parameters

in	*callback*	Callback pointer to get the response of requestServiceInfo.

Returns

Status of requestServiceInfo i.e., success or suitable status code.

Note

Eval: This is a new API and is being evaluated. It is subject to change and could break backwards compatibility.

6.3.2.4 virtual telux::common::Status telux::tel::IImsServingSystemManager↩ ::registerListener (std::weak_ptr< telux::tel::IImsServingSystemListener > *listener*) [pure virtual]

Add a listener to listen for specific events in the IMS Serving System subsystem.

Parameters

in	*listener*	Pointer to IImsServingSystemListener object that processes the notification

Returns

Status of registerListener i.e. success or suitable error code.

6.3.2.5 virtual telux::common::Status telux::tel::IImsServingSystemManager↩ ::deregisterListener (std::weak_ptr< telux::tel::IImsServingSystemListener > *listener*) [pure virtual]

Remove a previously added listener.

Parameters

in	*listener*	Listener to be removed.

Returns

Status of deregisterListener i.e. success or suitable error code.

6.4 telux::common::IServiceStatusListener Class Reference

Public member functions

- virtual void onServiceStatusChange (ServiceStatus status)
- virtual ~IServiceStatusListener ()

6.4.1 Constructors and Destructors

6.4.1.1 virtual telux::common::IServiceStatusListener::~IServiceStatusListener ( ) [virtual]

6.4.2 Member Function Documentation

6.4.2.1 virtual void telux::common::IServiceStatusListener::onServiceStatusChange (ServiceStatus *status*) [virtual]

This function is called when service status changes.

Parameters

in	*status*	- ServiceStatus

Reimplemented in telux::tel::IImsServingSystemListener.