API Reference Guide

AudioCodes WebRTC Solutions for Enterprises

WebRTC iOS Client SDK

Version 1.5.0

audiocodes

Table of Contents

Notice

Information contained in this document is believed to be accurate and reliable at the time of printing. However, due to ongoing product improvements and revisions, AudioCodes cannot guarantee accuracy of printed material after the Date Published nor can it accept responsibility for errors or omissions. Updates to this document can be downloaded from https://www.audiocodes.com/library/technical-documents.

This document is subject to change without notice.

Date Published: June-23-2026

Security Vulnerabilities

All security vulnerabilities should be reported to vulnerability@audiocodes.com.

Customer Support

Customer technical support and services are provided by AudioCodes or by an authorized AudioCodes Service Partner. For more information on how to buy technical support for AudioCodes products and for contact information, please visit our website at https://www.audiocodes.com/services-support/maintenance-and-support.

Stay in the Loop with AudioCodes

Abbreviations and Terminology

Each abbreviation, unless widely used, is spelled out in full when first used.

Related Documentation

Document Name
WebRTC-Gateway
WebRTC Softphone Client Quick Guide
WebRTC Client Installation Manual
WebRTC Click-to-Call Widget Installation and Configuration Guide
WebRTC Android Client SDK API Reference Guide
WebRTC Web Browser Client SDK API Reference Guide

Document Revision Record

LTRT	Description
14080	Initial document release for Version 1.0
14081	■ Updated to software Version 1.0.1. ■ Updated additional information on SDK installation and usage. ■ Updated the API with the new "contactRewrite" function.
14082	■ Updated to software Version 1.1.0. ■ Blind Transfer: • New function: transferCall:(RemoteContact*)remoteContact • New property: transferState • New property: transferContact ■ OAuth authorization – New function setOauthToken ■ Click to call support (calls without registration): • New function – login:(BOOL) autoRegister ■ Push Support: • New function: setPushNotifications ■ Support for Google WebRTC 1.0.27828 ■ Support for Delayed Offer
14083	■ Updated to software Version 1.2.0 ■ Instant Messaging: • New function: sendInstantMessage ■ Call Persistence on broken RTP Stream: • New property: disconnectOnBrokenConnection ■ Attended Transfer: • New function: attendedTransferCall ■ Support for IPV6 ■ Full Bitcode Support ■ Minimum iOS Version required is 10.0
14084	■ Updated to software Version 1.2.5 ■ Updated Bitcode section ■ Added Swift section ■ Added CallKit Integration: • Added ACNativeCallService API • Added CallKit section • Updated Installation section • Updated WebRTCAudioManager class • Updated WebRTCAudioManager Notifications section • Updated Use Case Examples section; added CallKit use cases
14085	■ Added TerminationInfo type ■ Added terminationInfo property to AudioCodesSession class
14086	■ Updated to Version 1.2.7 ■ Added TLS certificate verification API: • Added verifyServerCertificate property • Added caCertFilePath property

LTRT	Description
14087	■ Updated to Version 1.2.8 ■ Added the “Simulator Support With Xcode 12” section ■ ACNativeCallService: Added the isCallAssociatedWithNative method ■ Added the Genesys 3PCC API for remote control events: • AudioCodesEventListener : Added the optional infoAlert parameter to the incomingCall delegate callback • AudioCodesSessionEventListener : Added the callNotifyEvent delegate callback • Updated Use Case Examples section; Added Section for remote-control events • Demo Client: Updated the implementation for the incomingCall delegate callback, to handle the infoAlert parameter for auto-answering the call • Demo Client: Added implementation for the callNotifyEvent delegate callback, to handle incoming notify messages for remote-control functionalities to manage a call (‘hold’ / ‘talk’ / ‘dtmf’)
14088	■ Updated to Version 1.2.9 ■ SDK format was updated from '.framework' to '.xcframework'
14089	■ Updated to Version 1.3.0 ■ Updated minimum requirements to Xcode 12.5.1 ■ Removed 32-bit support, architectures armv7 and i386 are no longer supported ■ Added Apple Silicon arm64 support, for running the simulator on Apple Silicon machines ■ Added Push Notifications API support, updated the setPushNotifications API ■ Added Push Notifications Use Cases
14090	■ Updated to Version 1.3.1 ■ Added Delayed Offer Configuration Support: • Added isDelayedOffer getter property to AudioCodesSession • The showVideo method is now able to add video to delayed-offer incoming calls • Added a use-case example of configuring an incoming delayed-offer call as a video call
14091	■ Updated to Version 1.3.3 ■ Updated installation and usage notes. ■ Added handling audio interrupts and GSM calls. ■ Added SIP connection binding capabilities: • Added method setConnectionBinding. • Added an optional parameter to handleNetworkChange. • Added new section for the ACNetworkConnectionAttributes type. • Added “Binding SIP Connections”.
14092	■ Updated to Version 1.3.4 ■ Updated the AudioCodesEventListener section for SIP instant message: • Added incomingInstantMessage and instantMessageStatus callbacks. ■ Added SIP INFO support: • Updated the AudioCodesSession section: added the sendInfo method • Updated the AudioCodesSessionEventListener section: added the incomingInfo event callback.

LTRT	Description
14093	■ Updated to Version 1.3.5. ■ Updated minimum requirements to Xcode 14.1 and iOS 12. ■ Added the *ACRTCIceServer* and *ACRTCIceServerFactory* APIs to the AudioCodesUA section. ■ Logging: • The SDK default logger now uses the recommended unified logging *os_log* function. • The SDK default log level is now 'error' instead of 'info'. ■ Removed Bitcode support: • Updated the Bitcode Support section, which has now been changed to Bitcode Support Deprecation. • The *WebRTC.xcframwwork* bundle is now provided with the SDK, instead of separately. ■ Moved the *ACNativeCallService* class, as well as all CallKit references, to a separate optional framework in the SDK: *MVWebRTCNativeCall.xcframework*.
14094	■ Updated to Version 1.3.7 ■ Minimum requirements updated to iOS 13.4 ■ New section 'Viewing SDK Logs' under Usage Notes
14095	■ Updated logout and forceClose
14096	■ Updated ACLogLevel logLevel, ACLogLevel sdkLogLevel, ACLogLevel sipLogLevel, and ACLogLevel webRTCLogLevel.
14096	■ Added includes Swift API extension
14097	■ Added Swift SDK API Reference ■ Added Objective-C and Swift API ■ Updated Use Case examples

Documentation Feedback

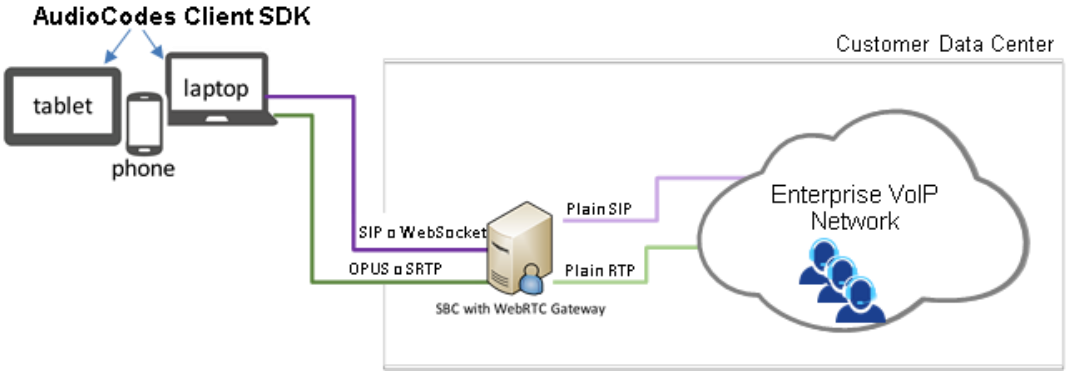
AudioCodes continually strives to produce high quality documentation. If you have any comments (suggestions or errors) regarding this document, please fill out the Documentation Feedback form on our Web site at https://online.audiocodes.com/documentation-feedback.

1 Introduction

WebRTC technology enriches the user experience by adding voice, video and data communication to the browser, as well as to mobile applications. AudioCodes WebRTC gateway provides seamless connectivity between WebRTC clients and existing VoIP deployments.

A typical WebRTC solution is comprised of a WebRTC Gateway, which is an integrated functionality on AudioCodes SBCs, and a client application running on a browser or a mobile app. The AudioCodes WebRTC iOS client SDK is based on Objective-C and allows iOS developers to integrate WebRTC functionality into iOS applications for placing calls from the iOS device to the SBC.

Figure 1: Typical WebRTC Solution



For a simple click-to-call button use case, a WebRTC widget is offered, which can be easily integrated into websites and blogs without any JavaScript knowledge. Refer to the *WebRTC Widget Installation and Configuration Guide*.

1.1 Purpose

This *Reference Guide* defines the Application Programming Interface (API) use of the Web Real-Time Communications (RTC) SDK.

1.2 Scope

This *Reference Guide* describes the API that must be implemented to use AudioCodes' WebRTC iOS SDK to build an iOS application. This application will interact with the AudioCodes' server to establish voice and video calls.

This *Reference Guide* may be used by iOS developers who want to use the AudioCodes-provided SDK to build WebRTC clients.

1.3 Benefits

Here is a summary of the benefits of using WebRTC:

- Simple deployment - a single WebRTC gateway device is used for both signaling and media
- Strong security and interoperability capabilities resulting from integration with SBC
- Client SDK for iOS application
- OPUS-powered IP phones for superb, transcoder-less voice quality
- Optional recording of WebRTC calls

2 iOS SDK

2.1 Getting Started

The following provides the requirements for working with the iOS SDK:

- Xcode 14.1 or later
- iOS device running Version iOS 14.0 or later
- SDK includes the following framework bundles:
  - **ACWebRTCSdk.xcframework:** This framework bundle provides the Swift API for the WebRTC SDK. It is a native Swift integration layer built on top of the existing Objective-C implementation and is intended as the primary choice for native Swift applications. It must be linked and embedded together with MVWebRTCFramework.xcframework, MVWebRTCInterface.xcframework, and WebRTC.xcframework.

Some advanced APIs are available only through the Objective-C API. When those features are required, the Swift API can be used alongside the Objective-C API (See Section 2.5).

  - **MVWebRTCFramework.xcframework:** This is the main WebRTC SDK framework bundle providing the primary Objective-C API. Its WebRTC media-related functionalities require the project to include MVWebRTCInterface. On its own, it is extension-safe and can be used in app extensions, mainly for push-notification use cases (See Section 6.15).
  - **MVWebRTCInterface.xcframework:** This must be included with MVWebRTCFramework, to utilize real-time media functionalities. It depends on the WebRTC.xcframework that is also provided here.
  - **WebRTC.xcframework:** This must be included with MVWebRTCFramework to utilize real-time media functionalities.
  - **MVWebRTCNativeCall.xcframework:** Optional framework containing the SDK API and functionality for using the iOS native telephony integration, i.e., CallKit. All the SDK references to the iOS CallKit framework are contained in here. This functionality is not available through the Swift SDK API. Applications that do not wish to link to CallKit can simply exclude this framework from their project.
  - **iOS demo client project**: This is the Xcode project which can be used as a reference. This is a fully-working client and shows how to use the SDK.

Until Version 1.3.4, the WebRTC.xcframework package has been delivered separately. From Version 1.3.5, the SDK now includes WebRTC.xcframework as part of its core delivery.

2.2 Installation

The procedure below describes how to install the SDK.

To install the SDK:

1. Install Xcode 14.1 or later.
2. Open a project for the demo client.
3. Verify that the iOS Deployment Target is Version 14.0. This is the lowest supported iOS version.
4. Add all SDK Frameworks in the project settings, under **General** > **Embedded Binaries**.
5. In most cases, , you do not need to manually add Apple system frameworks. Xcode automatically links the required frameworks when they are imported in Swift or Objective-C code. If the project fails to link because of missing system symbols, add the missing framework or library under **General** > **Frameworks, Libraries and Embedded Content**:
   - libresolv.tbd
   - SystemConfiguration.framework
   - Security.framework
   - CoreGraphics.framework
   - CoreMedia.framework
   - CoreVideo.framework
   - CoreAudio.framework
   - CFNetwork.framework
   - AudioToolbox.framework
   - AVFoundation.framework
   - CoreFoundation.framework
   - Foundation.framework
   - UIKit.framework
   - CallKit.framework. Only required for using MVWebRTCNativeCall.xcframework (Optional)

2.3 Usage Notes

1. **Background Modes:** To allow VoIP connection in the background, the following entries must be added to the app's info.plist file:

```
<key>UIBackgroundModes</key>
<array>
        <string>voip</string>
        <string>audio</string>
</array>
```

2. **Background State Transition:** When the application transitions to the Background state, it must call the `logout()` method in order to close the connection to the SIP server. It is strongly advised to do so while initiating a `UIBackgroundTask`, which will be ended on the `loginStateChanged` delegate method, indicating that logout has completed. See Section 6.15.1.
3. **Privacy:** Usage of WebRTC for VoIP purposes requires using the device's camera and microphone, which require the user's consent. To fine-tune the privacy access request from the user, the following entries should be added to the app's info.plist file:

```
<key>NSCameraUsageDescription</key>
```

```
<string>The Application requires camera access for full video
calls functionality</string>
<key>NSMicrophoneUsageDescription</key>
<string>The application requires microphone access for full
call functionality</string>
```

4. **Platform Consideration (Device / Simulator):** Every framework we provide, includes within it multiple code partitions (slices), compiled for the following architectures:
   - Device arm64
   - Simulator arm64 (Apple Silicon)

   This allows for the running / debugging of both device and simulator as needed.
5. **Viewing SDK logs:** By default, if no custom logger is defined for the SDK, the SDK uses the unified-logging *os_log* functionality to print its log entries to the console, using the subsystem value **"ac.webrtc.sdk"**. The log category of each log entry can be one of the following:
   - sdk: For higher-level SDK entries
   - sip: For SIP-level SDK entries
   - sip-signal: For incoming or outgoing SIP messages
   - media: For real-time media level SDK entries

   These log messages can be viewed in the Console app and filtered by subsystem and category.

   To fully view SIP messages in the console log, and overcome the 1024-byte size limit on logs, the application info.plist has to be configured to allow oversized os_log messages for the **sip-signal** category. To do so, add the following entry to the info.plist file:

```
<key>OSLogPreferences</key>
<dict>
    <key>ac.webrtc.sdk</key>
    <dict>
        <key>sip-signal</key>
        <dict>
            <key>Enable-Oversize-Messages</key>
            <true/>
        </dict>
    </dict>
</dict>
```

2.4 Bitcode Support Deprecation

AudioCodes no longer supports Bitcode, due to Apple's deprecation of Bitcode since Xcode 14 was released. This allows us to minimize the SDK bundle size.

2.5 Objective-C and Swift API

The SDK is written in Objective-C and provides both a legacy Objective-C API and a modern Swift API. New applications are encouraged to use the Swift API, which exposes a native Swift interface for nearly all SDK functionality. Existing applications can continue using the Objective-C API without modification. A small number of advanced features are currently available only through the Objective-C API.

2.5.1 Using Objective-C API

Use the Objective-C SDK directly from Swift by importing the framework module in the Swift file:

```
import MVWebRTCFramework

let phone = AudioCodesUA.getInstance()
phone?.setServerConfig(
    "webrtc.audiocodes.com",
    port: 5090,
    serverDomain: "audiocodes.com",
    transport: .tcp,
    iceServers: nil
)

phone?.setAccount(
    "1001",
    displayName: "John Smith",
    password: "secret",
    authName: "1001"
)

phone?.login()
```

2.5.2 Using Swift API

To use the Swift SDK API import ACWebRTCSdk:

```
import ACWebRTCSdk

let ua = UserAgent.shared

ua.configureServer(
    .init(
        proxyAddress: "webrtc.audiocodes.com",
        port: 5090,
        serverDomain: "audiocodes.com",
        transport: .tcp
    )
)

ua.configureAccount(
  .init(
      userName: "1001",
      authentication: .digest(userName: "1001", password:
"secret")
  )
)

ua.login()
```

2.5.3 Mixing Objective-C and Swift API

When you need a feature that is not yet exposed by the Swift API, import ACWebRTCSdk for the primary application flow and import MVWebRTCFramework to access the underlying Objective-C object through the escape hatch:

```
import ACWebRTCSdk
import MVWebRTCFramework

let userAgent = UserAgent.shared

// Swift API for the supported flow
userAgent.configureServer(...)
userAgent.configureAccount(...)
userAgent.login()

// Objective-C escape hatch for missing features
if let objcUA = userAgent.audioCodesUA as? AudioCodesUA {
    objcUA.setPushNotificationsTeamId(
        "TEAMID",
        bundleId: Bundle.main.bundleIdentifier ?? "",
        apnsToken: apnsToken,
        voipToken: voipToken
    )
}
```

The SDK exposes two temporary Objective-C escape hatches:

- 'UserAgent.audioCodesUA' for user-agent level features not yet available in Swift API.
- 'CallSession.audioCodesSession' for call-level features not yet available in Swift API.

2.5.4 Objective-C APIs Not Exposed by the Swift API

The Swift API does not expose the following Objective-C SDK members.

2.5.4.1 User Agent Configuration

- `AudioCodesUA.delegate`
- `AudioCodesUA.setPushNotificationsTeamId:bundleId:apnsToken:voipToken:`
- `AudioCodesUA.sendInstantMessage:to:`

2.5.4.2 Call Session Extras

- `AudioCodesSession.delegate`
- `AudioCodesSession.transferState`
- `AudioCodesSession.transferContact`
- `AudioCodesSession.associatedPushNotification`
- `AudioCodesSession.userData`
- `AudioCodesSession.transferCall:`
- `AudioCodesSession.attendedTransferCall:`

- AudioCodesSession.sendInfo:contentType:
- AudioCodesSession.showVideoLocalView:remoteView:completion:

2.5.4.3 Native CallKit Support

- ACNativeCallService
- ACIncomingCallPushNotification

2.5.4.4 Listener Protocols

- AudioCodesEventListener
- ACLoggerProtocol
- ACInfoAlertAttributes
- ACInfoMessage

2.5.4.5 Notification Names and User-Info Keys

- ACUALoginStateChangedNotification
- ACUALoginStateCauseUserInfoKey
- ACUALoginStateIsLoginUserInfoKey
- ACUAIncomingCallNotification
- ACUAIncomingCallUserInfoKey
- ACUAIncomingCallInfoAlertUserInfoKey
- ACUAIncomingMessageNotification
- ACUAIncomingMessageUserInfoKey
- ACUAIncomingMessageFromUserInfoKey
- ACUAMessageStatusNotification
- ACUAMessageStatusUserInfoKey
- ACUAMessageStatusIdUserInfoKey
- ACSessionCallProgressNotification
- ACSessionCallTerminatedNotification
- ACSessionCameraSwitchedNotification
- ACSessionIncomingInfoNotification
- ACSessionCameraSwitchedFrontCameraUserInfoKey
- ACSessionCallNotifyEventNotification
- ACSessionNotifyEventTypeUserInfoKey
- ACSessionNotifyDTMFUserInfoKey
- ACSessionIncomingInfoMessageUserInfoKey
- ACAudioRouteChangedNotification
- ACAudioRouteRouteChangedNotificationCurrentRouteKey
- ACAudioRouteRouteAvailabilityChangedNotification
- ACAudioRouteRouteAvailabilityChangedReceiverAvailableKey
- ACAudioRouteRouteAvailabilityChangedSpeakerAvailableKey
- ACAudioRouteRouteAvailabilityChangedBluetoothAvailableKey

- ACAudioInterruptNotification
- ACAudioSessionUserInfoKey
- ACAudioIsInterruptedUserInfoKey
- ACAudioShouldResumeUserInfoKey
- ACAudioWasSuspendedUserInfoKey

2.6 CallKit Framework

The SDK provides the **MVWebRTCNativeCall.xcframework** optional framework, to allow for the integration of the app's VoIP calls with other call-related apps in the system.

> Apps that are not linked with CallKit at all, can remove the inclusion of this framework from their Xcode project.

You can use the built-in CallKit integration via the `MVWebRTCNativeCall/ACNativeCallService` class (See Section 4.11 for API reference, and Section 6.12 for examples).

It is also possible to interface with the CallKit framework manually, and only use the SDK's standard AudioCodesUA and AudioCodesSession APIs for internally VOIP call flows.

This requires a more granular management of the App's audio session, and of the WebRTC audio unit that performs real-time VOIP processing. Use the `WebRTCAudioManager` API for this functionality. See Section 4.3 for API reference, and Section 6.13 for examples.

> It is **highly recommended** to use the `MVWebRTCNativeCall` framework for CallKit integration, and also for handling audio interruptions when not using CallKit at all. The `ACNativeCallService` class features a comprehensive logic layer to manage the interface with the CallKit framework and the Operating System (OS) audio interruptions mechanism, in a way that correctly handles multiple calls.
>
> Opting out of the built-in CallKit support of the SDK, and using CallKit manually, may render CallKit unstable when using it with multiple calls. This has been an inherent limitation with CallKit for a long time The `MVWebRTCNativeCall` framework resolves it.

2.7 Push Notifications

When using push notifications, the application MUST use the CallKit framework for incoming calls, because incoming call push notifications arrive via VOIP Push. See Section 6.15.

2.8 App Extensions

ACWebRTCSdk.xcframework and MVWebRTCFramework.xcframework are extension-safe and can be used in app extensions for performing SIP login (for example, to implement the most robust strategy to respond to register-refresh push notifications).

> MVWebRTCInterface.xcframework and WebRTC.xcframework contain all media-related references to APIs that are not extension safe, which is why they are decoupled from MVWebRTCFramework.xcframework. See Section 6.15.2.

3 Swift SDK API Reference

This chapter documents the Swift SDK API as a reference guide. The section is organized by the main entry points first, then by supporting configuration and value types, and finally by the audio and SDK-wide helpers that complete the public surface.

The Swift SDK API is the primary Swift integration layer for the WebRTC client SDK. Use `UserAgent` (section3.1) to configure SIP connectivity and account registration, `CallSession` (section3.9) to control calls, `AudioManager` (section3.11) to manage audio routing, and SDKConfiguration (section3.12) to control shared SDK settings.

3.1 UserAgent (class)

The shared user agent is the main entry point for Swift applications. It configures the SIP server and account, starts and stops registration, creates outgoing calls, handles incoming calls, and delivers account-level callbacks.

3.1.1 Properties

Property	Description
shared	The shared user agent instance used by the application.
audioCodesUA	A temporary interoperability escape hatch that exposes the wrapped Objective-C user agent when it is available. Use this only for migration or for features that are not yet exposed through Swift.
callbackQueue	The queue used to deliver user-agent callbacks such as log messages, login state changes, and incoming calls. The default queue is the main queue.
userAgentHeader	The SIP User-Agent header value used for outgoing SIP messages. Set it before login when the application needs a specific client identity.
inviteHeaders	Global SIP INVITE headers that are merged into newly created call sessions. Per-call INVITE headers override global values for duplicate header names.
logLevel	The general logging level used by the SDK.
sdkLogLevel	The logging level used by the AudioCodes SDK code.
sipLogLevel	The logging level used for SIP stack messages.
webRTCLogLevel	The logging level used for WebRTC adapter and native WebRTC messages.
serverCertificateVerification	The TLS SIP server certificate verification mode. Use it to control whether TLS server certificates are validated, disabled, or checked against a custom CA file.
disconnectOnBrokenConnection	A Boolean value that indicates whether the account should disconnect when the SIP connection breaks.
allowedNotifyEvents	The NOTIFY event names advertised as allowed for call sessions. This affects SIP Allowed-Events signaling, not the delivery of `onNotifyEvent` callbacks.
logger	A callback that receives SDK log messages. The callback is delivered on `callbackQueue`.
sessions	The call sessions currently known to the user agent.
onLoginStateChanged	A callback invoked when the account login state changes.
onIncomingCall	A callback invoked when a new incoming call arrives.

3.1.2 Methods

Methods	Description
configureServer(_:)	Configures the SIP server environment used by the account. Call this before login. The server configuration provides the proxy address, port, SIP domain, transport, and optional ICE server list.
configureAccount(_:)	Configures the SIP account identity and authentication method used for registration and calls. This also selects the active authentication method. Configuring no authentication, digest authentication, or deferred OAuth clears any previously configured OAuth bearer token.
updateOAuthToken(_:)	Updates the OAuth bearer token used for SIP authentication. Call this after configuring the account with OAuth when the token arrives later.
login()	Starts the SIP account using the default login options. The default behavior enables automatic registration, uses the default REGISTER expiration interval, enables session timers with default values, sends no custom REGISTER headers, applies automatic Contact rewrite, and leaves SIP connection binding disabled.
login(_:)	Starts the SIP account using the specified login options. Use this overload when the application needs custom registration expiration, session timer behavior, REGISTER headers, Contact rewrite behavior, or connection binding.
startCall(to:options:)	Starts an outgoing audio call to the specified remote contact. The call can be started as audio-only or audio+video by setting 'OutgoingCallOptions.media'. The call session inherits the user-agent callback queue and the global INVITE headers, and it can be further customized with per-call options.
logout()	Gracefully unregisters the account and shuts down the SIP stack. This is the convenience form of logout with the graceful shutdown mode.
logout(_:)	Logs out using the specified shutdown behavior. Use graceful shutdown to unregister before shutting down, force shutdown to skip unregister, or unregister-only to unregister while keeping the SIP stack and active calls alive.
handleNetworkChange()	Notifies the user agent that the active network connection changed when no address-family preference is available.
handleNetworkChange(_:)	Notifies the user agent that the active network connection changed and supplies the preferred local address family. This is used to refresh account and call connectivity, and it can reapply binding when SIP connection binding is active.

3.2 ServerConfiguration (struct)

Server configuration is the SIP connectivity description used by `UserAgent`. It combines the SIP proxy settings with optional ICE servers for media connectivity.

3.2.1 Properties

Property	Description
proxyAddress	The SIP proxy address.
port	The SIP proxy port.
serverDomain	The SIP server domain used in SIP signaling.
transport	The SIP transport protocol used for signaling.
iceServers	The ICE servers used for media connectivity.

3.2.2 Initialization

init(proxyAddress:port:serverDomain:transport:iceServers:)

Creates the server settings. The ICE server list defaults to an empty array when media connectivity does not require custom STUN or TURN servers.

3.2.3 Nested Types

Enum type	Enum cases	Definition
Transport (enum)	*'undefined'*	No transport protocol is specified.
	'udp'	User Datagram Protocol.
	'tcp'	Transmission Control Protocol.
	'tls'	Transport Layer Security over TCP.

Struct type	Struct Property	Definition
IceServer (struct)	*stun(urlStrings:)*	Creates a STUN server description for one or more stun: URLs. STUN servers do not use TURN authentication or TURN-over-TLS settings.
	turn(urlStrings:authentication:)	Creates a TURN server description for one or more turn: URLs. Authentication defaults to none when no credentials are needed.
	turns(urlStrings:authentication:tls:)	Creates a TURN-over-TLS server description for one or more turns: URLs. The TLS settings default to secure certificate validation, hostname-from-URL behavior, and no ALPN or elliptic-curve overrides.

Enum type	Enum cases	Definition
TurnAuthentication (enum)	*'none'*	No TURN credentials are sent.
	'credentials(username:credential:)'	TURN credentials are sent with the specified user name and credential.

Struct type	Struct Property	Definition
TurnTLSConfiguration (struct)	*certificatePolicy*	The TLS certificate verification policy. The default validates certificates.
	hostName	The TLS hostname source used for SNI. Use the URL hostname by default, or provide a custom hostname when the URL contains an IP address or another host alias.
	alpnProtocols	The protocol names advertised in the TLS ALPN extension.
	ellipticCurves	The TLS elliptic-curve names advertised to the TLS stack.

Enum type	Enum cases	Definition
TLSHostName (enum)	*'fromURL'*	Use the hostname from the TURN-over-TLS URL.
	'custom(String)'	Use a custom TLS hostname, typically for SNI when the URL contains an IP address.
TLSCertificatePolicy (enum)	*'secure'*	Validate TLS certificates.
	'insecureNoCheck'	Disable TLS certificate validation. Use only in controlled environments.

3.3 AccountConfiguration (struct)

Account configuration describes the SIP identity and authentication method used by the user agent. It is applied before login and determines how registration and calls authenticate.

3.3.1 Properties

Property	Description
userName	The SIP user name used for the account identity.
displayName	The display name presented for the account. When this value is nil, the SIP identity is sent without a display name.
authentication	The authentication method used for registration and calls.

3.3.2 Initialization

init(userName:displayName:authentication:)

Creates the account settings. Authentication is required so callers choose explicitly between no authentication, digest authentication, and OAuth.

3.3.3 Nested Types

Enum type	Enum cases	Definition
AccountAuthentication (enum)	*'none'*	No digest credentials or OAuth bearer token are configured. Selecting this method clears any previously configured OAuth bearer token when the account is configured.
	'digest(userName:password:)'	SIP digest authentication using an explicit authentication user name and password. Selecting this method clears any previously configured OAuth bearer token when the account is configured.
	'oauth(OAuth)'	OAuth bearer authentication.
Oauth (enum)	*'deferred'*	OAuth is selected and the access token will be provided later with `updateOAuthToken`(_:). Selecting this option clears any previously configured OAuth bearer token when the account is configured.
	'accessToken(String)'	OAuth is selected with an access token that is already available. The token is applied when the account is configured.

3.4 LoginOptions (struct)

Login options describe the policy applied when the user agent starts the SIP account. Several of these values are applied only when login starts, so changing them later requires another login cycle after the account shuts down.

3.4.1 Properties

Property	Description
autoRegister	Controls whether the account registers with the SIP server after initialization.
registrationExpirationSeconds	The SIP REGISTER refresh interval in seconds.
connectionBinding	The SIP connection binding policy applied for this login.
sessionTimer	The SIP session timer policy applied for this login.
registerHeaders	Additional SIP headers included in REGISTER requests for this login.
contactRewrite	The SIP Contact-header rewrite policy applied for this login.

3.4.2 Initialization

init(autoRegister:registrationExpirationSeconds:sessionTimer:registerHeaders:contactRewrite:connectionBinding:)

Creates the login options. The defaults enable automatic registration, use a 600-second REGISTER expiration interval, enable session timers, send no custom REGISTER headers, use automatic Contact rewrite, and leave connection binding disabled.

3.4.3 Nested Types

Enum type	Enum cases	Definition
ContactRewritePolicy (enum)	*'automatic'*	Use the default behavior for the selected registration mode. Contact rewrite is enabled automatically when automatic registration is disabled; otherwise it is disabled.
	'enabled'	Always enable SIP Contact-header rewrite for this login.
SessionTimer (enum)	*'disabled'*	Do not use SIP session timers.
	'enabled'	Use SIP session timers with the default SDK and SIP stack expiration.
	'enabledWithExpiration(seconds:)'	Use SIP session timers with an explicit Session-Expires interval.
ConnectionBinding (enum)	*'none'*	Do not apply SIP connection binding for this login. This does not remove binding from an already connected account.

Enum type	Enum cases	Definition
	'deferred'	Bind after a SIP connection is established, using the address family selected by that connection.
	'ipv4'	Bind to an IPv4 SIP connection before it is established.
	'ipv6'	Bind to an IPv6 SIP connection before it is established.

3.5 NetworkConnectionAttributes (struct)

Network connection attributes describe the address family used when the application reports a network change. They are mainly used together with connection binding.

3.5.1 Properties

Property	Description
localAddressFamily	The preferred local address family for SIP connections.

3.5.2 Initialization

init(localAddressFamily:)

Creates the network connection attributes.

3.5.3 Nested Types

Enum type	Enum cases	Definition
AddressFamily (enum)	*'unspecified'*	No address family is specified. During network-change handling this leaves existing connection binding unchanged.
	'ipv4'	IPv4. Use this to bind before the SIP connection is established or to reapply binding during network-change handling.
	'ipv6'	IPv6. Use this to bind before the SIP connection is established or to reapply binding during network-change handling.

3.6 RemoteContact (struct)

Remote contact is the SIP identity of the remote party. It is used for outgoing calls and for representing the remote side of a session.

3.6.1 Properties

Property	Description
displayName	The display name of the remote party.
userName	The user name of the remote party.
domain	The SIP domain of the remote party.

3.6.2 Initialization

init(displayName:userName:domain:)

Creates the remote contact. All fields are optional so the app can construct partial SIP identities when needed.

3.7 IncomingCall (struct)

Incoming call information combines the incoming call session with optional Alert-Info data parsed from the SIP INVITE.

3.7.1 Properties

Property	Description
session	The incoming call session.
infoAlert	Alert-Info attributes received with the call, when present.

3.7.2 Initialization

init(session:infoAlert:)

Creates the incoming call information.

3.7.3 Nested Types

Struct type	Struct Property	Definition
InfoAlert (struct)	*autoAnswer*	Indicates whether the call should be answered automatically.
	delay	The auto-answer delay.
init(autoAnswer:delay:)		Creates the Alert-Info attributes.

3.8 OutgoingCallOptions (struct)

Outgoing call options customize the media type, SIP INVITE headers, and the UUID associated with the call.

3.8.1 Properties

Property	Description
media	The media type to include when the call is created.
inviteHeaders	SIP INVITE headers sent with this call. They are merged with the global UserAgent.inviteHeaders dictionary, and per-call values override global values for duplicate header names.
uuid	The UUID assigned to the call. Use it when the application needs to associate the call with an existing CallKit transaction; otherwise the SDK creates one.

3.8.2 Initialization

init(media:inviteHeaders:uuid:)

Creates the outgoing call options. The media defaults to audio, the INVITE headers default to an empty dictionary, and the UUID defaults to nil so the call can be started without additional customization.

3.8.3 Nested Types

The media type used when starting an outgoing call.

Enum type	Enum cases	Definition
OutgoingCallMedia (enum)	*'audio'*	Start the call with audio media only.
	'video'	Start the call with audio and video media.

3.9 CallSession (class)

The call session represents one incoming or active call. It exposes call state, metadata, callbacks, and the basic call-control operations used during the call lifetime.

3.9.1 Properties

Property	Description
audioCodesSession	A temporary interoperability escape hatch that exposes the wrapped Objective-C call session when one exists. Use this only for migration or for features that are not yet exposed through Swift.
callbackQueue	The queue used to deliver call-session callbacks. Sessions created by `UserAgent` inherit UserAgent.callbackQueue when they are created.
sessionID	The SDK-assigned session identifier.
callState	The current call state.
terminationInfo	Information describing why the call ended, when available.
isDelayedOffer	A Boolean value that indicates whether the call arrived as a delayed offer. This is useful for UI decisions and media presentation choices.
duration	The current call duration.
callStartTime	The Unix timestamp, in whole seconds, when the call started. For outgoing calls this is captured when the call begins; for incoming calls it is captured when the call is answered.
isLocallyHeld	A Boolean value that indicates whether the local party placed the call on hold.
isRemotelyHeld	A Boolean value that indicates whether the remote party placed the call on hold.
isAudioMuted	A Boolean value that indicates whether outgoing audio is muted.
isVideoMuted	A Boolean value that indicates whether outgoing video is muted.
callUUID	The UUID associated with the call.
isOutgoing	A Boolean value that indicates whether this is an outgoing call.
hasVideo	A Boolean value that indicates whether the call has video media. This is read-only metadata for app UI decisions. Use the video methods and '`VideoStreamView`' to control and render video.
remoteContact	The remote party associated with the call.
onProgress	A callback invoked when call progress changes. Use it to observe updated state values such as call state, hold state, and duration.
onTerminated	A callback invoked when the call terminates.
onNotifyEvent	A callback invoked when the call receives a SIP NOTIFY event. The callback reports NOTIFY events received by the call session.
onCameraSwitched	A callback invoked after the active camera switches. The callback is delivered on '`callbackQueue`'.

3.9.2 Methods

Methods	Description
answer()	Answers the incoming call using the default SIP headers.
answer(headers:)	Answers the incoming call with custom SIP headers. Header names and values are passed through without validation.
reject()	Rejects the incoming call using the default SIP headers.
reject(headers:)	Rejects the incoming call with custom SIP headers. Header names and values are passed through without validation.
terminate()	Terminates the call.
hold()	Places the call on hold.
resume()	Resumes a held call.
sendDTMF(_ digit:)	Sends a single DTMF digit during the call. The SDK uses the shared DTMF configuration from SDKConfiguration.
sendDTMF(_ digits:)	Sends a string of supported DTMF characters during the call. Unsupported characters are ignored.
enableVideo(completion:)	Enables video for the call using attached '`VideoStreamView`' renderers. Add '`VideoStreamView`' instances for the streams that should be rendered. If the call is already connected as audio-only, this method starts video and lets the SDK perform media renegotiation.
stopVideo()	Stops local video capture and removes the local and remote video renderers.
switchCamera()	Switches between the front and back camera. The result is reported through '`onCameraSwitched`'.
muteVideo()	Mutes outgoing local video.
unmuteVideo()	Unmutes outgoing local video.

3.9.3 Nested Types

Enum type	Enum cases	Definition
State (enum)	*'undefined'*	The call state is not defined.
	'calling'	The outgoing call is being placed.
	'ringing'	The incoming or outgoing call is ringing.
	'connected'	The call is connected.

Struct type	Struct Property	Definition
TerminationInfo (struct)	*reason*	The high-level termination reason.
	sipStatusCode	The SIP status code associated with the termination.
	sipStatusText	The SIP status text associated with the termination.
	sipReasonHeaderValue	The value of the SIP Reason header, when present.
	sipMessage	The SIP message associated with the termination, when available.
init(reason:sipStatusCode:sipStatusText:sipReasonHeaderValue:sipMessage:)		Creates call termination details.

Enum type	Enum cases	Definition
Reason (enum)	*'undefined'*	The termination reason is not defined.
	'terminatedLocally'	The local party terminated the call.
	'terminatedRemotely'	The remote party terminated the call.
	'rejected'	The call was rejected.
	'timedOut'	The call timed out.
	'failed'	The call failed.

Struct type	Struct Property	Definition
NotifyEvent (struct)	*kind*	The notification event kind.
	dtmfString	The DTMF string associated with the event, when present.
init(kind:dtmfString:)		Creates a call notification event.

Enum type	Enum cases	Definition
Kind (enum)	*'undefined'*	An undefined notification event.
	'talk'	A talk event.
	'hold'	A hold event.
	'dtmf'	A DTMF event.
	'conference'	A conference event.
VideoStream (enum)		Video streams that can be rendered for a call session.
	'localPreview'	The local camera preview.
	'remote'	The remote peer video.
CameraPosition (enum)		Active camera positions.
	'front'	The front-facing camera.
	'back'	The back-facing camera.

DTMF enum	Definition
DTMF (enum)	The DTMF type represents the digits that can be sent during a call. Supported values are 0 through 9, *, and #.
init?(_ character:)	Creates a DTMF digit from a character. The initializer returns nil when the character is not supported.

3.10 VideoStreamView (struct)

A SwiftUI view that renders one video stream from a call session. One view instance is bound to one stream. Use separate instances to render the remote video and the local camera preview. The framework owns the platform-specific rendering container internally.

3.10.1 Initialization

init(session: CallSession, stream: VideoStream)

The 'session' parameter selects the call whose video is rendered and the 'stream' parameter selects the stream of that call.

3.11 AudioManager (class)

The audio manager coordinates audio session activation and audio route changes. It is separate from UserAgent.callbackQueue and uses its own callback queue for route-change notifications.

3.11.1 Properties

Property	Description
shared	The shared audio manager instance.
callbackQueue	The queue used to deliver audio route callbacks. The default queue is the main queue.
usesManualAudio	A Boolean value that indicates whether the application manually controls audio activation.
isAudioEnabled	A Boolean value that indicates whether the audio unit is enabled.
availableRoutes	The currently available audio output routes.
currentRoute	The active audio output route.
currentRoutingOptions	The currently applied audio routing options.
onAvailableRoutesChanged	A callback invoked when the set of available audio routes changes.
onCurrentRouteChanged	A callback invoked when the current audio route changes.

3.11.2 Methods

Methods	Description
activateAudioSession()	Activates the application audio session. The method throws when activation fails.
deactivateAudioSession()	Deactivates the application audio session. The method throws when deactivation fails.
configureAudioSession(_:)	Configures the audio session for the specified preset.
setAudioRoutingOptions(_:)	Applies audio routing options and returns whether the options were applied.
overrideOutputToSpeaker()	Overrides audio output to the loudspeaker.
stopOverridingOutputToSpeaker()	Stops overriding audio output to the loudspeaker.
allowBluetoothRouting()	Allows Bluetooth audio routing.
disallowBluetoothRouting()	Disallows Bluetooth audio routing.
audioSessionDidActivate(_:)	Notifies the audio manager that the system activated the audio session.
audioSessionDidDeactivate(_:)	Notifies the audio manager that the system deactivated the audio session.

3.11.3 Nested Types

Struct type	Struct Property	Definition
RoutingOptions (struct)	*none*	No routing options are enabled.
	overrideToLoudSpeaker	Overrides output to the loudspeaker.
	allowBluetooth	Allows Bluetooth audio routing.

Enum type	Enum cases	Definition
Route (enum)	*'earpiece'*	The earpiece route.
	'loudSpeaker'	The loudspeaker route.
	'bluetooth'	A Bluetooth audio route.
	'undefined'	No audio route is defined.
Preset (enum)	*'default'*	The default audio session preset.
	'voip'	A preset configured for VoIP calls.

3.12 SDKConfiguration (class)

The shared SDK configuration exposes settings that are not tied to a specific user agent or call session.

3.12.1 Properties

Property	Description
shared	The shared configuration instance.
version	The SDK version string.
localServerPort	The local server port used by the SDK.
dtmfConfiguration	The DTMF settings used when CallSession.sendDTMF(_:) is called.
videoCaptureConfiguration	The camera capture settings used when outgoing video is started.

3.12.2 Nested Types

Struct type	Struct Property	Definition
VideoCaptureConfiguration (struct)	*default*	The default camera capture settings: 640 by 480 pixels at 15 frames per second.
	width	Capture width in pixels.
	height	Capture height in pixels.
	frameRate	Capture frame rate in frames per second.
init(width:height:frameRate:)		Creates the camera capture settings.
DTMFConfiguration (struct)	*default*	The default DTMF settings: WebRTC delivery, 100 millisecond duration, and 70 millisecond gap.
	method	The DTMF delivery method.
	durationMilliseconds	The duration of each DTMF event in milliseconds.
	intervalGapMilliseconds	The interval between DTMF events in milliseconds when WebRTC DTMF delivery is used.
init(method:durationMilliseconds:intervalGapMilliseconds:)		Creates the DTMF delivery settings.

Enum type	Enum cases	Definition
DTMFDeliveryMethod (enum)	*'webRTC'*	Send DTMF using WebRTC RTP events.
	'sipInfo'	Send DTMF using SIP INFO messages.

4 Objective-C SDK API Reference

The API consists of the following:

- Main Classes:
  - AudioCodesUA – AudioCodes User Agent (*Singleton*) – see Section 4.1
  - AudioCodesSession – For call representation (*Interface*) – see Section 4.2
  - WebRTCAudioManager – Class for managing audio routes – see Section 4.3
  - ACNativeCallService (iOS only) – Class for integrating call management with the OS native telephony features – See Section 4.11
- Helper Classes:
  - ACConfiguration – (Optional) Class for general configuration options – see Section 4.4
  - VideoConfiguration – (Optional) Class for video configuration – see Section 4.5
  - DTMFOptions – Class for settings DTMF options – see Section 4.6
  - RemoteContact – Class representing the remote contact – see Section 4.7
- Listener Interfaces:
  - AudioCodesEventListener – Event listener for incoming calls and login state changes – see Section 5.1
  - AudioCodesSessionEventListener – Event listener for call related events – see Section 5.2
  - WebRTCAudioRoutesListener – Event listener for audio route events – see Section 5.3

4.1 AudioCodesUA

This is used to initialize the framework before starting to make and receive calls. It is mostly used to initialize the WebRTC engine and register it to the service.

```
@interface AudioCodesUA : NSObject
@property (nonatomic, weak) id <AudioCodesEventListener> delegate;
@property (nonatomic, assign) BOOL useSessionTimer;
@property (nonatomic, assign) int regExpires;
@property (nonatomic, assign) ACLogLevel logLevel;
@property (nonatomic, assign) id<ACLoggerProtocol> logger;
@property (nonatomic, strong) NSString* userAgent;
@property (nonatomic, assign) BOOL verifyServerCertificate;
@property (nonatomic, strong) NSString* caCertFilePath;
@property (nonatomic, assign) BOOL disconnectOnBrokenConnection;
@property (nonatomic, assign) BOOL contactRewrite;
@property (nonatomic, strong) SipHeadersDictionary*
registerExtraHeaders;
@property (nonatomic, strong) SipHeadersDictionary*
inviteExtraHeaders;
@property (nonatomic, readonly) NSArray <AudioCodesSession*>*
sessions;
+ (instancetype) getInstance;
- (void) setServerConfig:(NSString*)proxyAddress
                    port:(int)port
            serverDomain:(NSString*)serverDomain
               transport:(ACTransportType)transport
              iceServers:(NSArray
<id<ACRTCIceServer>>*)iceServers;
```

```
- (void) setAccount:(NSString*)userName
        displayName:(NSString*)displayName
           password:(NSString*)password
           authName:(NSString*)authName;
- (void) setPushNotificationsTeamId:(NSString*)teamId
                           bundleId:(NSString*)bundleId
                          apnsToken:(NSString*)apnsToken
                          voipToken:(NSString*)voipToken;
- (void) setOauthToken:(NSString*)accessToken;
- (void) setConnectionBinding:
(ACNetworkConnectionAttributes*)initialPrefs
- (void) login;
- (void) login:(BOOL)autoRegister;
- (void) logout:(ACUALogoutMode)mode;
- (void) logout;
- (void) logoutWithForceClose:(BOOL)forceClose;
- (void) handleNetworkChange:
(ACNetworkChangeAttributes*)attributes;
- (AudioCodesSession*) call:(RemoteContact*)call_to
                  withVideo:(BOOL)withVideo
              inviteHeaders:(SipHeadersDictionary*)inviteHeaders;
- (NSString*) sendInstantMessage:(NSString*)message
to:(RemoteContact*)contact;
@end
```

4.1.1 Standard Methods / Properties

4.1.1.1 getInstance

Returns the singleton object instance of class AudioCodesUA.

4.1.1.2 setServerConfig

Configures the server.

Parameters

- proxyAddress [NSString]: Address of server
- port [integer]: Port of the proxy server address
- serverDomain [NSString]: Domain name to which to register
- transport [ACTransportType]: Transport for connection to the server – UDP/TCP/TLS]
- iceServers [NSArray <id<ACRTCIceServer>>]: List of STUN and TURN servers of the ACRTCIceServer protocol type, which can be created using the *ACRTCIceServerFactory* class. For more information on creating *ACRTCIceServer* instances, refer to the developer documentation in the *ACRTCIceServer.h* header file.

> iceServers are only applicable when real-time media is concerned, and so cannot be used without MVWebRTCInterface.xcframework being linked.

Return Values

N/A

4.1.1.3 setAccount

Defines the account details.

Parameters

- userName [string]:, User name]
- password [string]:Authenticating user password
- displayName [string]: Display name for the user]
- authName [string]: Authorization user name, optional parameter. If nil, the user authorization with be the userName value.

Return Values

N/A

4.1.1.4 id <AudioCodesEventListener> delegate

Sets / gets the delegate object.

Setter Parameters / Getter Return Value

- delegate [id <AudioCodesEventListener>]: Instance implementation of the AudioCodesEventListener interface that holds the methods to be triggered; see Section 5.1 for details on how it is defined; see also Section 6.2 for an example]. API Callbacks/ Listeners User Agent: Set

4.1.1.5 login:(BOOL)autoRegister

Initiates the SIP account with the configuration setters applied. It must be called after `setServerConfig` and `setAccount`.

Parameters

- autoRegister [boolean]: Determines whether or not to perform SIP Registration once the account is initiated:
  - **True:** The SDK logs in to the service and the delegate method `loginStateChanged` is called once the SIP Registration has been completed.
  - **False:** The SDK does not log in to the service, however calls can be made. The method should be regarded as synchronous without delegate callback.

Return Values

N/A

4.1.1.6 login

Initiates the service and performs registration. This is a convenience method which calls login (see Section 4.1.1.5) with the `autoRegister` parameter set to **Yes**.

Parameters

N/A

Return Values

N/A

4.1.1.7 logout:(ACUALogoutMode)mode

Performs logout according to the requested mode.

Parameters

- mode [ACUALogoutMode]: Enum value that determines the behavior of logout
  - ACUALogoutModeGracefulShutdown – unregister and shut down the SIP account, equivalent to the behavior of logoutWithForceClose:NO, resp. logout.
  - ACUALogoutModeForceShutdown – force shut down the SIP account and close all connections without sending un-REGISTER, equivalent to the behavior of logoutWithForceClose:YES.
  - ACUALogoutModeUnregisterOnly – unregister only and keep active calls.

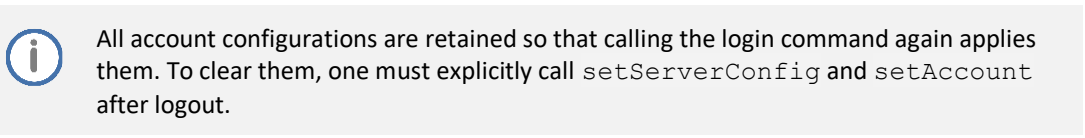
Return Values

N/A

4.1.1.8 logoutWithForceClose:(BOOL)forceClose

Compatibility version for legacy boolean parameter logout semantics, equivalent behavior to the new logout:ACUALogoutModeForceShutdown and logout:ACUALogoutModeGracefulShutdown.

Performs SIP Un-REGISTER if necessary and shuts down the SIP account.



All account configurations are retained so that calling the login command again applies them. To clear them, one must explicitly call `setServerConfig` and `setAccount` after logout.

Parameters

- forceClose [boolean]:
  - YES is equivalent to the behavior of logout:ACUALogoutModeForceShutdown - huts down the SIP account and closes all connections without sending un-REGISTER, unSUBSCRIBE or any other message. Note that setting `forceClose` to YES still yields the `loginStateChanged` delegate call.
  - NO is equivalent to the behavior of logout:ACUALogoutModeGracefulShutdown - sends SIP Un-REGISTER, then shut down the SIP account.

Return Values

N/A

4.1.1.9 logout

Convenience method which calls logout: `ACUALogoutModeGracefulShutdown`.

Parameters

N/A

Return Values

N/A

4.1.1.10 call

Initiates an outgoing call.

Parameters

- call_to [RemoteContact]: Destination address/number.
- withVideo [boolean]: 'True' if the call is initiated with video.
- inviteHeaders [SipHeadersDictionary]: Includes a list of headers defined as key/value pairs, where each key is added as a header to the SIP INVITE with the specified value.

Return Values

- session [AudioCodesSession]: A call session object defined here.

4.1.1.11 sendInstantMessage

Initiates a SIP Instant Message to a remote contact, according to *RFC 3428 – Session Initiation Protocol (SIP) Extension For Instant Messaging*.

Status updates for a sent message arrive via the instantMessageStatus:messageId: delegate method.

Parameters

- message [string]: The message string.
- contact [RemoteContact]: The message destination.

Return Values

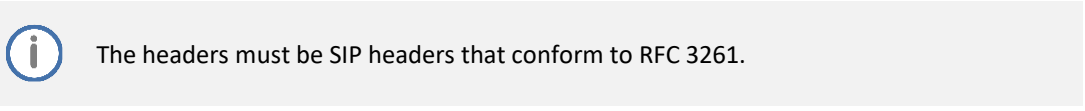
- message ID string, [string]: Used by the delegate method instantMessageStatus:messageId: to notify of status updates.

4.1.2 Advanced Methods / Properties

The advanced methods are optional. They provide an extra level of flexibility to the API, which is based on SIP (Session Initiation Protocol). Developers who are familiar with SIP can make use of the advanced methods.

4.1.2.1 SipHeadersDictionary* registerExtraHeaders

Allows the adding of additional headers to the registration request.



The headers must be SIP headers that conform to RFC 3261.

Setter Parameter / Getter Return Value

- SipHeadersDictionary: SIP headers are defined as key/value pairs, where each key is added as a header to the registration request with the specified value.

4.1.2.2 SipHeadersDictionary* inviteExtraHeaders

Allows adding additional headers to the INVITE request or response.

The headers must be SIP headers that conform to RFC 3261.

Setter Parameters / Getter Return Value

- SipHeadersDictionary: SIP headers are defined as key/value pairs, where each key is added as a header to the SIP INVITE request with the specified value.

4.1.2.3 NSString* userAgent

Gets / Sets user-agent string, used to build the SIP header User-Agent.

Setter Parameters / Getter Return Value

- userAgent: [string]: Text describing the SIP user agent.

4.1.2.4 BOOL verifyServerCertificate

This is applicable to TLS only. Determines whether or not the SIP TLS transport should verify the server certificate. The default value is False.

Setter Parameters / Getter Return Value

- verifyServerCertificate [boolean]: If 'False', then the server certificate is not validated. If so, the TLS connection can operate with untrusted server certificates. Otherwise, the server certificate must pass validation, for the TLS connection to succeed.

4.1.2.5 NSString* caCertFilePath

The path to a custom Certificate Authority's root certificate .pem file, to be used for TLS connections, for validating server certificates. The .pem file can be either a single root certificate, or a bundled certificate chain.

The default value is nil, in which case the default OS trust store is used for validation.

The value must be a valid full path to a file within the application bundle. For example:
[[NSBundle mainBundle] pathForResource:@"custom-root-cert-filename" ofType:@"pem"];

Setter Parameters / Getter Return Value

- caCertFilePath: [string]: The path in the application bundle, of the custom root certificate file.

4.1.2.6 BOOL contactRewrite

Updates the transport address and the Contact header of the REGISTER request. When this option is enabled, the SDK keeps track of the public IP address from the response of the REGISTER request. Once it detects that the transport address has changed, it unregisters the current Contact, updates the Contact with the transport address learned from the Via header, and registers a new Contact to the registrar. It also updates the public name of the UDP transport if STUN (Session Traversal Utilities for NAT) is configured.

Default: 'False'

Setter Parameters / Getter Return Value

- enable [boolean]:
  - **True:** The library tracks the public IP address from the response of the REGISTER request.
  - **False:** The library does not track the public IP address from the response of the REGISTER request.

Return Values

N/A

4.1.2.7 BOOL disconnectOnBrokenConnection

Changes the method by which call handover is handled by the SDK. The default value is TRUE. Note that SBC configuration is required to allow the call to continue on broken media.

Setter Parameters / Getter Return Value

- True / False [boolean]:
  - **True:** Disconnects the call when a network connection error occurs in the media stream.
  - **False:** Allows the call to continue when there is a broken network connection in the media stream.

4.1.2.8 int regExpires

Gets / Sets the default registration interval. The default value is 600 seconds.

Setter Parameters / Getter Return Value

- regExpires [integer (seconds)]

4.1.2.9 BOOL useSessionTimer

Allows enabling session timers in the call session.

Setter Parameters / Getter Return Value

- useSessionTimer [boolean]: If 'False', then the session timers will be not be enabled. Otherwise (default value) session timers are optionally supported. e.g., the SBC initiates session timers if configured.

4.1.2.10 ACLogLevel logLevel

Sets / Gets the global log level used by the SDK. The value is applied to SDK logs, SIP/PJSIP logs, and WebRTC logs. For separate control, use `sdkLogLevel`, `sipLogLevel`, and `webRTCLogLevel`.

For security reasons, release builds may want to set the log level lower.

For WebRTC logs, `ACLogLevelVerbose` maps to info-level logging, while all other values map to error-level logging.

If not set, the default log level is `ACLogLevelError`.

Setter Parameters / Getter Return Value

- logLevel [ACLogLevel]: Enumeration value which represents the log level.

4.1.2.11 ACLogLevel sdkLogLevel

Sets / Gets the log level for logs emitted by the AudioCodes SDK code. This setting controls SDK-log level messages only; To configure SIP/PJSIP logs, use sipLogLevel; to configure WebRTC logs, use `webRTCLogLevel`, or use `logLevel` to configure all logging areas together.

If not set, the default log level is *ACLogLevelError.*

Setter Parameters / Getter Return Value

- logLevel [ACLogLevel]: Enumeration value which represents the log level.

4.1.2.12 ACLogLevel sipLogLevel

Sets / Gets the log level for SIP/PJSIP logs, including SIP signaling messages such as RX/TX traffic. This setting controls SIP/PJSIP logging only; for AudioCodes SDK code logs use `sdkLogLevel`, for WebRTC logs use `webRTCLogLevel`, or use `logLevel` to set all logging areas together.

If not set, the default log level is *ACLogLevelError.*

Setter Parameters / Getter Return Value

- logLevel [ACLogLevel]: Enumeration value which represents the log level.

4.1.2.13 ACLogLevel webRTCLogLevel

Sets / Gets the log level for WebRTC logs, including WebRTC adapter logs and native Google WebRTC logs. This setting controls WebRTC logging only. To configure AudioCodes SDK code logs, use `sdkLogLevel`; to configure SIP/PJSIP logs, use `sipLogLevel`; or use `logLevel` to set all logging areas together.

If not set, the default log level is *ACLogLevelError.*

Setter Parameters / Getter Return Value

- logLevel [ACLogLevel]: Enumeration value which represents the log level.

4.1.2.14 id<ACLoggerProtocol> logger

Changes the logger used by the SDK.

If set, then all the SDK log messages will go through the `acLogMessage` method, except for WebRTC internal log entries, that would be printed to the console if the `logLevel` property is equal to `ACLogLevelVerbose`.

If not set, by default the SDK uses unified-logging to print the log entries to console, and distinguishes log entry categories. See "Viewing SDK Logs" in Section Usage Notes.

Setter Parameters / Getter Return Value

- logger [id <ACLoggerProtocol>]: Instance object implementing `ACLoggerProtocol`

4.1.2.15 handleNetworkChange

Handles network changes when called. This function re-registers the client and re-establishes the audio sessions when the network has been changed.

This function must be explicitly called by the client application. The SDK does not automatically detect a network change. Ideally, this function must be called when the network is reconnected and not when it is disconnected.

Parameters

- attributes [ACNetworkConnectionAttributes]: Optional attributes for managing network change handling.
  - If a SIP connection binding has been applied via `setConnectionBinding`, then the value in attributes.localAddressFamily MUST be either `ACNetworkAddressFamilyIPV4` or `ACNetworkAddressFamilyIPV6` to maintain registrations or calls on network changes as best as possible.

Return Values

N/A

4.1.2.16 setConnectionBinding

Configures or disables the method by which the user-agent may bind its SIP connections. See section 5.17 for example usage.

This method must be called BEFORE calling *login*.

Discussion:

SIP connection binding is the behavior which forces the SIP account to reuse the current SIP connection for all outgoing messages.

Once SIP connection binding is applied, binding cannot be cancelled until the account is shut down. Binding can only update on a network change, to be re-applied to a new connection (see `handleNetworkChange` note below).

Connection binding is performed as follows:

- if `initialPrefs` is nil, then no connection binding is performed.
- if *initialPrefs.localAddressFamily* is `ACNetworkAddressFamilyUnspecified`, then the user agent will adapt to any IP-address family by waiting for a connection to be established, and then perform binding for the SIP account. This achieves the best support for maintaining calls and registration during network changes.
- if *initialPrefs.localAddressFamily* is either `ACNetworkAddressFamilyIPV4` or `ACNetworkAddressFamilyIPV6`, then the user agent binds the SIP account connection before it is established.

handleNetworkChange: Once called with the "attributes" parameter: If attributes.localAddressFamily is `ACNetworkAddressFamilyIPV4` or `ACNetworkAddressFamilyIPV6`, then binding will re-apply according to the steps above. Otherwise, connection binding remains unchanged.

Using this method is generally not recommended. Binding the SIP connection is recommended only for specific environments, because it limits the dynamic nature of connectivity to the server per transaction, as well as limits the ability to maintain registrations, subscriptions and calls on network changes between IP-address types.

When using the SIP account to make calls without registration (not calling the `connectSipAccount` method), connection binding will work only if initialPrefs.localAddressFamily is either `ACNetworkAddressFamilyIPV4` or `ACNetworkAddressFamilyIPV6`, because the `ACNetworkAddressFamilyIPV4` or `ACNetworkAddressFamilyUnspecified` value defers binding in a manner that is not suitable without registration.

Parameters

- initialPrefs [ACNetworkConnectionAttributes]: The network connection attributes that determine the initial binding behavior of the SIP connection.

Using a *nil* value removes SIP connection binding.

Return Values

N/A

4.1.2.17 NSArray <AudioCodesSession*>* sessions

Gets the current session list.

Parameters

N/A

Return Values

sessions [NSArray<AudiocodesSession*>: List of current existing sessions

4.1.2.18 setPushNotification

Allows the SDK to use push for incoming calls. This is an optional method. If set, the SDK sends the push credentials to the SBC which allows the SBC to send Push messages for incoming calls and registration refresh (see Demo client for example).

This method sets the Push parameters for the PNS according to:
Push Notification with the Session Initiation Protocol (SIP).
For more information, see https://tools.ietf.org/html/draft-ietf-sipcore-sip-push-20.

The values are stored in permanent memory and are used by the WebRTC SDK until the values are reset to nil values. It is recommended to call this method before calling Login, as calling this method afterwards, causes a re-register of the SIP stack.

Setting any of the parameters with a nil value, disables the Push Notifications entries in SIP.

Parameters

- teamId [string]: Typically should be the unique Apple Developer Team Identifier.
- bundleId [string]: The application Bundle Identifier. If this method is called from within an app extension, then this MUST be the bundle identifier of the **containing application.**
- apnsToken [string]: The APNS Push Token string, used for REGISTER refresh push notifications.
- voipToken [string]: The VOIP Push Token string, used for high-priority incoming call push notifications

Return Values

N/A

4.1.2.19 setOauthToken

Allows the SDK to use Oauth authentication for registration to the service. This is an optional method. The SDK adds an authorization header with the supplied access token (the SBC needs to be configured to use Oauth authorization as well).

Parameters

- **accessToken** [string]: Access token as received from the Oauth server (see Demo client for example on Oauth registration).

Return Values

N/A

4.2 AudioCodesSession

Represents a call session using the following scenarios:

- When initiating a call via the AudioCodesUA
- When receiving a call back of an incoming call

Syntax

```
@interface AudioCodesSession : NSObject
@property (nonatomic, weak) id<AudioCodesSessionEventListener>
delegate;
@property (nonatomic, readonly) int sessionID;
@property (nonatomic, assign, getter=isAudioMuted) BOOL muteAudio;
@property (nonatomic, assign, getter=isVideoMuted) BOOL muteVideo;
@property (nonatomic, readonly) BOOL isOutgoing;
@property (nonatomic, readonly) BOOL hasVideo;
@property (nonatomic, readonly) BOOL isLocalHold;
@property (nonatomic, readonly) BOOL isRemoteHold;
@property (nonatomic, readonly) BOOL isDelayedOffer;
@property (nonatomic, readonly) CallState;
@property (nonatomic, readonly) NSInteger duration;
@property (nonatomic, readonly) NSInteger callStartTime;
@property (nonatomic, readonly) RemoteContact* remoteNumber;
@property (nonatomic, readonly) CallTransferState transferState;
@property (nonatomic, readonly) RemoteContact* transferContact;
@property (nonatomic, readonly) TerminationInfo *terminationInfo;
@property (nonatomic, readonly) NSUUID *callUUID;
@property (nonatomic, strong) id userData;
- (void) answer:(SipHeadersDictionary*)headers;
- (void) reject:(SipHeadersDictionary*)headers;
- (void) terminate;
- (void) sendDTMF:(DTMF)dtmf;
- (void) hold:(BOOL)hold;
- (void) transferCall:(RemoteContact*)remoteContact;
- (void) attendedTransferCall:(AudioCodesSession*)destination;
- (void) switchCamera;
- (void) showVideoLocalView:(UIView*)localView
remoteView:(UIView*)remoteView
completion:(ACTaskCompletion)completion;
- (void) stopVideo;
- (void) sendInfo:(NSString*)body
contentType:(NSString*)contentType;
@end
```

4.2.1 Standard Methods / Properties

4.2.1.1 int sessionID;

Retrieves the internal identifier for the session. This identifier can be used in case there is more than one session.

Return Values

- sessionID [integer]: ID of the session

4.2.1.2 answer

Initiates the object and establishes the call. This method is only valid for incoming calls.

Parameters

- headers [SipHeadersDictionary]: List of headers with a key/value where each key is added as a header to the SIP response with the specified value.
- with video [boolean]:
  - 'True': The call is answered with video and video unmuted. Both sides will see each other.
  - 'False' :The call is answered with video; however, video is muted. The local side sees the remote video; however, the remote side cannot see the video of the local side.

Return Values

N/A

4.2.1.3 reject

Rejects a call. This method is only valid for incoming calls.

Parameters

- headers [SipHeadersDictionary]: List of headers with a key/value where each key is added as a header to the SIP response with the specified value.

Return Values

N/A

4.2.1.4 Terminate

Terminates an active call. This method is only valid for outgoing and established calls.

Parameters

N/A

Return Values

N/A

4.2.1.5 BOOL muteAudio (getter=isAudioMuted)

Sets / Gets the status of the audio mute (on/off).

Setter Parameter / Getter return value

- muteAudio [boolean]: 'True' to mute audio; 'False' to unmute audio.

4.2.1.6 BOOL muteVideo (getter=isVideoMuted)

Sets / Gets the status of the video mute (on/off).

Setter Parameter / Getter return value

- muteVideo [boolean]: 'True' to mute video; 'False' to unmute video

4.2.1.7 sendDTMF

Sends a DTMF character.

Parameters

- dtmf [DTMF]: Enumeration value which represents a DTMF character.

Return Values

N/A

4.2.1.8 BOOL isOutgoing

Checks if a call is outgoing.

Parameters

N/A

Return Values

- [boolean]*:* 'True' if outgoing, 'False' if incoming.

4.2.1.9 BOOL hasVideo

Checks if a call includes video.

Parameters

N/A

Return Values

- [boolean]: 'True' if the call includes video, 'False' if the call includes audio only.

4.2.1.10 CallState

Gets the call state of the session.

Parameters

N/A

Return Values

- callState [CallState]*:* Enumeration value which represents the current call state.

4.2.1.11 TerminationInfo terminationInfo

Gets the termination information of the session, if terminated.

Parameters

N/A

Return Values

- terminationInfo [TerminationInfo]: Object of the TerminationInfo type, representing termination-related data. If the call is not terminated, the return value is *nil*.

4.2.1.12 NSInteger duration

Defines the call duration in seconds. It is '-1' if the call has not yet been established.

Parameters

N/A

Return Values

- duration [integer]: Call duration in seconds. This value is '-1' if the call has not yet been established.

4.2.1.13 BOOL isLocalHold

Parameters

N/A

Return Values

- [boolean]: 'True' if call is on local hold, otherwise 'False'.

4.2.1.14 BOOL isRemoteHold

Parameters

N/A

Return Values

- [boolean]: 'True' if the call is placed on hold by remote side, otherwise 'False'.

4.2.1.15 BOOL isDelayedOffer

Parameters

N/A

Return Values

- [boolean]: 'True' if this is an incoming call with a delayed offer SDP, meaning that no SDP offer was included in the incoming SIP INVITE message.

4.2.1.16 id userData

Sets / Gets user-created data to be attached to the session. The reference is removed from the session after the session is terminated.

Setter Parameter / Getter Return Value

- userData: id type for any type of data

4.2.1.17 hold

Sets call on hold (or un-hold). The `callProgress` callback in AudioCodesSessionEventListener indicates when the call has been placed on hold/unhold. Use the `isLocalHold` property to retrieve the status.

Parameters

- Hold [boolean]: Set call to hold

Return Values

N/A

4.2.1.18 switchCamera

Switches the camera between the front and back camera. This method requires the device to have two cameras. A successful camera switch is returned in the `cameraSwitched` callback in AudioCodesSessionEventListener.

Parameters

N/A

Return Values

N/A

4.2.1.19 (void) showVideoLocalView:(UIView*)localView remoteView:(UIView*)remoteView completion:(ACTaskCompletion)completion

Displays a video during a call, with the provided UIView objects for local and remote video rendering. These objects must be empty views to act as containers to the video rendering.

- If the call is currently defined to be audio only, then a re-INVITE is initiated to negotiate the video media. The AudioCodesSession will invoke the `callStateChanged` event after the video re-negotiation is complete.
- For privacy considerations, local video camera capture does not begin if the local view is nil. The user must be able to see the video that is captured by the camera. So if `localView` is nil, no video is captured locally and sent to the remote side.

Parameters

- localView [UIView: iOS standard UIView object]. This parameter can be nil, in which case no local video is captured.
- remoteView [UIView, iOS standard UIView object]. This parameter can be nil, in which case the remote video is not displayed].
- completion [ACTaskCompletion*:* Optional completion block to be called when video rendering setup is complete.

Return Values

N/A

4.2.1.20 stopVideo

Stops the capturing of the video and removes the remote and local renderer. In order to start the video again, `showVideo` needs to be called. This call has no effect on the local and remote video UIView objects that have been provided for `showVideo`.

Parameters

N/A

Return Values

N/A

4.2.1.21 id<AudioCodesSessionEventListener> delegate

Sets / Gets an event listener to listen for session events. The client application might add multiple listeners. The listeners will receive events until they are either removed or the session is terminated.

Setter Parameter / Getter Return Value

- delegate [id <AudioCodesSessionEventListener>]: Implementation object of the AudioCodesSessiobEventListener protocol.

4.2.1.22 RemoteContact *remoteNumber

Gets the call destination details within a `RemoteContact` type. Note that for the incoming call transfer process, the `remoteNumber` value changes after the call transfer operation completes successfully.

Parameters

N/A

Return Value

- remoteNumber [RemoteContact]: represents details of the remote destination

4.2.1.23 transferCall (Blind Transfer)

Transfers the other side of the current AudioCodesSession to the remote contact supplied.

This is a blind transfer and should be used when there is only one session. (See Demo Client for usage)

Once the call transfer has been initiated, status updates are delivered via the `callProgress` delegate method. The `transferContact` and `transferState` properties are then updated to represent the transfer operation status.

Upon successful completion, the AudioCodesSession terminates automatically, and the `callTerminated` delegate method is invoked.

Upon transfer failure, the AudioCodesSession resumes the current call, and the `callProgress` delegate method is invoked, with the *connected* call state.

Parameters

- remoteContact [RemoteContact]: Contains the transfer destination data.

Return Values

N/A

4.2.1.24 attendedTransferCall (Attended Transfer)

Transfers the other side of the current AudioCodesSession to the AudioCodesSession supplied. This is an attended transfer and should be used when there is more than one session. (See Demo Client for usage). Once the call transfer has been initiated, status updates are delivered via the `callProgress` delegate method. The `transferContact` and `transferState` properties are then updated to represent the transfer operation status.

Upon successful completion, the AudioCodesSession terminates automatically, and the `callTerminated` delegate method is invoked.

Upon transfer failure, the AudioCodesSession resumes the current call, and the `callProgress` delegate method is invoked with the *connected* call state.

Parameters

- transferToSession [AudioCodesSession]: Transfer destination call. AudioCodesSession to which the other side of the current call is transferred. For example, the other side of the current AudioCodesSession tries to replace this call with a call to the number of the supplied AudioCodesSession.

Return Values

N/A

4.2.1.25 RemoteContact *transferContact

Retrieves the call destination details for the call during a call transfer operation. Valid when the call transfer state is any value other than TRANSFER_STATE_UNDEFINED.

Parameters

N/A

Return Value

- transferContact [RemoteContact]: The transfer contact. This parameter can be the contact to which the other side of the current call is transferred or the remote contact to which this call is transferred.

4.2.1.26 CallTransferState transferState

Gets the status of a call transfer operation. Default is TRANSFER_STATE_UNDEFINED, denoting that there is no ongoing transfer process. Whenever this property value is updated, the `callProgress` delegate method is invoked.

Parameters

N/A

Return Values

- Transfer state: `CallTransferState` is the state of the transfer for this AudioCodesSession. Possible states:
  - **TRANSFER_STATE_UNDEFINED:** No transfer is in progress.
  - **TRANSFER_REQUEST_RECEIVED_IN_PROGRESS:** The other side has sent a transfer request, `transferContact` returns the contact to whom this call is transferred.
  - **TRANSFER_REQUEST_RECEIVED_FAILED:** The other side has sent a transfer request, however the transfer did not succeed.
  - **TRANSFER_REQUEST_RECEIVED_SUCCEEDED:** The other side has sent a transfer request and the transfer succeeded.
  - **TRANSFER_REQUEST_SEND_IN_PROGRESS:** This side has sent a transfer request which is being processed. `transferContact` returns the contact to whom the other side of this call is transferred.
  - **TRANSFER_REQUEST_SEND_FAILED:** This side has sent a transfer request which has failed.
  - **TRANSFER_REQUEST_SEND_SUCCEEDED:** This side has sent a transfer request which has succeeded.
  - **TRANSFER_REPLACED:** (Applicable for Attended Transfer, currently not supported) This AudioCodesSession is in a call with a remote party and the remote party has been replaced (side C in an attended transfer). `remoteContact` returns the new number to where the call is transferred.

4.2.1.27 NSUUID *callUUID

Defines an auto-generated UUID value associated with the call. It is used primarily for the SDK's built-in integration with the CallKit framework.

4.2.1.28 sendInfo

Sends a SIP INFO request within the session.

Parameters

- body [string]: The message body is converted to a string. e.g., a JSON structure has to be converted to a JSON-string.
- contentType [string] The SIP content-type header value. Has to be a valid MIME type, for example: “application/json”.

Return Values

N/A

4.3 WebRTCAudioManager

Defines WebRTC SDK Audio management. This class handles audio routing during WebRTC calls, as well as manual audio management for manually using CallKit.

Syntax

```
@interface WebRTCAudioManager : NSObject
@property (nonatomic, weak) id <WebRTCAudioRoutesListener>
delegate;
@property (nonatomic, readonly) AudioRoutingOptions
currentRoutingOptions;
+ (WebRTCAudioManager*) getInstance;
- (NSArray<AudioRouteNumber*>*) getAvailableAudioRoutes;
- (AudioRoute) getAudioRoute;
- (BOOL) setAudioRoute:(AudioRoutingOptions)options;
- (BOOL) overrideAudioRouteToSpeaker:(BOOL)enable;
- (BOOL) routeAudioToEnableBluetooth:(BOOL)enable;
// Manual Audio Management
@property (assign, nonatomic) BOOL useManualAudio;
@property (assign, nonatomic, getter=isAudioEnabled) BOOL
audioEnabled;
- (NSError*) setActiveAudioSession:(BOOL)setActive;
- (NSError*) configureAudioSession:(ACAudioPreset)preset;
- (void) audioSessionDidActivate:(AVAudioSession*)audioSession;
- (void) audioSessionDidDeActivate:(AVAudioSession*)audioSession;
@end
```

4.3.1 Notes on iOS Audio Routing

The general approach for audio routing in iOS is that iOS includes different behavioral patterns for routing audio in various schemes (e.g., Default mode, Audio calls, Playback, Recording) and the audio routing is determined internally by iOS as a function of the following variants:

1. The desired audio scheme (called “Audio Category / Mode”)
2. The currently available audio input / output hardware
3. General preferences whether to override audio to Loudspeaker, to allow Bluetooth, to mix audio with system playback.

This product provides functionality to control the 3rd listed variant; to make audio routing control as streamlined as possible, given iOS audio routing is non-deterministic. For instance, there is no method for explicitly setting the audio route into a specific route (e.g., `setAudioRoute` Bluetooth), rather we provide the `setAudioRoute` method with flag bitmask for preferences, because iOS audio routing control only allows for stating preferences to allow Bluetooth if the hardware is available.

4.3.2 Notes on Using CallKit

The SDK provides the `ACNativeCallService` class as the recommended means to utilize the CallKit framework in the application. However, one can opt to interface the CallKit framework manually. In that case, special audio management must be performed in the various flows involving CallKit.

The `WebRTCAudioManager` provides such functionality. (See “Manual Audio Management”)

4.3.3 Standard Methods / Properties

4.3.3.1 getInstance

Gets the singleton instance of the `WebRTCAudioManager` class.

Parameters

N/A

Return Values

- instance [WebRTCAudioManager]: Singleton object instance

4.3.3.2 id <WebRTCAudioRoutesListener> delegate

Gets / Sets a listener for listening to updates in the current audio route and available audio routes.

Setter Parameter / Getter Return Value

- delegate [id <WebRTCAudioRoutesListener>]: Implementation instance of the `WebRTCAudioRoutesListener` protocol.

4.3.3.3 setAudioRoute

Sets the audio route. This method changes the audio route of the device. This function generally should be used during a call. The audio is only routed if the new audio route is available.

Parameters

- options [AudioRoutingOptions]: Flags bitmask describing the audio routing preferences.

Return Values

- [boolean] 'True' : If the new audio route was successfully applied.
- 'False': If the new audio route is not successful.

4.3.3.4 getAudioRoute

Gets the current audio route.

Parameters

N/A

Return Values

- audioRoute [AudioRoute]: Enumeration value which represents the current audio route.

4.3.3.5 getAvailableAudioRoutes

Gets the available audio routes.

Parameters

N/A

Return Values

- audio routes [NSArray<AudioRouteNumber*>*]: Array of NSNumber objects representing the available audio routes' enumeration values.

4.3.3.6 overrideAudioRouteToSpeaker

Sets to override / disable overriding of the audio routing to the Loudspeaker.

Parameters

enable [boolean]: 'True' to override audio to speaker from any current route, 'False' to stop overriding to speaker and resume regular audio routing.

Return Values

- [boolean]: True: If the new audio route was successfully applied.
- False: If the new audio route is not successful.

4.3.3.7 routeAudioToEnableBluetooth

Sets allow / not allow for audio routing to Bluetooth if the Bluetooth audio route is available.

Parameters

enable [boolean]: 'True' to allow audio routing to Bluetooth, 'False' to prevent audio from routing through Bluetooth.

Return Values

- [boolean]: True: If the new audio route was successfully applied.
- False: If the new audio route is not successful.

4.3.4 Manual Audio Management

4.3.4.1 BOOL useManualAudio

Sets / Gets the property value to determine whether audio is managed manually for calls. Relevant for when using the CallKit framework manually (i.e., without utilizing the `ACNativeCallService` class).

Setter Parameter / Getter Return Value

- useManualAudio [Boolean]:'True' for manually managing audio, 'False' for having the SDK manage audio for calls

4.3.4.2 BOOL audioEnabled

Enables / Disables the audio unit, which is responsible for the VOIP real-time audio processing. It is only applicable if `useManualAudio` is True, and if CallKit is used manually.

Setting this value to 'True' is required upon CallKit's delivery of the **didActivateAudioSession** event.

Setting this value to False is required upon CallKit's delivery of the **didDeactivateAudioSession** event.

Setter Parameter / Getter Return Value

- audioEnabled [Boolean]:'True' for activating the audio unit, 'False' deactivating the audio unit.

4.3.4.3 setActiveAudioSession

Manually activates / deactivates the App's audio session.

Parameters

- setActive [Boolean]:'True' for activating the audio session, 'False' deactivating the audio session.

Return Values

- error [NSError*]: Error object for error, or nil for success.

4.3.4.4 configureAudioSession

Configures the App's audio session to adjust audio routing and hardware configuration, for a given preset. This affects the audio session's audio category, audio mode and category options.

Parameters

- preset [ACAudioPreset]: Audio preset to configure the app's audio. Can be one of the following values:
  - **ACAudioPresetDefault** – Configure audio to be ready for calls in idle state, where the audio route defaults to the Loudspeaker, and external Bluetooth audio devices are disabled.
  - **ACAudioPresetVOIP** – Configure audio to be ready for calls in active state, where audio route defaults either to the earpiece or to an externally connected audio device, and Bluetooth connectivity is enabled for audio.

Return Values

- error [NSError*]: Error object for error, or nil for success.

4.3.4.5 audioSessionDidActivate

Notifies the SDK of receiving CallKit's delegate callback of `audioSessionDidActivate`.

This method must be used to propagate CallKit's activation of the audio session to the SDK.

Parameters

- audioSession [AVAudioSession*]: The App's audio session, provided in the CallKit delegate callback parameter.

Return Values

- N/A

4.3.4.6 audioSessionDidDeActivate

Notifies the SDK of receiving CallKit's delegate callback of `audioSessionDidDeActivate`.

This method must be used to propagate CallKit's de-activation of the audio session to the SDK.

Parameters

- audioSession [AVAudioSession*]: The App's audio session, provided in the CallKit delegate callback parameter.

Return Values

- N/A

4.4 ACConfiguration

Used to provide additional configuration options for the WebRTC SDK. Using this class is optional. The class is a singleton object. The configuration object can be retrieved through `getConfiguration`. Any changes to this object are applied to the SDK. It is recommended to apply any changes before calling the AudioCodesUA login method.

```
@interface ACConfiguration : NSObject
+ (ACConfiguration*) getConfiguration;
@property (nonatomic, readonly) NSString *version;
@property (nonatomic, readwrite) int localServerPort;
@property (nonatomic, copy) DTMFOptions* dtmfOptions;
@property (nonatomic, copy) VideoConfiguration*
videoConfiguration;
@end
```

4.4.1 Standard Methods / Properties

4.4.1.1 getConfiguration

Defines the static method that returns the currently used configuration object.

Parameters

N/A

Return Values

- configuration [ACConfiguration]: Currently used configuration object; see Section 4.4.

4.4.1.2 NSString *version

Defines the static method that returns the current version of the SDK.

Parameters

N/A

Return Values

- Version [string]: Version of the SDK, e.g., 1.x

4.4.1.3 int localServerPort

Sets / Gets the current default local port used by the SIP stack. Default value is 6000.

Setter Parameter / Getter Return Value

- localServerPort [integer]: Default local user port (default 6000)

4.4.1.4 DTMFOptions* dtmfOptions

Gets / Changes the DTMFOptions class used by the SDK. This allows the sending of DTMF through either the WebRTC or SIP INFO. The class allows the changing of the DTMF duration and interval (if applicable for the chosen method). See Section 4.6 for more information.

Setter Parameter

- dtmfOptions: DTMFOptions class for setting the handling of DTMF tones.

Getter Return Value

- dtmfOptions copy [DTMFOptions]: DTMFOptions class for setting the handling of DTMF tones; the default value is for the WebRTC to handle DTMF tones.

The return value is a copy (internal property value) of the DTMFOptions object used by the ACConfiguration singleton. Therefore, a call like [ACConfiguration getConfiguration].dtmfOptions.dtmfMethod = SIP_INFO will not take effect.

For changes to take effect, a call must be made [ACConfiguration getConfiguration].dtmfOptions = someDtmfOptions, i.e., the property setter with a DTMFOptions object must be explicitly used.

4.4.1.5 VideoConfiguration* videoConfiguration

Gets / Changes the current video configuration used by the SDK. See also Section 4.5.

Setter Parameter

- videoConfiguration: Object containing video configuration options.

Getter Return Value

- videoConfiguration copy [VideoConfiguration]: Class containing video configuration options.

The return value is a copy (internal property value) of the `VideoConfiguration` object used by the ACConfiguration singleton. Therefore, a call like [ACConfiguration `getConfiguration`].videoConfiguration.cameraWidth = 480 will not take effect.

In order for changes to take effect, you must call [ACConfiguration getConfiguration]. videoConfiguration = someVideoConfiguration, i.e., the property setter with a VideoConfiguration object must be explicitly used.

4.5 Video Configuration

Provides additional configuration options for the WebRTC SDK. Using this class is optional. The class provides access to public parameters that can be changed if needed.

The configuration object can be retrieved through the `videoConfiguration` property in the ACConfiguration class. Calling the `videoConfiguration` setter in the ACConfiguration class will apply the changes. It is recommended to set `videoConfiguration` before `showVideo` is called.

```
@interface VideoConfiguration : NSObject <NSCopying>
@property (nonatomic, readwrite) NSInteger cameraWidth;
@property (nonatomic, readwrite) NSInteger cameraHeight;
@property (nonatomic, readwrite) NSInteger cameraFrameRate;
@end
```

4.5.1 Camera Parameters

- **cameraWidth** – Captures the width of the camera (default 640)
- **cameraHeight** - Captures the height of the camera (default 480)
- **cameraFrameRate** - Captures the frame rate of the camera (default 15)

4.6 DTMFOptions

Provides additional configuration options for the WebRTC SDK. Using this class is optional. The class provides access to public parameters that can be changed if needed. The class allows configuration of sending DTMF events.

```
@interface DTMFOptions : NSObject <NSCopying>
@property (nonatomic, readwrite) DTMFMethod dtmfMethod;
@property (nonatomic, readwrite) NSInteger duration;
@property (nonatomic, readwrite) NSInteger intervalGap;
@end
```

4.6.1 DTMF Parameters

- **dtmfMethod**: DTMFMethod enum parameter that supports sending of DTMF through:
  - **WEBRTC** - DTMF is sent through media by telephone-event using the WebRTC engine. This is the default method.
  - **SIP_INFO** - DTMF events are sent through SIP_INFO events.
- **duration**: Duration of the DTMF event (milliseconds). When using SIP_INFO, the minimum is 100 (default value).
- **intervalGap**: The interval gap in milliseconds between sending DTMF events. This is only relevant for WEBRTC DTMF events. Default: 70.

4.7 RemoteContact

Represents a remote contact. This contact can be either a dialed number or a remote contact received through an incoming call.

```
@interface RemoteContact: NSObject
@property (nonatomic, strong) NSString *displayName;
@property (nonatomic, strong) NSString *userName;
@property (nonatomic, strong) NSString *domain;
@end
```

4.7.1 Standard Methods / Properties

4.7.1.1 NSString *displayName

Sets / Gets the optional contact display name. Since this does not affect SIP signaling, it's optional; allows for easy retrieval of the display name used in the call.

Setter Parameters / Getter Return Values

- displayName [NSstring]: Display name of the remote contact

4.7.1.2 NSString *userName

Sets / Gets the contact user name.

Setter Parameter / Getter Return Values

- userName [string]: User name of the remote contact

4.7.1.3 NSString *domain

Sets / Gets the contact domain.

Setter Parameter / Getter Return Values

- domain [string]: Defines the domain of the remote contact. This value can remain unset, which defaults to the same domain defined as the `serverDomain` parameter of the `setServerConfig` method.

4.8 ACAlertInfoAttributes

Represents the data contained in the Alert-Info header of an incoming INVITE request for an incoming call.

```
@protocol ACInfoAlertAttributes <NSObject>
@property (nonatomic, readonly) BOOL autoAnswer;
@property (nonatomic, readonly) NSInteger delay;
@end
```

4.8.1 Standard Methods / Properties

4.8.1.1 BOOL autoAnswer

Getter property for indicating whether the call should be answered automatically after the *delay* property value.

Getter Return Values

- True: The call should be answered automatically after the amount of seconds in the *delay* property value
- False: The call should not be answered automatically

4.8.1.2 NSInteger delay

Setter Parameter / Getter Return Values

- delay [Integer]: The amount of time, in seconds, that needs to pass before the call is answered automatically.

4.9 ACNetworkConnectionAttributes

Represents optional network connection attributes for the user-agent.

```
@interface ACNetworkConnectionAttributes: NSObject
@property (nonatomic, assign) ACNetworkAddressFamily
localAddressFamily;
+ (instancetype) attrWithLocalAddressFamily:
(ACNetworkAddressFamily)addressFamily;
@end
```

4.9.1 Standard Methods / Properties

4.9.1.1 ACNetworkAddressFamily localAddressFamily

Gets / Sets the local IP-address family type, as determined by the application, to be the primary address family for connections.

Discussion:

It is the application's responsibility to determine which IP address type is the primary one to consider.

When binding the UA to a connection using the `setConnectionBinding` method, determining the primary address type is valuable for the following:

- When making calls without registrations, connection binding can only occur if `setConnectionBinding` was called with initialPrefs.localAddressFamily value to be `ACNetworkAddressFamilyIPV4` or `ACNetworkAddressFamilyIPV6`.
- On network changes with SIP connection binding applied, maintaining calls and registrations can only occur if `handleNetworkChange` is called with attributes.localAddressFamily to be `ACNetworkAddressFamilyIPV4` or `ACNetworkAddressFamilyIPV6`.

Possible Values of the ACNetworkAddressFamily type:

- ACNetworkAddressFamilyUnspecified: Represents any IP-address family
- ACNetworkAddressFamilyIPV4: Represents IPV4 address family
- ACNetworkAddressFamilyIPV6: Represents IPV6 address family

4.10 TerminationInfo

Represents a remote contact. This contact can either be a dialed number or a remote contact received through an incoming call.

```
@interface TerminationInfo: NSObject
@property (nonatomic) CallTermination termination;
@property (nonatomic) NSInteger sipStatusCode;
@property (nonatomic, strong) NSString *sipStatusText;
@property (nonatomic, strong) NSString *sipReasonHeaderValue;
@property (nonatomic, strong) NSString *sipMessage;
@end
```

4.10.1 Properties

4.10.1.1 CallTermination termination

Returns an enumeration value representing a general call termination reason.

4.10.1.2 NSInteger sipStatusCode

Returns the SIP response code that the call was terminated with. If the call was not terminated with a SIP-related status, it returns 0.

4.10.1.3 NSString *sipStatusText

Returns the string of the SIP response text corresponding to the SIP response code. If not applicable, returns *nil*.

4.10.1.4 NSString *sipReasonHeaderValue

Returns the value of the SIP "Reason" header, if exists in the SIP message that has caused the termination. If the call wasn't terminated by a SIP message with a "Reason" header, the returned value is *nil*.

4.10.1.5 NSString *sipMessage

Returns the contents of the last SIP message, if available, of the SIP transaction that has terminated the call. If not available, or if the call was not terminated with a SIP transaction, *nil* is returned.

4.11 ACNativeCallService

Used for high-level interaction with the native telephony system of the device's operating system. Currently this applies to the following:

- **iOS** CallKit**:** Manages the usage of the CallKit framework for call management.
- **iOS Native Audio Management:** Interacts with the system audio management, with or without CallKit, and handles the entire audio management aspect with regards to VOIP calls and the App runtime. This also includes managing audio categories, modes and routing with respect to the different flows of call management.

```
@interface ACNativeCallService : NSObject
@property (nonatomic, readonly) BOOL callGroupSupported;
@property (nonatomic, readonly) BOOL usingCallKit;
+ (instancetype) sharedInstance;
- (void)
initiateWithConfiguration:(CXProviderConfiguration*)config;
- (void) invalidate;
- (void) reportNewIncomingCall:(AudioCodesSession*)call
           localizedCallerName:(NSString*)callerName
                answerCallback:(ActionExecutionBlock
_Nullable)answerCallback
                rejectCallback:(ActionExecutionBlock
_Nullable)rejectCallback
                        result:(ACCallKitTaskSetupCompletion
_Nullable)completion;
- (void) reportCallTerminated:(AudioCodesSession*)call
terminationStatusCode:(int)statusCode;
```

```
- (void) reportCallUpdated:(AudioCodesSession*)call;
- (void) reportCallStartedConnecting:(AudioCodesSession*)call;
- (void) reportCallEstablished:(AudioCodesSession*)call;
- (void) initiateStartCall:(AudioCodesSession*)call
                callerName:(NSString*)callerName
            actionCallback:(ActionExecutionBlock
_Nullable)actionCallback
                    result:(ACCallKitTaskSetupCompletion
_Nullable)completion;
- (void) initiateAnswerCall:(AudioCodesSession *)call
             actionCallback:(ActionExecutionBlock
_Nullable)actionCallback
                     result:(ACCallKitTaskSetupCompletion
_Nullable)completion;
- (void) initiateEndCall:(AudioCodesSession *)call
          actionCallback:(ActionExecutionBlock
_Nullable)actionCallback
                  result:(ACCallKitTaskSetupCompletion
_Nullable)completion;
- (void) initiateHoldCall:(AudioCodesSession *)call
                   onHold:(BOOL)onHold
           actionCallback:(ActionExecutionBlock
_Nullable)actionCallback
                   result:(ACCallKitTaskSetupCompletion
_Nullable)completion;
- (void) initiateMuteCall:(AudioCodesSession *)call
                    muted:(BOOL)muted
           actionCallback:(ActionExecutionBlock
_Nullable)actionCallback
                   result:(ACCallKitTaskSetupCompletion
_Nullable)completion;
- (void) initiateSendDtmfCall:(AudioCodesSession *)call
                        digit:(UInt8)digit
                actionCallback:(ActionExecutionBlock
_Nullable)actionCallback
                       result:(ACCallKitTaskSetupCompletion
_Nullable)completion;
@end
```

4.11.1 Class Type Definitions

4.11.1.1 typedef NS_ENUM (NSInteger, ACCallKitExecutionBlockResult)

Defines the possible return values of a callback block, that is executed by a call action.

- **AC**CallKit**ExecutionBlockResultUndefined** – Execution block result has no effect on the corresponding call action. The action handler will proceed with its default behavior.
- **AC**CallKit**ExecutionBlockResultFulfill** – Execution block determines that the call action is fulfilled. This return value of an action callback prevents the action handler from proceeding with its default behavior.
- **AC**CallKit**ExecutionBlockResultFail** – Execution block determines that the call action has failed. This return value of an action callback prevents the action handler from proceeding with its default behavior.

4.11.1.2 typedef ACCallKitExecutionBlockResult (^ActionExecutionBlock)(void);

Defines the code that is executed when the call action handler operates. The return values that are different than `ACCallKitExecutionBlockResultUndefined`, informs the handler that the `ActionExecutionBlock` completely replaces its default behavior. This way, one can determine exactly how to use the AudioCodesSession object that corresponds to the call action.

4.11.1.3 typedef void (^ACCallKitTaskSetupCompletion)(NSArray * _Nullable actionUUIDs, NSError * _Nullable error)

Defines the completion block of a call action setup. When it executes, the setup of the call action is complete and the call action is underway to be performed in conjunction with CallKit.

If CallKit is not in use, this code executes immediately:

- **ActionUUIDs** – Defines an array of call action identifiers, specific to CallKit. Currently it is only used to optionally maintain an identification to the corresponding call action.
- **Error** – Defines the NSError object to set up the corresponding call action, in case of failure.

4.11.2 Standard Methods / Properties

4.11.2.1 sharedInstance

Returns the singleton instance of the `ACNativeCallService`. Calling it for the first time also starts monitoring the system's delivery of audio interruption notifications.

> **Note:** When using CallKit and audio interruption begins (such as for incoming GSM calls, activating Siri, Alarm Clock alerts firing), the SDK automatically holds any active calls.
>
> When audio interruption ends, the SDK automatically resumes the calls held by the interruption.
>
> Developers can be notified of these actions via the delivery of `ACAudioInterruptNotification`. See Section 5.4.2.
>
> Without CallKit, it is the developer's responsibility to hold / unhold the calls.

4.11.2.2 BOOL usingCallKit

Defines the getter property that indicates whether CallKit is being used.

Getter Return Values

- **TRUE:** The method `initiateWithConfigutation:` was called with a valid `CXProviderConfiguration` object, and *invalidate* was not called.
- **FALSE:** The method `initiateWithConfiguration:` was not called, or *invalidate* was called.

> The `ACNativeCallService` class can operate fully without CallKit. In that case, the `usingCallKit` value is 'FALSE', and the `ACNativeCallService` object operates as a higher layer that uses the SDK regularly for call management. So every method call to `ACNativeCallService` without CallKit, has the underlying desired effect of calling the corresponding SDK functionality.

4.11.2.3 BOOL callGroupSupported

Indicates whether the SDK supports call grouping (conference) in conjunction with the native call system. It's value is always FALSE.

4.11.2.4 initiateWithConfiguration:(CXProviderConfiguration*)config

Initiates CallKit integration with the CallKit configuration given as a `CXProviderConfiguration` object.

The `CXProviderConfiguration` has the following methods that are supported by the SDK:

- ***initWithLocalizedName:(NSString*)localizedName*** – Init with the application name, given for the CallKit system to display with calls performed / received via the SDK.
- ***NSString *ringToneSound*** – Determines a sound file in the App bundle, to be played by CallKit on incoming call
- ***NSData *iconTemplateImageData*** – Defines image data of a template image to display by CallKit, in the native call screen UI, as the App icon.
- ***NSUInteger maximumCallGroups*** – Defines the maximum available calls. The SDK supports up to 4 calls simultaneously.
- ***BOOL includeCallsInRecent*** – Defines whether or not to include the app's calls in the native Phone's Recent database
- ***BOOL supportsVideo*** – Defines whether video should be supported. The SDK sully supports video.
- ***NSSet <NSNumber*>* supportedHandleTypes*** – Defines the types of calling destinations to support, corresponding to `CXHandleType` values. The SDK supports `CXHandleTypePhoneNumber`.

For more details on configuring the `CXProviderConfiguration` object, please refer to the CXProviderConfiguration documentation.

> This method is optional, and the `ACNativeCallService` class can be used without CallKit at all. Not calling the `initiateWithConfiguration`: tells the `ACNativeCallService` class to operate regularly as a higher-level layer on top of AudiocodesUA / AudioCodesSession. This way, applications can use this class in a way that enabling or disabling usage of CallKit is transparent to the SDK.

4.11.2.5 invalidate

Stops the CallKit provider, disables CallKit usage, and terminates every call currently ongoing, that has been initiated or received with CallKit.

4.11.2.6 reportNewIncomingCall

Notifies the call service of a new incoming call. If CallKit is used, then it will display the native incoming call UI.

Parameters

- call [AudioCodesSession]: The AudioCodesSession object that represents the corresponding call.
- callerName [NSString]: The display name of the remote party. Applications can provide a custom display name that is different from the AudioCodesSession RemoteNumber properties.

- answerCallback [ActionExecutionBlock]: Code to be executed when pressing the native call UI answer button. The return value of the callback determines how the answer handler should proceed. See `ACCallKitExecutionBlockResult`.
- rejectCallback [ActionExecutionBlock]: Code to be executed when pressing the native call UI reject button. The return value of the callback determines how the reject handler should proceed. See `ACCallKitExecutionBlockResult`.
- completion [ACCallKitTaskSetupCompletion] The completion block.

Return Values

N/A

4.11.2.7 reportCallTerminated

Notifies the call service that a call was terminated with a given status code.

This method should be called for every termination cause, both local or remote. Using this method is important for properly handle outgoing call connection, including audio routing and management.

Parameters

- call [AudioCodesSession]: The corresponding call object.
- statusCode [int]: Value corresponding to `CallTermination` type.

Return Values

N/A

4.11.2.8 reportCallUpdated

Triggers the call service for updating the call properties.

This method must be called at least once in order for the system to register the basic call functionality, such as whether hold / DTMF is supported.

Parameters

- call [AudioCodesSession]: The corresponding call object

Return Values

N/A

4.11.2.9 reportCallStartedConnecting

Updates the system from when the call has started connecting. Applies only for outgoing call.

Parameters

- call [AudioCodesSession]: The corresponding call object

Return Values

N/A

4.11.2.10 reportCallEstablished

Updates the system for when the call is established. Applies only for outgoing call.

Using this method is important for properly handle outgoing call connection, including audio routing and management.

Parameters

- call [AudioCodesSession]: The corresponding call object

Return Values

N/A

4.11.2.11 initiateStartCall

Registers a start call operation with the call service.

the AudioCodesSession "call" parameter is provided here, meaning that one should call this method right after the AudioCodesSession object created, i.e when the SIP INVITE message is being sent.

Parameters

- call [AudioCodesSession]: The corresponding call object
- callerName [NSString]: The display name of the remote party. Applications can provide a custom display name that is different from the AudioCodesSession RemoteNumber properties.
- actionCallback [ActionExecutionBlock]: Code to execute by the action handler when the call actually starts. This can be used to update UI or perform other related tasks.
- Completion [ACCallKitTaskSetupCompletion]: Completion block

Return Values

N/A

4.11.2.12 initiateAnswerCall

Registers answering a call with the call service. This applies to cases when the user answers the call from the application, and not from the native call UI.

Parameters

- call [AudioCodesSession]: The corresponding call object
- actionCallback [ActionExecutionBlock]: Code to execute by the action handler when the call is being answered. This can be used to update UI or perform other related tasks.
- Completion [ACCallKitTaskSetupCompletion]: Completion block of setting up the action.

Return Values

N/A

4.11.2.13 initiateEndCall

Registers ending a call with the call service. This applies to cases when the user ends the call from the application, and not from the native call UI.

Parameters

- call [AudioCodesSession]: The corresponding call object
- actionCallback [ActionExecutionBlock]: Code to execute by the action handler when the call is being terminated. This can be used to update UI or perform other related tasks. One can determine in this callback whether to terminate or reject.

> If this is an incoming call, which has been reported with a reject callback, and the call has not been established, then the reject callback will be invoked.

- Completion [ACCallKitTaskSetupCompletion]: Completion block of setting up the action.

Return Values

N/A

4.11.2.14 initiateHoldCall

Registers holding / resuming a call with the call service.

Parameters

- call [AudioCodesSession]: The corresponding call object
- onHold [BOOL]: The desired hold state.
- actionCallback [ActionExecutionBlock]: Code to execute by the action handler when the call is being held / resumed.
- Completion [ACCallKitTaskSetupCompletion]: Completion block of setting up the action.

Return Values

N/A

4.11.2.15 initiateMuteCall

Registers muting / unmuting audio in a call with the call service.

Parameters

- call [AudioCodesSession]: The corresponding call object
- muted [BOOL]: The desired mute state.
- actionCallback [ActionExecutionBlock]: Code to execute by the action handler when the call is being muted / unmuted.
- Completion [ACCallKitTaskSetupCompletion]: Completion block of setting up the action.

Return Values

N/A

4.11.2.16 initiateSendDTMFCall

Registers sending DTMF in a call with the call service.

Parameters

- call [AudioCodesSession]: The corresponding call object.
- digit [UInt8]: DTMF digit value corresponding to the SDK **DTMF** type.
- actionCallback [ActionExecutionBlock]: Code to execute by the action handler when DTMF is about to be sent.
- Completion [ACCallKitTaskSetupCompletion]: Completion block of setting up the action.

Return Values

N/A

4.11.2.17 isCallAssociatedWithNative

Determines whether the call was initiated using the `ACNativeCallService`.

Parameters

- call [AudioCodesSession]: The corresponding call object

Return Values

- **TRUE:** The `using`CallKit property returns True, and the method `reportNewIncomingCall` or `initiateStartCall` was called with the call object as a parameter.
- **FALSE:** Otherwise

5 API Callbacks / Delegate Protocols / Notifications

The API provides the capability to register to listen to different types of events and implement available callback functionalities. The following describes these supported features.

5.1 AudioCodesEventListener

Defines the interface for receiving SDK events. This interface must be implemented and set through the AudioCodesUA class to receive these events.

5.1.1 Login State Changed Event

Triggered when the login state has been changed.

Syntax

```
- (void) loginStateChanged:(BOOL)isLogin cause:(NSString*)cause;
```

Parameters

- IsLogin [boolean]: 'True' if logged in and 'False' if not logged in.
- cause [string]: Text describing the received SIP reason. This string can be predominantly used if more information on a login failure is required.
- This string can be one of the following:

```
extern NSString* const ACUALoginChangedReasonConnected;
extern NSString* const ACUALoginChangedReasonDisconnected;
extern NSString* const ACUALoginChangedReasonConnectionFailed;
```

5.1.2 Incoming Call Event

Triggered when receiving an incoming call.

Syntax

```
- (void) incomingCall:(AudioCodesSession*)call;
- (void) incomingCall:(AudioCodesSession*)call
infoAlert(id<ACInfoAlertAttributes>)infoAlert;
```

Parameters

- call [AudioCodesSession]: The incoming call session object
- infoAlert [ACInfoAlertAttributes]: Optional overloaded parameter, containing data from the “Alert-Info” header, if it exists in the incoming INVITE request. If no relevant data exists, this parameter value is nil.

5.1.3 Incoming Instant Message Event

Triggered when receiving an incoming SIP instant message.

Syntax

```
- (void) incomingInstantMessage:(NSString*)message
from:(RemoteContact*)remoteContact;
```

Parameters

- message [string]: The incoming message text
  - remoteContact [RemoteContact]: The message sender

5.1.4 Outgoing Instant Message Status Update

Triggered when there is an update on the status of a sent SIP instant message request.

Syntax

```
- (void) instantMessageStatus:(InstanceMessageStatus)status
messageId:(NSString*)messageId
```

Parameters

- status [InstanceMessageStatus]: The status of the outgoing message request. Possible values:
  - IM_UNDEFINED = -1: Undefined
  - SUCCESS: Received success response (SIP 200-OK)
  - ACCEPTED: Received accepted response (SIP 202)
  - NOT_FOUND: Received not-found error (SIP 404)
  - UNKNOWN_ERROR: Received other unknown error
- messageId [string]: The message identifier which corresponds to the one returned by the `sendInstantMessage` method call.

5.2 AudioCodesSessionEventListener

5.2.1 callTerminated

Callback when the session is terminated by the local or the remote side. Use the `terminationInfo` getter property for call termination data.

Syntax

```
- (void) callTerminated:(AudioCodesSession*)call;
```

Parameters

call [AudioCodesSession]: The call session object that was terminated. The object is removed by the `callTerminated` method.

5.2.2 callProgress

Callback for changes in the state of the call. The call progress state can be retrieved by `callState` property of the AudioCodesSession object.

Syntax

```
- (void) callProgress:(AudioCodesSession*)call;
```

Parameters

call [AudioCodesSession]: The call session object

5.2.3 callNotifyEvent

Callback for incoming SIP-Notify requests that are associated with the call, and can represent a certain remote-control event that the client application is required to perform.

Syntax

```
- (void) callNotifyEvent:(AudioCodesSession*)call
type:(CallNotifyEventType)type dtmfString:(NSString*)dtmf;
```

Parameters

- call [AudioCodesSession]: The call session object
- dtmf [String]: Optional parameter. For event type 'dtmf', the parameter value is the DTMF string. For any other event type, the value is nil.
- type [CallNotifyEventType]: The event type that corresponds to the incoming Notify request. It can be one of the following:
  - **ACCallNotifyEventUndefined** – a generic NOTIFY request was received
  - **ACCallNotifyEventTalk** – If the call is incoming and is not answered yet, then the client application is required to answer the call. If the call is already active and on hold, then the client application is required to un-hold it.
  - **ACCallNotifyEventHold** – If the call is active, then the client application is required to put the call on hold
  - **ACCallNotifyEventDTMF** – The client application is required to send DTMF characters provided in the *dtmf* string parameter. This should be performed using calls to the sendDTMF method consecutively for each character in the DTMF string, and in a way that is non-blocking to the current thread. The interval between sending each DTMF character should be MAX(DTMFOptions.intervalGap, DTMFOptions.duration).
  - ACCallNotifyEventConference – Currently not supported

5.2.4 cameraSwitched

Callback for when the camera has been switched between the front or the back camera.

Syntax

```
- (void) cameraSwitched:(BOOL)frontCamera;
```

Parameters

- frontCamera [boolean]: 'True' : the camera has switched to the front camera
- 'False': the camera has switched to the back camera.

5.2.5 incomingInfo

Callback for when a SIP INFO message arrives.

Syntax

```
- (void) incomingInfo:(id<ACInfoMessage>)infoMessage;
```

Parameters

- infoMessage [ACInfoMessage] The INFO message structure, containing these properties:
  - contentType [string]: The INFO message body MIME type
  - infoBody [string]: The INFO message body string

5.3 WebRTCAudioRoutesListener

Defines the interface for receiving audio routes events. The interface must be implemented and set through the `WebRTCAudioManager` class to receive these events.

5.3.1 audioRoutesChanged

Callback for when the list of available audio routes has been changed, for example, if the user is connected to a Bluetooth audio device.

Syntax

```
- (void)
audioRoutesChanged:(NSArray<AudioRouteNumber*>*)audioRouteList;
```

Parameters

- audioRouteList [`NSArray<AudioRouteNumber*>*`: List of available audio routes

5.3.2 currentAudioRouteChanged

Defines the callback for when the currently used audio route has been changed. e.g., if the user adds a Bluetooth audio device, the SDK routes the audio to the Bluetooth device and this callback is called.

Syntax

```
- (void) currentAudioRouteChanged:(AudioRoute)newAudioRoute;
```

Parameters

- newAudioRoute [AudioRoute]: New audio route through which the audio is routed.

5.4 NSNotifications

5.4.1 AudioCodesSession Notifications

Observable notifications for AudioCodesSession events, equivalent to its delegate methods. The Notification object is the relevant AudioCodesSession instance.

```
extern NSString* const ACSessionCallProgressNotification;
extern NSString* const ACSessionCallTerminatedNotification;
extern NSString* const ACSessionCallNotifyEventNotification;
```

The ACSessionCallNotifyEventNotification includes user info keys describing the CallNotifyEventType and DTMF values associated with the call notify event:

```
extern NSString* const ACSessionNotifyEventTypeUserInfoKey;
extern NSString* const ACSessionNotifyDTMFUserInfoKey;
extern NSString* const ACSessionCameraSwitchedNotification;
```

The ACSessionCameraSwitchedNotification notification includes a user-info key describing whether it's front or back camera. The value is a boolean wrapped in NSNumber object, whose value is 'True' for front camera and 'False' for back camera.

```
extern NSString* const
ACSessionCameraSwitchedFrontCameraUserInfoKey;
```

5.4.2 WebRTCAudioManager Notifications

Observable notifications for `WebRTCAudioManager` events, equivalent to its delegate methods. The Notification objects include user-info key-values described below.

```
extern NSString * const ACAudioRouteChangedNotification;
extern NSString * const ACAudioRouteRouteChangedNotificationCurrentRouteKey;
extern NSString * const ACAudioRouteRouteAvailabilityChangedNotification;
extern NSString * const ACAudioRouteRouteAvailabilityChangedReceiverAvailableKey;
extern NSString * const ACAudioRouteRouteAvailabilityChangedSpeakerAvailableKey;
extern NSString * const ACAudioRouteRouteAvailabilityChangedBluetoothAvailableKey;
extern NSString * const ACAudioInterruptNotification;
extern NSString * const ACAudioSessionUserInfoKey;
extern NSString * const ACAudioIsInterruptedUserInfoKey;
extern NSString * const ACAudioShouldResumeUserInfoKey;
extern NSString * const ACAudioWasSuspendedUserInfoKey; // Available only since iOS 10.3
```

- **ACAudioRouteChangedNotification**: Invoked when the current audio route is changed. Includes the user-info key. **ACAudioRouteRouteChangedNotificationCurrentRouteKey**: whose value is an NSNumber wrapping the corresponding `AudioRoute` enum value.
- **ACAudioRouteRouteAvailabilityChangedNotification**: Invoked when the list of available audio routes has been changed, for example, if the user is connected to a Bluetooth audio device. Includes user-info keys to describe whether Receiver / Speaker / Bluetooth routes are available:
  - Key: **ACAudioRouteRouteAvailabilityChangedReceiverAvailableKey**
    Value: NSNumber-wrapped boolean, 'True' for route available, 'False' for unavailable.
  - Key: **ACAudioRouteRouteAvailabilityChangedSpeakerAvailableKey**
    Value: NSNumber-wrapped boolean, 'True' for route available, 'False' for unavailable.
  - Key: **ACAudioRouteRouteAvailabilityChangedBluetoothAvailableKey**
    Value: NSNumber-wrapped boolean, 'True' for route available, 'False' for unavailable.
- **ACAudioInterruptNotification**: Invoked when the system delivers an audio interruption. This is relevant when CallKit is not used.
  - Key: **ACAudioSessionUserInfoKey**
    Value: AVAudioSession, the app's AVAudioSession object.

- Key: **ACAudioIsInterruptedUserInfoKey**
  Value: NSNumber-wrapped boolean, 'True' for audio being interrupted, 'False' for audio un-interrupted.
- Key: **ACAudioShouldResumeUserInfoKey**
  Value: NSNumber-wrapped boolean, 'True' for whether audio is allowed to resume in the app, 'False' for audio should not resume. Only applicable for audio un-interrupted notification.
- Key: **ACAudioWasSuspendedUserInfoKey**
  Value: NSNumber-wrapped boolean, 'True' whether the interruption notification was delivered as a result of app suspension, 'False' otherwise.

When using CallKit with the `MVWebRTCNativeCall` framework, the `ACNativeCallService` automatically holds calls when the audio is interrupted, and un-holds these calls when audio interruption ends. This requires the `ACNativeCallService` instance to be initialized with the first call to [ACNativeCallService sharedInstance].

6 Use Case Examples

This chapter includes use case examples for reference.

6.1 User Agent: Create Instance, Set server and Account

6.1.1 Swift API

```
import ACWebRTCSdk

  let phone = UserAgent.shared

  phone.configureServer(
      ServerConfiguration(
          proxyAddress: "webrtclab.audiocodes.com",
          port: 5080,
          serverDomain: "example.com",
          transport: .tcp,
          iceServers: []
      )
  )

  phone.configureAccount(
      AccountConfiguration(
          userName: "John",
          displayName: "John Smith",
          authentication: .digest(
              userName: "jsmit",
              password: "*****"
          )
      )
  )
```

6.1.2 Objective-C API

```
AudioCodesUA *phone = [AudioCodesUA getInstance];
[phone setServerConfig:@"webrtclab.audiocodes.com"
                  port:5080
          serverDomain:@"example.com"
             transport:ACTransportTCP
            iceServers:nil];
[phone setAccount:@"John"
      displayName:@"John Smith"
         password:@"*****"
         authName:@"jsmit"];
```

6.2 User Agent: Set Listeners (Callbacks)

6.2.1 Swift API

```
phone.onLoginStateChanged = { change in
    if change.isLoggedIn {
        // Code to handle successful login.
    } else {
        // Code to handle logout or login failure.
        let cause = change.cause
    }
}

phone.onIncomingCall = { incomingCall in
    let call = incomingCall.session

    // Code to handle the incoming call event.
}
```

6.2.2 Objective-C API

```
phone.delegate = self;
- (void) loginStateChanged:(BOOL)isLogin cause:(NSString*)cause {
    // Code to handles login-related events
}
- (void) incomingCall:(AudioCodesSession*)call {
    // Code to handles incoming call event
}
```

6.3 User Agent Login: Connection to SBC Server and Login

6.3.1 Swift API

```
// This will start connecting to SBC and will trigger
// the loginStateChanged delegate method
phone.login()
```

6.3.2 Objective-C API

```
// This will start connecting to SBC and will trigger
// the loginStateChanged delegate method
[phone login];
```

6.4 Make a Call, Set Call Delegate

6.4.1 Swift API

```
let remoteContact = RemoteContact(userName: "Jane")

let call = phone.startCall(to: remoteContact)

call.callbackQueue = .main

call.onProgress = { call in
    // Code to handle call state changes.
}

call.onTerminated = { call in
    // Code to handle call termination.
}
```

An example of video call creation:

```
let remoteContact = RemoteContact(userName: "Jane")
let call = phone.startCall(
   to: remoteContact,
   options: .init(media: .video)
   )

call.callbackQueue = .main
call.onProgress = { call in
   // Code to handle call state changes.
   }
call.onTerminated = { call in
   // Code to handle call termination.
   }
call.onCameraSwitched = { position in
   // position is .front or .back
   }
```

6.4.2 Objective-C API

```
BOOL useVideo = YES;
RemoteContact *remoteContact = [[RemoteContact alloc] init];
remoteContact.userName = @”Jane”;
AudioCodesSession *call = [phone call:remoteContact
withVideo:useVideo inviteHeaders:nil];
call.delegate = self;
- (void) callProgress:(AudioCodesSession*)call {
    // Code to handle call state changes
}
- (void) callTerminated:(AudioCodesSession*)call {
    // Code to handle call termination
}
- (void) cameraSwitched:(BOOL)frontCamera {
    // Code to handle camera switch
}
```

6.5 Send DTMF During Call

6.5.1 Swift API

```
activeCall.sendDTMF(.nine)
```

6.5.2 Objective-C API

```
[self.activeCall sendDtmf:DTMF_9];
```

6.6 Mute / Unmute During Call

6.6.1 Swift API

The Swift API provides CallSession.isAudioMuted for audio muting:

```
self.activeCall?.isAudioMuted = true
```

and methods for video muting/unmuting:

```
self.activeCall?.muteVideo()
self.activeCall?.unmuteVideo()
```

6.6.2 Objective-C API

```
self.activeCall.muteAudio = YES;
self.activeCall.muteVideo = NO;
```

6.7 Accept Incoming Call (with Video)

6.7.1 Swift API

```
// Add VideoStreamView(session: call, stream: .remote)
// and VideoStreamView(session: call, stream: .localPreview) in
the UI.

self.activeCall?.enableVideo { _ in
     self.activeCall?.answer()
     }
```

6.7.2 Objective-C API

```
// To answer with video we first need to add local and / or remote
UIViews
// To the call for video rendering, and upon completion, answer
the call
[self.activeCall showVideoLocalView:localView
remoteView:remoteView completion:^{
    [self.activeCall answer:nil];
}];
```

6.8 Delayed-offer: Treat incoming calls as video calls

6.8.1 Swift API

```
// Incoming delayed-offer calls can optionally be treated as incoming
video calls.
// This is useful when the INVITE has no SDP, so the app cannot determine
the media type up front.
// Call enableVideo() even if no VideoStreamView instances are attached
yet.
phone.onIncomingCall = { incomingCall in
    let call = incomingCall.session

    if call.isDelayedOffer && treatDelayedOfferAsIncomingVideoCall {
        call.enableVideo { _ in
            // code to handle incoming video call
        }
    } else {
        // code to handle regular incoming call
    }
}
```

6.8.2 Objective-C API

```
// Incoming delayed-offer calls can optionally be treated as incoming
video calls. We might need this because there is no SDP to allow us to
determine whether this is a video call or not.
// In order to perform this, we can call showVideo even with no
renderers.
func incomingCall(_ call: AudioCodesSession!, infoAlert:
ACInfoAlertAttributes!)  {
    if call.isDelayedOffer && treatDelayedOfferAsIncomingVideoCall {
        call.showVideoLocalView(nil, remoteView: nil) {
            // code to handle incoming video call
        }
    } else {
        // code to handle regular incoming call
    }
}
```

6.9 Reject Incoming Call

6.9.1 Swift API

```
incomingCall.reject()
```

6.9.2 Objective-C API

```
[incomingCall reject:nil];
```

6.10 Terminate a Call

6.10.1 Swift API

```
activeCall.terminate()
```

6.10.2 Objective-C API

```
[activeCall terminate];
```

6.11 Use of Video

6.11.1 Swift API

To use video, the following conditions must apply:

- To capture and send video from the camera, the application UI must include a `VideoStreamView` that renders the local preview, for example:

  ```
  VideoStreamView(session: call, stream: .localPreview).
  ```

- To display video from the remote party, the application UI must include a `VideoStreamView` that renders the remote stream, for example:

  ```
  VideoStreamView(session: call, stream: .remote).
  ```

- To use video during a call, attach the required `VideoStreamView` instances and call `enableVideo`(completion:). The local and remote views can be attached independently. However, for privacy reasons, local video is captured and transmitted **only** when a local preview view is attached.

```
// Start an outgoing video call by requesting video media.
private func startOutgoingCall() {
    let remote = RemoteContact(userName: "1000", domain:
"audiocodes.com")
    let call = UserAgent.shared.startCall(
        to: remote,
        options: .init(media: .video)
    )
    // Enable local camera capture/sending.
    call.enableVideo()
}

// Answer an incoming call with video.
func answerIncomingCall(call: CallSession) {
    if call.hasVideo {
        call.enableVideo { error in
            guard error == nil else { return }
            call.answer()
        }
    } else {
        call.answer()
    }
}
```

```
// Render remote video and local camera preview.
VideoStreamView(session: call, stream: .remote)
VideoStreamView(session: call, stream: .localPreview)

// Basic video controls.
call.switchCamera()
call.muteVideo()
call.unmuteVideo()

// End the call.
call.terminate()
```

6.11.2 Objective-C API

To use video, the following conditions must apply:

- To capture and send video from the camera, the application GUI should include UIView for rendering local video.
- To display video from the remote side, the application GUI should include a UIView for rendering remote video.
- To use video during the call, use the showVideoLocalView:remoteVide:completion method, and pass the views as parameters. These can be passed as nil values; however, note that **local video will NOT be captured and sent unless there is a UIView to render it**, for privacy reasons.
- The showVideoLocalView:remoteVide:completion method can be called at any time / state of the call, and it will internally perform the appropriate tasks. Notable cases:
  - For incoming calls before answering: video rendering and camera capture is started, and video media is added to the answer signal SDP.
  - For active calls without video: video capture is started and rendered, and also media re-negotiation (re-INVITE) is performed with renewed video SDP.
  - Whenever the local renderer is nil (e.g., when discarding the call GUI, but keeping the call active), the sent video is an RTP stream of blank frames.

6.12 Using Built-In CallKit Support – ACNativeCallService

6.12.1 Swift API

`ACNativeCallService` is not part of the Swift API but it remains available through `import MVWebRTCNativeCall`. Application that use Swift API can access the underlying AudioCodesSession instance via CallSession.audioCodesSession for interoperability with `ACNativeCallService`.

```
let nativeCallService = ACNativeCallService.sharedInstance()

// Configure and activate native call presentation.
private func startNativeCallService() {
    let configuration = CXProviderConfiguration()
    configuration.maximumCallsPerCallGroup = 1
    configuration.supportsVideo = false
    configuration.supportedHandleTypes = [.phoneNumber]
    configuration.ringtoneSound = "Ringtone.aif"
```

```
    configuration.iconTemplateImageData = UIImage(systemName:
"phone.fill")?.pngData()

    nativeCallService.initiate(with: configuration)
}

// Stop native call handling when the demo/screen is no longer
active.
private func stopNativeCallService() {
    nativeCallService.invalidate()
}

// Report a SIP INVITE as a native incoming call.
private func reportIncomingCall(_ call: CallSession) {
    guard let legacyCall = call.audioCodesSession as?
AudioCodesSession else { return }

    let callerName = call.remoteContact?.displayName
        ?? call.remoteContact?.userName
        ?? "Unknown"

    nativeCallService.reportNewIncomingCall(
        legacyCall,
        localizedCallerName: callerName
    ) {
        call.answer()
        return .fulfill
    } rejectCallback: {
        call.reject()
        return .fulfill
    } result: { _, error in
        guard error == nil else { return }
    }
}

// Start an outgoing SIP call and register it with CallKit
immediately.
private func startOutgoingNativeCall() {
    let remote = RemoteContact(userName: "1000", domain:
"audiocodes.com")
    let call = UserAgent.shared.startCall(to: remote)

    guard let legacyCall = call.audioCodesSession as?
AudioCodesSession else { return }

    nativeCallService.initiateStartCall(
        legacyCall,
        callerName: remote.displayName ?? remote.userName ??
"Unknown",
        actionCallback: nil,
        result: nil
    )
```

```
}

// Keep CallKit synchronized with outgoing call progress.
private func reportOutgoingProgress(_ call: CallSession) {
    guard let legacyCall = call.audioCodesSession as?
AudioCodesSession else { return }

    if call.callState == .calling {
        nativeCallService.reportCallStartedConnecting(legacyCall)
    }

    if call.callState == .connected {
        nativeCallService.reportCallEstablished(legacyCall)
    }
}

// Report SIP termination to the native call service.
private func reportTerminatedCall(_ call: CallSession) {
    guard let legacyCall = call.audioCodesSession as?
AudioCodesSession,
          let terminationInfo = call.terminationInfo
    else {
        return
    }

    nativeCallService.reportCallTerminated(
        legacyCall,
        terminationStatusCode:
Int32(terminationInfo.sipStatusCode)
    )
}
```

6.12.2 Objective-C API

The following examples are provided in Swift using Objective-C SDK API:

1. Import the **MVWebRTCNativeCall** framework:

```
import MVWebRTCNativeCall
```

2. Configuring the CXProviderConfiguration object:

```
lazy var providerConfiguration: CXProviderConfiguration = {
    let appDisplayName =
Bundle.main.infoDictionary!["CFBundleDisplayName"] as! String
    let config = CXProviderConfiguration(localizedName:
appDisplayName)
    config.supportsVideo = true
    config.supportedHandleTypes =
[CXHandle.HandleType.phoneNumber]
    config.maximumCallGroups = 4
    config.iconTemplateImageData = UIImage.init(named:
"iconMask")?.pngData()
    config.ringtoneSound = "incoming_call_ringtone.wav"
    if #available(iOS 11.0, *) {
        config.includesCallsInRecents = true
```

```
    }
    return config
}()
```

3. Initializing the API:

```
ACNativeCallService.sharedInstance().initiate(with:
self.providerConfiguration)
```

4. Displaying an incoming call using CallKit (most common approach) upon incoming call event from the SDK:

```
func incomingCall(_ call: AudioCodesSession!, infoAlert:
ACInfoAlertAttributes!)  {

ACNativeCallService.sharedInstance().reportNewIncomingCall(
        call,
        localizedCallerName: call.remoteNumber.displayName,
        answerCallback: { () -> ACCallKitExecutionBlockResult
in
            // Code to update UI for call accept if necessary
            return ACCallKitExecutionBlockResult.undefined
        },
        rejectCallback: { () -> ACCallKitExecutionBlockResult
in
            // Code to update UI for call reject if necessary
            return ACCallKitExecutionBlockResult.undefined
        }
    )
    { (_, error: Error?) in
        // Handle error reporting incoming call to the system
    }
}
```

5. Displaying an incoming call using CallKit, with the customized answer / reject handling with additional SIP headers, upon incoming call event from SDK. (Notice the change in the callback return values.):

```
func incomingCall(_ call: AudioCodesSession!, infoAlert:
ACInfoAlertAttributes!) {
    let answerHeaders = ["Custom-Answer-Header": " Custom
Header Value - Answer"]
    let rejectHeaders = ["Custom-Reject-Header": " Custom
Header Value - Reject"]

ACNativeCallService.sharedInstance().reportNewIncomingCall(
        call,
        localizedCallerName: call.remoteNumber.displayName,
        answerCallback: { () -> ACCallKitExecutionBlockResult
in
            // Code to update UI for call accept if necessary
            call.answer(answerHeaders)
            return ACCallKitExecutionBlockResult.fulfill
        },
        rejectCallback: { () -> ACCallKitExecutionBlockResult
in
            // Code to update UI for call reject if necessary
            call.reject(rejectHeaders)
```

```
            return ACCallKitExecutionBlockResult.fulfill
        }
    )
    { (_, error: Error?) in
        // Handle error reporting incoming call to the system
    }
}
```

6. Initiating an outgoing call:

```
let remoteContact = RemoteContact()
// configure the remoteContact object
if let call = self.phoneUA?.call(remoteContact, withVideo:
true, inviteHeaders: nil) {
    ACNativeCallService.sharedInstance().initiateStartCall(
        call,
        callerName: remoteContact.displayName,
        actionCallback: { () -> ACCallKitExecutionBlockResult
in
            // Optional code to update UI for call initiation
            return ACCallKitExecutionBlockResult.undefined
        }
    )
    { (_, error: Error?) in
        // Handle error initiating call with CallKit
    }
}
```

7. Reporting call updates on call progress events:

```
func callProgress(_ call: AudioCodesSession!) {

ACNativeCallService.sharedInstance().reportCallUpdated(call)
    if call.isOutgoing {
        if isFirstTimeACCallStateCalling {

ACNativeCallService.sharedInstance().reportCallStartedConnecti
ng(call)
        }
        if isFirstTime ACCallStateConnected {

ACNativeCallService.sharedInstance().reportCallEstablished(cal
l)
        }
    }
}
```

8. Terminating a call:

```
ACNativeCallService.sharedInstance().initiateEndCall(
    call,
    actionCallback: nil,
    result: { (_, error: Error?) in
        if (error != nil) {
            call.terminate()
        }
    }
```

```
)
```

9. Reporting a terminated call (required **always** when the `callTerminated` delegate is invoked):

```
func callTerminated(_ call: AudioCodesSession!) {
    ACNativeCallService.sharedInstance().reportCallTerminated(
        call,
        terminationStatusCode: Int32((call?.terminationInfo.
termination)!.rawValue))
    )
}
```

6.13 Using CallKit Manually

6.13.1 Swift API

When using CallKit manually through the Swift API, use `AudioManager` instead of `WebRTCAudioManager`.

1. **Incoming Calls:** When reporting a new incoming call to the CXProvider:

```
let audioManager = AudioManager.shared
audioManager.usesManualAudio = true
audioManager.configureAudioSession(.voip)

let cxUpdate = CXCallUpdate()
// Configure the CXCallUpdate, for example:
// cxUpdate.remoteHandle = CXHandle(type: .phoneNumber, value:
callerNumber)
// cxUpdate.localizedCallerName = callerName
// cxUpdate.hasVideo = call.hasVideo

provider.reportNewIncomingCall(
    with: call.callUUID,
    update: cxUpdate
) { error in
    // handle error reporting incoming call
}
```

2. **Answering Calls:** In `CXAnswerCallAction` handler, use the `.voip` configuration:

```
func provider(_ provider: CXProvider, perform action:
CXAnswerCallAction) {
    let audioManager = AudioManager.shared

    do {
        try audioManager.configureAudioSession(.voip)
    } catch {
        // Handle audio session configuration failure.
        action.fail()
        return
    }

    // Get the Swift call session according to the
action.callUUID.
```

```
    let call: CallSession = /* lookup call by action.callUUID
*/

    call.answer()
    action.fulfill()
}
```

3. **Initiating outgoing calls:** Before initiating a new call, it is recommended to place any active calls on hold by invoking `hold()`. Then creating the new `CallSession` object, and request the CallKit transaction to start the call:

```
func startOutgoingCall(
    to remoteContact: ACWebRTCSdk.RemoteContact,
    withVideo: Bool,
    cxController: CXCallController
) {
    let userAgent = UserAgent.shared
    let audioManager = AudioManager.shared

    // Hold current calls that are not already held.
    for call in userAgent.sessions where !call.isLocallyHeld {
        call.hold()
    }

    // Setup manual audio when CallKit controls audio
activation.
    audioManager.usesManualAudio = true

    // Perform the outgoing call with the Swift SDK API.
    let newCall = userAgent.startCall(
        to: remoteContact,
        options: .init(media: withVideo ? .video : .audio)
    )

    // Create the CXTransaction that would initiate the
CallKit call.
    let handle = CXHandle(
        type: .phoneNumber,
        value: remoteContact.userName ?? ""
    )
    let startCallAction = CXStartCallAction(
        call: newCall.callUUID,
        handle: handle
    )
    startCallAction.isVideo = withVideo
    let transaction = CXTransaction(action: startCallAction)

    cxController.request(transaction) { error in
        // Handle error initiating CallKit transaction.
    }
}
```

4. **Handling call updates and outgoing call established:** When the `callProgress` event for an outgoing `CallSession` call arrives with the connected state, configure the audio session using the `.voip` preset and notify CallKit that the call has connected:

```
func trackCallProgress(
    for call: CallSession,
    provider: CXProvider
) {
    var didReportStartedConnecting = false
    var didReportConnected = false

    call.onProgress = { call in
        let callUpdate = CXCallUpdate()
        callUpdate.hasVideo = call.hasVideo
        callUpdate.remoteHandle = CXHandle(
            type: .phoneNumber,
            value: call.remoteContact?.userName ?? ""
        )
        callUpdate.localizedCallerName =
call.remoteContact?.displayName

        provider.reportCall(
            with: call.callUUID,
            updated: callUpdate
        )

        guard call.isOutgoing else {
            return
        }

        do {
            try
AudioManager.shared.configureAudioSession(.voip)
        } catch {
            // Handle audio session configuration failure.
            return
        }

        if call.callState == .calling &&
!didReportStartedConnecting {
            didReportStartedConnecting = true
            provider.reportOutgoingCall(
                with: call.callUUID,
                startedConnectingAt: Date()
            )
        }

        if call.callState == .connected && !didReportConnected
{
            didReportConnected = true
            provider.reportOutgoingCall(
                with: call.callUUID,
                connectedAt: Date()
```

```
            )
        }
    }
}
```

5. **Handling termination of all calls:** When all calls have terminated, configure the audio session using then `.default` preset:

```
func handleAllCallsTerminated() {
    do {
        try
AudioManager.shared.configureAudioSession(.default)
    } catch {
        // Handle audio session configuration failure.
    }
}
```

6. CallKit**'s audio session activation and deactivation:** CallKit integrates the application's audio with the system in such a way that it elevates the audio session's priority and starts or stops it in conjunction with other calling apps or other calls within the app itself. When handling these events, one must activate / deactivate the audio unit responsible for VOIP processing:

```
func provider(_ provider: CXProvider, didActivate
audioSession: AVAudioSession) {
    AudioManager.shared.audioSessionDidActivate(audioSession)
    AudioManager.shared.isAudioEnabled = true
}

func provider(_ provider: CXProvider, didDeactivate
audioSession: AVAudioSession) {

AudioManager.shared.audioSessionDidDeactivate(audioSession)
    AudioManager.shared.isAudioEnabled = false
}
```

7. **Handling call operations (hold / mute / send DTMF):** In the `CXProviderDelegate` methods that perform call actions, the application must invoke the corresponding SDK methods. For example, mute / unmute action:

```
func provider(_ provider: CXProvider, perform action:
CXSetMutedCallAction) {
    // Get the Swift call session according to
action.callUUID.
    let call: CallSession = /* lookup call by action.callUUID
*/

    call.isAudioMuted = action.isMuted
    action.fulfill()
}
```

6.13.2 Objective-C API

When using CallKit manually, one has to be familiar with the CallKit-related use cases and flows for the various actions related to VOIP calls.

For further details on utilizing CallKit in the application, please refer to the official CallKit documentation.

The following tasks should be performed with the SDK when using CallKit manually:

1. **Incoming Calls:** When reporting a new incoming call to the CXProvider, use the `WebRTCAudioManager` to setup audio:

```
let audioManager = WebRTCAudioManager.getInstance()
audioManager?.useManualAudio = true
audioManager?.configureAudioSession(ACAudioPreset.VOIP)
let cxUpdate: CXCallUpdate = …. // Create a corresponding
CXCallUpdate
provider.reportNewIncomingCall(
    with: call.callUUID,
    update: cxUpdate)
{ (error: Error?) in
    // handle error reporting incoming call
}
```

2. **Answering Calls:** In `performAnswerCallAction`, use the ACAudioPreset.VOIP configuration:

```
func provider(_ provider: CXProvider, perform action:
CXAnswerCallAction) {
    // configure audio session
    audioManager?.configureAudioSession(ACAudioPreset.VOIP)
    // Get the call object according to the action
    let call: AudioCodesSession = .....
    // perform the actual VOIP operation
    call.answer(nil)
    action.fulfill()
}
```

3. **Initiating outgoing calls:** When initiating a new call, it is recommended to first invoke hold on the current calls that are not held. Then, start the call by creating the new AudioCodesSession object, and then request the transaction that would start the call with CallKit:

```
// hold current calls that are not held
// setup manual audio
audioManager?.useManualAudio = true
// perform the outgoing call with the SDK
let remoteContact = RemoteContact()
if let newCall = AudioCodesUA.getInstance()?.call(
     remoteContact,
     withVideo: true, inviteHeaders: nil
) {
    // create the CXTransaction that would initiate the
CallKit call
    cxController.request(transaction) { (error: Error?) in
    }
}
```

4. Handling call updates and outgoing call established: When the `callProgress` event of an outgoing AudioCodesSession call arrives, with the connected state, the audio session configuration should be of the VOIP preset, and CallKit should be notified with the call being connected:

```
func callProgress(_ call: AudioCodesSession!) {
    let provider = CXProvider()
    // report call update to CallKit
    let callUpdate: CXCallUpdate = ....
    provider.reportCall(with: call.callUUID, updated:
callUpdate)
    if call.isOutgoing {

WebRTCAudioManager.getInstance()?.configureAudioSession(ACAudi
oPreset.VOIP)
        if isFirstTimeCallingEvent {
            provider.reportOutgoingCall(with: call.callUUID,
            startedConnectingAt: Date())
        }
        if isFirstTimeConnectedEvent {
            provider.reportOutgoingCall(with: call.callUUID,
connectedAt: Date())
        }
    }
}
```

5. **Handling termination of all calls:** When all calls are terminated, the audio session configuration should become default:

```
WebRTCAudioManager.getInstance()?.configureAudioSession(ACA
udioPreset.default)
```

6. CallKit**'s events for activating / deactivating the audio session:** CallKit integrates the app's audio with the system in such a way that it elevates the audio session's priority, and starts or stops it in conjunction with other calling apps or other calls within the app itself. When using these methods, one must activate / deactivate the audio unit responsible for VOIP processing:

```
func provider(_ provider: CXProvider, didActivate
audioSession: AVAudioSession) {

WebRTCAudioManager.getInstance()?.audioSessionDidActivate(audi
oSession)
    WebRTCAudioManager.getInstance()?.isAudioEnabled = true
}
func provider(_ provider: CXProvider, didDeactivate
audioSession: AVAudioSession) {

WebRTCAudioManager.getInstance()?.audioSessionDidDeActivate(au
dioSession)
    WebRTCAudioManager.getInstance()?.isAudioEnabled = false
}
```

7. **Handling call operations (hold / mute / send DTMF):** In the `CXProviderDelegate` methods for performing the call actions, one must call the SDK methods for the corresponding actions. For example, mute / unmute action:

```
func provider(_ provider: CXProvider, perform action:
CXSetMutedCallAction) {
    let call: AudioCodesSession = ... // get the call object
according to the action
```

```
    call.isAudioMuted = action.isMuted
    // call the appropriate SDK method
    action.fulfill()
}
```

6.14 Responding to Remote Control Events – Genesys 3PCC API

6.14.1 Swift API

1. **Responding to an incoming call with the Alert-Info header data:**

`ACNativeCallService` is not included in the Swift API. Therefore, the example uses CallSession.audioCodesSession only to pass the call to `ACNativeCallService`. All other call handling continues through the Swift API.

```
let phone = UserAgent.shared
phone.callbackQueue = .main

phone.onIncomingCall = { incoming in
    let call = incoming.session

    guard let objcCall = call.audioCodesSession as?
AudioCodesSession else {
        return
    }


ACNativeCallService.sharedInstance().reportNewIncomingCall(
        objcCall,
        localizedCallerName: "",
        answerCallback: {
            if call.callState == .ringing {
                call.answer()
            }
            return .fulfill
        },
        rejectCallback: {
            if call.callState == .ringing {
                call.reject()
            }
            return .fulfill
        },
        result: { _, error in
            guard
                error == nil,
                let alertInfo = incoming.infoAlert,
                alertInfo.autoAnswer,
                alertInfo.delay >= 0
            else {
                return
            }

            DispatchQueue.main.asyncAfter(
```

```
                deadline: .now() + Double(alertInfo.delay)
            ) {
                guard call.callState == .ringing else {
                    return
                }

                call.answer()
                // Update the application UI.
            }
        }
    )

    configureRemoteControlEvents(for: call)
}
```

2. **Responding to incoming Notify events that are associated with a call:**

Configure the accepted NOTIFY event packages before creating or receiving calls:

```
phone.allowedNotifyEvents = [.talk, .hold, .dtmf,
.conference]
```

Attach the event handler to each call:

```
private let remoteDTMFQueue = DispatchQueue(
    label: "com.example.remote-dtmf",
    qos: .userInteractive
)

func configureRemoteControlEvents(for call: CallSession) {
    call.onNotifyEvent = { [weak call] event in
        guard let call else {
            return
        }

        switch event.kind {
        case .talk:
            if call.callState == .connected {
                if call.isLocallyHeld {
                    call.resume()
                }
            } else if !call.isOutgoing && call.callState ==
.ringing {
                call.answer()
            }
            // Update the application UI.

        case .hold:
            if call.callState == .connected &&
!call.isLocallyHeld {
                call.hold()
            }
            // Update the application UI.

        case .dtmf:
            if let digits = event.dtmfString {
```

```
                sendRemoteDTMF(digits, on: call)
            }

        case .conference:
            // Currently not supported.
            break

        case .undefined:
            break
        }
    }
}

func sendRemoteDTMF(_ string: String, on call: CallSession) {
    let options = SDKConfiguration.shared.dtmfConfiguration
    let interval = max(
        options.durationMilliseconds,
        options.intervalGapMilliseconds
    )

    let digits = string.compactMap(CallSession.DTMF.init)

    for (index, digit) in digits.enumerated() {
        remoteDTMFQueue.asyncAfter(
            deadline: .now() + .milliseconds(interval * index)
        ) {
            call.sendDTMF(digit)
        }
    }
}
}
```

6.14.2 Objective-C API

1. Responding to an incoming call with the Alert-Info header data:

```
func incomingCall(_ call: AudioCodesSession!, infoAlert:
ACInfoAlertAttributes!) {

ACNativeCallService.sharedInstance().reportNewIncomingCall(
        call,
        localizedCallerName: "",
        answerCallback: { () -> ACCallKitExecutionBlockResult
in
            // code to answer call from CallKit if enabled
            return .fulfill
        },
        rejectCallback: { () -> ACCallKitExecutionBlockResult
in
            // code to answer call from CallKit if enabled
            return .fulfill
        },
        result: { (_,error: Error? ) in
```

```
            if infoAlert != nil && infoAlert.delay >= 0 &&
infoAlert.autoAnswer && error == nil {
                DispatchQueue.main.asyncAfter(deadline: .now()
+ Double(infoAlert.delay), qos: DispatchQoS.userInteractive) {
                    // answer the call if not already
answered, update GUI
                }
            }
        }
    )
}
```

2. **Responding to incoming Notify events that are associated with a call:**

```
func callNotifyEvent(_ call: AudioCodesSession!, type:
CallNotifyEventType, dtmfString dtmf: String!) {
    switch type {
    case .ACCallNotifyEventTalk:
        DispatchQueue.main.async {
            if call.callState == .ACCallStateConnected {
                // un-hold the call, update GUI
            } else if !call.isOutgoing {
                // answer the call, update GUI
            }
        }
        break
    case .ACCallNotifyEventHold:
        DispatchQueue.main.async {
            if call.callState == .ACCallStateConnected {
                // hold the call, update GUI
            }
        }
        break
    case .ACCallNotifyEventDTMF:
        DispatchQueue.global(qos: .userInteractive).async {
        /*
         perform send DTMF for each character in the dtmf
string parameter. The interval between calls to sendDTMF is
the maximum of
ACConfiguration.getConfiguration()?.dtmfOptions.intervalGap
and ACConfiguration.getConfiguration()?.dtmfOptions.duration
         */
        }
        break
    case .ACCallNotifyEventConference:
        // currently not supported
        break
    default:
        break
    }
}
```

6.15 Push Notifications Use Cases

The Swift API supports the SIP operations required for push-notification workflows, including configuring the server and account, refreshing registration, receiving login-state and incoming-call callbacks, and shutting down the user agent. APNs/VoIP token configuration and native CallKit integration are not part of the Swift API and must use the Objective-C API when required.

If the app uses the AudioCodesUA.setPushNotification API with valid parameters, then for the purpose of supporting incoming call push notifications, the SBC can initiate the delivery of two types of push notifications to the app:

- Trigger SIP registration refresh (APNS notification)
- Notify Incoming call (VOIP Push notification)

The following use cases must be implemented in the application, in order to adhere to the following rules:

- Apple enforces applications to display a CallKit incoming call screen **immediately** upon receiving a VOIP push notification, before any SIP message arrives for the incoming SIP call. This also means that when using push notifications, **the application MUST use** CallKit**.**
- Since iOS 13, Apple does not allow using VOIP push for any other purposes than incoming calls.
- Incoming call notifications indicate that there is a pending INVITE awaiting the application. The application must wake up and perform REGISTER by calling `login()`, so that the SBC can deliver the pending INVITE message. The SBC guarantees that the pending INVITE will be delivered only after the REGISTER completes.
- The SBC is responsible for maintaining registrations, by initiating a registration-refresh push, to wake the application to perform REGISTER. The application is expected to do so **automatically** from every possible state, including being terminated.
- The registration expiration interval is large (at least 24 hours). Upon expiration or UnREGISTER, the SBC discards the device tokens, and deems the user unreachable for push calls.

6.15.1 Handling the Application Transition to Background

The application must always perform a logout when going to the background state. However, when using push notifications, the application has to avoid sending an un-REGISTER when calling logout, for push functionality to work. Therefore, the application must call logout: `ACUALogoutModeForceShutdown` in that case:

```
func applicationDidEnterBackground(_ application: UIApplication) {
    // begin a background task which will be ended at
loginStateChanged
        shutdownBackgroundTask =
UIApplication.shared.beginBackgroundTask { [weak self] in
            guard let self = self else { return }

UIApplication.shared.endBackgroundTask(self.shutdownBackgroundTask
)
            self.shutdownBackgroundTask =
UIBackgroundTaskIdentifier.invalid
        }
    // if we use push notifications, logout without unREGISTER
```

```
    let usingPushNotifications = ...
    if usingPushNotifications {
        AudioCodesUA.getInstance().logout(.forceShutdown)
    } else {
        AudioCodesUA.getInstance().logout()
    }
}
```

6.15.2 Handling SIP Registration-Refresh Notifications

The application must be able to reliably wake-up and send a REGISTER from every possible state, without being dependent on user interactions to do so. This is not trivial for APNS push, and can be achieved by using one of the following strategies:

6.15.2.1 Using Background ("silent") APNS notifications

Pros:

1. Easiest to implement in the application.
2. Truly silent, has no requirements on presentation to the user in any way

Cons:

1. This is not reliable enough Background notifications are lower in priority, and their delivery can be throttled in various conditions, especially after being sent multiple times per hour.

Requirements:

1. The application info.plist must include the remote-notification entry under `UIBackgroundModes`.
2. The push server must be configured to send background notifications.

Refreshing registration upon receiving background APNS push notification:

```
func application (
        _ application: UIApplication,
        didReceiveRemoteNotification userInfo: [AnyHashable
: Any],
        fetchCompletionHandler completionHandler: @escaping
(UIBackgroundFetchResult) -> Void
    ) {
        if let pushType = userInfo["push_type"] as? String,
pushType == "REGISTER" {
            // Here we perform login to refresh
registration
            AudioCodesUA.getInstance().login()
        }
        completionHandler(.noData)
    }
```

6.15.2.2 Using the Notification Service App Extension

Pros:

1. This is by far the most reliable method. Notifications delivery is not throttled, and the extension can use the SDK to refresh registration from every possible application state.

Cons:

1. This is harder to implement. It requires implementing the App Extension, as well as sharing SIP account and configuration information between the extension and the containing application, so that the extension can call the SDK APIs to configure an AudioCodesUA instance to perform login properly.
2. Automatic, but not entirely silent. The extension can automatically perform login, however the notification must be displayed in some form to the user, even without requiring the user's interaction. For example, the notification alert can only have a text message, indicating that SIP registration was refreshed successfully.

Requirements:

1. Setting up the Notification Service App Extension, and shared storage with the application, based on App Groups.
2. The extension must use the MVWebRTCFramework.xcframework bundle, which is extension-safe, and must **not** use the MVWebRTCInterface.xcframework.
3. There should be a mechanism that would inform the extension, that the containing app is in the foreground. That is because it is strongly advised that the extension does NOT perform a REGISTER when the containing app is in the foreground and is managing registration as well. Having these two processes performing registration in parallel can cause SIP registration errors.

1. **Notification Service Extension Class – Main Entry Point:**

```
override func didReceive(
    _ request: UNNotificationRequest,
    withContentHandler contentHandler: @escaping
(UNNotificationContent) -> Void
) {
    self.contentHandler = contentHandler
    self.bestAttemptContent = (request.content.mutableCopy()
as? UNMutableNotificationContent)
    guard let bestAttemptContent = bestAttemptContent else {
        return
    }
    // If this is a register refresh push, setup AudioCodesUA
and perform login.
    if let pushType = bestAttemptContent.userInfo["push_type"]
as? String, pushType == "REGISTER" {
    bestAttemptContent.title = "VOIP Registration"
    /* If the containing application is in foreground, do
nothing in order not to interfere with its existing
registration. The containing application will handle the
notification. */
        if sharedStorage.appInForeground {
            contentHandler(bestAttemptContent)
            return
        }
        // load AudioCodesUA account, device tokens, and other
configurations data from shared storage, and login.
        // The content handler will be called from the
loginStateChanged delegate callback.
        AudioCodesUA.getInstance().setAccount(...)
        AudioCodesUA.getInstance().setServerConfig(...)
// Note that for setPushNotification…, the bundleId parameter
must be the bundle identifier of the CONTAINING APP.
AudioCodesUA.getInstance().setPushNotificationsTeamId(...)
        AudioCodesUA.getInstance().regExpires = ...
        ACConfiguration.getConfiguration().localServerPort =
...
        // Listening to login state change events. We use
notification center because multiple instances of the service
extension can be active in parallel.
        NotificationCenter.default.addObserver(self, selector:
#selector(loginStateChangedNotification(_:)), name:
.ACUALoginStateChanged, object: nil)
        AudioCodesUA.getInstance().login()
        return
    }
}
```

2. **Notification Service Extension Class – Handling notification delivery in loginStateChanged:**

```
        @objc func loginStateChangedNotification(_
notification: Notification) {
        guard let userInfo = notification.userInfo else {
            return
        }
        let isLogin =
userInfo[ACUALoginStateIsLoginUserInfoKey] as? Bool ?? false
        let cause = userInfo[ACUALoginStateCauseUserInfoKey]
as? String ?? ""
        // If cause isn't error, then this is our logout
event. In that case, finish.
        if cause == ACUALoginChangedReasonDisconnected {
            return
        }
        // login operation completed, so we shut down
AudioCodesUA. We force-close to avoid sending un-REGISTER.
        NotificationCenter.default.removeObserver(self, name:
.ACUALoginStateChanged, object: nil)
        AudioCodesUA.getInstance().logout(.forceShutdown)
        if isLogin {
            // For a successful registration event, it should
be as non-intrusive as possible, so no sound needed.
            bestAttemptContent?.sound = nil
            bestAttemptContent?.body = "Updated registration
to SIP host successfully."
        } else {
            bestAttemptContent?.body = "Not registered. Cause:
\(cause)"
        }
        contentHandler(bestAttemptContent)
    }
```

3. **Notification Service Extension Class – Handling Expiration:**

```
    override func serviceExtensionTimeWillExpire() {
        // Shut down AudioCodesUA
        NotificationCenter.default.removeObserver(self, name:
.ACUALoginStateChanged, object: nil)
        AudioCodesUA.getInstance().logout(.forceShutdown)
        bestAttemptContent?.body = "Timeout reached when
handling notification"
        contentHandler(bestAttemptContent)
    }
```

4. **Main App – UIApplicationDelegate Relevant Methods – Coordinate Notification Handling With The Extension**

```
    func application(_ application: UIApplication,
                     didFinishLaunchingWithOptions
launchOptions: [UIApplication.LaunchOptionsKey: Any]?
    ) -> Bool {
..................
    // In case of crash recovery, write the default value for
this flag
    sharedStorage.storeAppInForeground(false)
..................
    }
    func applicationWillEnterForeground(_ application:
UIApplication) {
        sharedStorage.storeAppInForeground(true)
    }
    func applicationDidBecomeActive(_ application:
UIApplication) {
        sharedStorage.storeAppInForeground(true)
    }
    func applicationDidEnterBackground(_ application:
UIApplication) {
        sharedStorage.storeAppInForeground(false)
    }
    func applicationWillTerminate(_ application:
UIApplication) {
        sharedStorage.storeAppInForeground(false)
    }
```

5. **Main App – UNUserNotificationCenterDelegate Relevant Methods – Coordinate Notification Handling With The Extension**

```
    func userNotificationCenter(
        _ center: UNUserNotificationCenter,
        willPresent notification: UNNotification,
        withCompletionHandler completionHandler: @escaping
(UNNotificationPresentationOptions) -> Void
    ) {
        let userInfo = notification.request.content.userInfo
        if let pushType = userInfo["push_type"] as? String,
pushType == "REGISTER" {
    // Received register refresh notification in Foreground.
The extension did not handle this.
    // Refreshing SIP Register from push notification
            AudioCodesUA.getInstance().login()
        }
        completionHandler([])
    }
    func userNotificationCenter(
        _ center: UNUserNotificationCenter,
        didReceive response: UNNotificationResponse,
        withCompletionHandler completionHandler: @escaping ()
-> Void
    ) {
        let userInfo =
response.notification.request.content.userInfo
        if let pushType = userInfo["push_type"] as? String,
pushType == "REGISTER" {
    // The user pressed the register refresh notification
alert. This notification was already handled in the extension.
    completionHandler()
    return
        }
    }
```

6.15.3 Handling Incoming Call Notifications

Handling incoming call notifications is performed by the following:

1. Parse call details from the notifications payload: caller name, caller display name (optional), is the call video (optional)
2. Use the SDK to report an incoming CallKit call from the notification
3. Call AudioCodesUA.login() to perform REGISTER, after which the pending incoming INVITE will arrive.
4. Upon the INVITE arrival, within the `incomingCall` delegate callback, use the `MVWebRTCNativeCall` framework to call to ACNativeCallService.reportNewIncomingCall, to associate the CallKit report of the push call to the incoming SIP call. See section 6.12.

If the push payload doesn't include information that this is a video call, then the SDK updates it automatically in step #4.

PKPushRegistryDelegate – Handling Incoming Call Notification

```
    func pushRegistry(
        _ registry: PKPushRegistry,
        didReceiveIncomingPushWith payload: PKPushPayload,
        for type: PKPushType
    ) {
        let caller = RemoteContact()
        /*
        Parse incoming call details from the payload. If no
SIP username is available, we must use a default one,
"unknown", in order to display the CallKit screen.
         */
        caller.userName =
payload.dictionaryPayload["caller_sip_username"] as? String ??
"unknown"
        // remove SIP domain name from the caller username
        caller.userName =
caller.userName.components(separatedBy: "@").first
        // obtain optional SIP display name
        caller.displayName =
payload.dictionaryPayload["caller_sip_displayname"] as? String
        // obtain optional video flag value
        let isVideoCallString =
(payload.dictionaryPayload["call_has_video"] as? String) ??
"false"
        let isVideoCall = isVideoCallString == "true" ? true :
false
        // Generate a CallKit call with the SDK. We must do
this immediately upon receiving the notification.
        let acnotification =
ACIncomingCallPushNotification(caller: caller, hasVideo:
isVideoCall)

ACNativeCallService.sharedInstance().report(acnotification!,
result: nil)
   /*
         Here we trigger the client to perform SIP register.
         After registration completes, an incoming SIP call
will arrive, corresponding to the push call.
         upon the incoming SIP call, we will call the
ACNativeCallService reportIncomingCall,
         which will automatically associate the SIP call to
the CallKit call that is generated here.
         */
        AudioCodesUA.getInstance().login()
    }
```

6.16 Handling Audio Interruptions and GSM Calls

When there are existing calls, and the application receives an audio interrupt, we distinguish between whether CallKit is used or not:

6.16.1 Using CallKit

With CallKit, the application is granted the highest audio usage priority, and so during calls, it cannot be interrupted by other audio playback from other apps. However, it might be interrupted by apps with a similar level of priority: Either the native Phone app, or apps that use CallKit as well, and receive incoming calls.

In that case, audio interruptions are managed as part of CallKit usage, and there is no special handling required from the SDK perspective.

Call-related events that are integrated with other apps, either by switching from this app to a native phone call, or by accepting a call from another CallKit app and holding the current call, are automatically managed by the system, which either holds or unholds or terminate the call according to the user's input to the native Telephony GUI.

Note that the `ACNativeCallService` provides the best handling of audio interruptions automatically.

`ACNativeCallService` is not part of the Swift API and must be accessed through the Objective-C API when its automatic CallKit handling is required. Applications that implement CallKit manually can use AudioManager.shared to handle audio session activation and deactivation, and `CallSession` to hold, resume, or terminate calls

6.16.2 Not Using CallKit

6.16.2.1 Swift API

When CallKit is not used, the application should observe the system AVAudioSession.interruptionNotification directly. When an interruption begins, established calls should be placed on hold, and calls that are not yrt established should be terminated. When the interruption ends and the system indicates that audio may resume, calls that were held because of the interruption can be resumed through the Swift `CallSession` API.

See example below:

```
private var interruptionHeldCallIDs = Set<UUID>()

func setupAudioInterruptObserver() {
    NotificationCenter.default.addObserver(
        self,
        selector: #selector(handleAudioInterrupt(_:)),
        name: AVAudioSession.interruptionNotification,
        object: AVAudioSession.sharedInstance()
    )
}

@objc
private func handleAudioInterrupt(_ notification: Notification) {
    guard
        let value = notification.userInfo?[
            AVAudioSessionInterruptionTypeKey
        ] as? NSNumber,
        let type = AVAudioSession.InterruptionType(
            rawValue: value.uintValue
        )
```

```
    else {
        return
    }

    switch type {
    case .began:
        interruptionHeldCallIDs.removeAll()

        for call in UserAgent.shared.sessions {
            if call.callState == .connected {
                guard !call.isLocallyHeld else {
                    continue
                }

                call.hold()
                interruptionHeldCallIDs.insert(call.callUUID)
            } else {
                call.terminate()
            }
        }

        // Update the application UI.

    case .ended:
        let optionsValue = notification.userInfo?[
            AVAudioSessionInterruptionOptionKey
        ] as? NSNumber

        let options = AVAudioSession.InterruptionOptions(
            rawValue: optionsValue?.uintValue ?? 0
        )

        guard options.contains(.shouldResume) else {
            return
        }

        for call in UserAgent.shared.sessions
        where interruptionHeldCallIDs.contains(call.callUUID) {
            if call.callState == .connected && call.isLocallyHeld
{
                call.resume()
            }
        }

        interruptionHeldCallIDs.removeAll()
        // Update the application UI.

    @unknown default:
        break
    }
}
```

6.16.2.2 Objective-C API

When not using CallKit, the app audio usage priority is lower, and might compete with other apps that seek to obtain audio resources. It is the system's responsibility to allocate audio resources to different apps, and notify each app's Audio-Session whether its audio resources are unavailable, or become available again. This is done via audio interruption notifications.

The SDK API includes the `ACAudioInterruptNotification` from `WebRTCAudioManager` (see section 5.4.2), which delivers the system's audio interrupt notification in a streamlined manner.

Generally speaking, when not using CallKit and the audio session is interrupted, all established calls must be put on hold, and all non-established calls must be terminated.

When audio interruption ends and the app may resume audio usage, the held calls may be resumed.

See example below:

```
    func setupAudioInterruptObserver() {
        NotificationCenter.default.addObserver(self,
                                               selector:
#selector(handleAudioInterrupt(notification:)),
                                               name:
.ACAudioInterrupt,
                                               object: nil)
    }


    @objc func handleAudioInterrupt(notification: Notification) {
        guard let userInfo = notification.userInfo,
              let isInterrupttionBegin =
userInfo[ACAudioIsInterruptedUserInfoKey] as? Bool,
              let session = self.obtainActiveSesison()
        else {
            return
        }
        if (isInterrupttionBegin) {
            // hold the current call, notify user GUI that the
call is interrupted
            session.hold(true)

            // ...terminate all sessions that are not established
        } else {
            // unhold the current call
            session.hold(false)
        }
    }
```

6.17 Binding SIP Connections

6.17.1 Swift API

In the Swift API, SIP connection binding is configured through LoginOptions.connectionBinding when the user agent logs in. Use `.deferred` for the recommended address-family-independent binding with registration, `.ipv4` or `.ipv6` when the address family must be selected before connecting, and `.none` when connection binding is not required. The binding cannot be changed or removed until the user agent is shut down.

See example below:

```
let userAgent = UserAgent.shared

func getLocalIPAddressFamily()
    -> NetworkConnectionAttributes.AddressFamily {
    /*
     Determine whether the active connection uses IPv4 or IPv6.
     Return .unspecified when it cannot be determined.
     */
}

func bindingForCurrentAddressFamily() -> ConnectionBinding? {
    switch getLocalIPAddressFamily() {
    case .ipv4:
        return .ipv4
    case .ipv6:
        return .ipv6
    case .unspecified:
        return nil
    }
}

func setupUserAgent() {
    userAgent.configureAccount(accountConfiguration)
    userAgent.configureServer(serverConfiguration)

    let connectionBinding: ConnectionBinding

    if !shouldBindSIPConnection {
        connectionBinding = .none
    } else if shouldAutoRegister {
        // Bind to the connection selected during registration.
        connectionBinding = .deferred
    } else {
        // Without registration, the address family must be known
in advance.
        guard let explicitBinding =
bindingForCurrentAddressFamily() else {
            return
        }
        connectionBinding = explicitBinding
    }
```

```
    let loginOptions = LoginOptions(
        autoRegister: shouldAutoRegister,
        connectionBinding: connectionBinding
    )

    userAgent.login(loginOptions)
}

func networkHasChanged() {
    /*
     Network-change handling applies to a registered account.
Supplying IPv4
     or IPv6 reapplies connection binding for the new address
family.
     */
    guard shouldAutoRegister else {
        return
    }

    let addressFamily = getLocalIPAddressFamily()

    switch addressFamily {
    case .ipv4, .ipv6:
        userAgent.handleNetworkChange(
            NetworkConnectionAttributes(
                localAddressFamily: addressFamily
            )
        )

    case .unspecified:
        userAgent.handleNetworkChange()
    }
}
```

6.17.2 Objective-C API

This demonstrates the usage described in section 4.1.2.16 of the `setConnectionBinding` method.

SIP connection binding forces the SIP account to reuse the current SIP connection for all outgoing messages. To maintain proper operation, this also requires an enhancement to network change handling.

See example below on how to configure SIP connection binding, and handling network change:

```
    func getLocalIpAddressFamily() -> ACNetworkAddressFamily {
        /*
         Obtain the local ip-address family: "IPV4", "IPV6" or
"Unspecified"
         NOTE: Applications should implement finding the ip
address family in the way most suited to their needs.
         */
```

```
    }

    func setupPhoneUA() {
        self.phoneUA = AudioCodesUA.getInstance()

        // ..... Call SIP account setters before login
        self.phoneUA?.setAccount(...
        self.phoneUA?.setServerConfig(...

        // Manage SIP connection binding
        if shouldBindSipConnection {
            // Defer binding to the currently established
connection, which would be agnostic to the IP-address family. This
is the recommended mode.
            let attr =
ACNetworkConnectionAttributes.attr(withLocalAddressFamily:
.unspecified)
            if !shouldAutoRegister {
                // If we make calls without registration, then we
must determine the ip-address family in advance.
                attr.localAddressFamily =
getLocalIpAddressFamily()
            }
            self.phoneUA?.setConnectionBinding(attr)
        } else {
            // Remove SIP connection binding from the SIP account
            self.phoneUA?.setConnectionBinding(nil)
        }

        // ......
        self.phoneUA?.login(shouldAutoRegister)
    }

    func networkHasChanged() {
        /*
         Determine the ip address family to pass down to the
network change handler.
         Finding the ip-address family is optional, and is
important when using the setConnectionBinding API. Alternatively,
handleNetworkChange can receive a nil parameter.
         */
        let ipAddressFamily = getLocalIpAddressFamily()
        let attr =
ACNetworkConnectionAttributes.attr(withLocalAddressFamily:
ipAddressFamily)
        self.phoneUA?.handleNetworkChange(attr)

    }
```

International Headquarters
Naimi Park
6 Ofra Haza Street
Or Yehuda, 6032303, Israel
Tel: +972-3-976-4000
Fax: +972-3-976-4040

AudioCodes Inc.
80 Kingsbridge Rd
Piscataway, NJ 08854, USA
Tel: +1-732-469-0880
Fax: +1-732-469-2298

Contact us: https://www.audiocodes.com/corporate/offices-worldwide
Website: https://www.audiocodes.com

Document #: **LTRT-14097**