



Application Enablement Services

Device, Media and Call Control API

.NET Programmers Guide

Release 4.1

An Avaya MultiVantage[®] Communications
Application

02-602658
Issue 1.0
December 2007

Notice

While reasonable efforts were made to ensure that the information in this document was complete and accurate at the time of printing, Avaya Inc. can assume no liability for any errors. Changes and corrections to the information in this document may be incorporated in future releases.

For full support information, please see the complete document, *Avaya Support Notices for Software Documentation*, document number 03-600758.

To locate this document on our Web site, simply go to <http://www.avaya.com/support> and search for the document number in the search box.

Documentation disclaimer

Avaya Inc. is not responsible for any modifications, additions, or deletions to the original published version of this documentation unless such modifications, additions, or deletions were performed by Avaya. Customer and/or End User agree to indemnify and hold harmless Avaya, Avaya's agents, servants and employees against all claims, lawsuits, demands and judgments arising out of, or in connection with, subsequent modifications, additions or deletions to this documentation to the extent made by the Customer or End User.

Link disclaimer

Avaya Inc. is not responsible for the contents or reliability of any linked Web sites referenced elsewhere within this documentation, and Avaya does not necessarily endorse the products, services, or information described or offered within them. We cannot guarantee that these links will work all of the time and we have no control over the availability of the linked pages.

Warranty

Avaya Inc. provides a limited warranty on this product. Refer to your sales agreement to establish the terms of the limited warranty. In addition, Avaya's standard warranty language, as well as information regarding support for this product, while under warranty, is available through the following Web site: <http://www.avaya.com/support>.

Copyright

Except where expressly stated otherwise, the Product is protected by copyright and other laws respecting proprietary rights. Unauthorized reproduction, transfer, and or use can be a criminal, as well as a civil, offense under the applicable law.

Avaya support

Avaya provides a telephone number for you to use to report problems or to ask questions about your product. The support telephone number is 1-800-242-2121 in the United States. For additional support telephone numbers, see the Avaya Web site: <http://www.avaya.com/support>.

Contents

Scope of this document	7
Intended Audience.	7
Conventions used in this document	8
Related documents	9
Application Enablement Services documents.	9
Communication Manager documents	10
ECMA documents	10
Providing documentation feedback	11
Chapter 1: API Services.	13
Exposed .Net classes	13
Service Provider Class	13
Device Class	16
Phone Class	17
Media Class	20
Call Information Link Class	22
Third Party Call Controller Class	23
XML Processor Class	27
Constant Classes	28
Chapter 2: Getting started	29
Setting up the development environment	29
Downloading the Application Enablement Services Device, Media and Call Control .NET API SDK	29
Setting up your test environment.	30
Understanding basic CSTA concepts	30
Devices	31
Physical elements	31
Logical elements.	31
Calls	32
Events	32
Negative acknowledgements	32
Signaling Encryption	33
Media Encryption	33
Accessing the client API reference documentation	34
Using the Device, Media and Call Control Dashboard	34
Learning from sample code.	34

Chapter 3: Writing a client application	37
The Dashboard tool	37
Expanded help feature.	48
Writing Code	50
Writing applications that run in a browser.	63
Registering devices	63
Controllable telephone types	64
Registration modes	64
The "share-talk" button	70
Choosing a codec	71
Choosing the media encryption	72
Telephony Logic	73
Knowing what buttons are administered.	73
Detecting an incoming call	73
Determining that far end has ended the call	73
Getting ANI information for a call.	74
Recording and playing voice media	74
Media Operations	76
Determining when far-end RTP media parameters change.	76
Recovery	76
Cleanup.	78
Media Encryption	79
The AES Encryption Scheme	80
Specifying the Devices' Encryption Capability	84
MediaStartEvent Handling	84
Media Encryption Information	85
Encrypting and Decrypting the RTP Stream	85
Roll Over Counter (ROC)	85
Creating the Encryption Keys Using the Pseudo Random Function.	86
Creating the Initialization Vectors (IV)	87
Decrypting the Media Payload	88
Test Data	88
Security considerations.	89
Chapter 4: Debugging	91
Debugging	91
Possible race conditions	92
Improving performance	92
Getting support	93

Appendix A: The J-Scripts Library	95
J-Script	95
ActiveX Service Provider	96
Static event handlers	96
Requests, response events, and events	97
J-Script object	99
Service Provider Methods and Events	99
Device Methods and Events.	100
Phone Methods and Events.	101
.	Media
Methods and Events	102
Third Party Call Controller Methods and Events	103
Call Information Methods and Events	103
J-Script library configuration	103
Server-side Instructions.	104
Client-side Instructions	104
Appendix B: Communication Manager Features	107
Appendix C: Constant Values	111
Phone Object Enumerations and Constants.	111
Media Enumerations.	123
Service Provider Enumeration	123
Third Party Call Control Enumerations.	124

Contents

About this document

This chapter describes the:

- [Scope of this document](#)
- [Intended Audience](#)
- [Conventions used in this document](#)
- [Related documents](#)
- [Providing documentation feedback](#)

Scope of this document

This document shows you how to use the Avaya MultiVantage Application Enablement (AE) Services Device, Media and Call Control API to develop, debug, and deploy .NET applications that require third party device, media and call control.

- [Chapter 1: API Services](#) provides background information about the AE Services Device, Media and Call Control API and CSTA.
- [Chapter 2: Getting started](#) gets you ready to program to this API.
- [Chapter 3: Writing a client application](#) and [Chapter 4: Debugging](#) guide you in developing and debugging applications.
- [Appendix B: Communication Manager Features](#) lists the Communication Manager features that your application can take advantage of.
- [Appendix E: TSAPI Error Code Definitions](#) appendix lists all of the values for the TSAPI error codes.
- The [Glossary](#) defines the terminology and acronyms used in this document.

Intended Audience

This document is written for applications developers. The .Net Client Library has been tested with application written in the following languages:

- C#
- Visual Basic

About this document

In addition a developer should be familiar with:

- telephony concepts

You do not need to understand CSTA concepts or Avaya Communication Manager features and concepts, but they both might be helpful.

If you are new to CSTA, you may wish to start by reading *ECMA-269*, section 6.1, “CSTA Operational Model: Switching Sub-Domain Model”. Also become familiar with the table of contents so that you know the kinds of information available there. All of the descriptions of the services implemented by this API are also found in the *Avaya Application Enablement Services Device, Media and Call Control .NET Programmers Reference*, found online on the Avaya DevConnect website (<http://www.avaya.com/devconnect>) and on the Avaya Support Centre website (<http://www.avaya.com/support>) under “Communication Systems”.

For those new to Avaya Communication Manager, you may wish to take a course from Avaya University (<http://www.avaya.com/learning>) to learn more about Communication Manager and its features. It is recommended that you start with the *Avaya Communication Manager Overview* course (course ID AVA00383WEN). You may also wish to peruse [Appendix B: Communication Manager Features](#) in this guide to get some ideas of how applications can take advantage of Communication Manager’s abilities.

Conventions used in this document

The following fonts are used in this document:

To represent...	This font is used...
Code and Linux commands	<code>request = new GetDeviceId();</code>
.NET class, method and field names	the <code>getDeviceID</code> method
Window names	The buttons are assigned on the Station form.
Browser selections	Select Member Login
Hypertext links	Go to the http://www.avaya.com/support website. The term connector can be found in the glossary. For details, see The Dashboard tool on page 37.

Related documents

While planning, developing, deploying, or troubleshooting your application, you may need to reference other Avaya MultiVantage Application Enablement Services documents, Avaya Communication Manager documents, or CSTA documents listed below.

Application Enablement Services documents

For developers, the other important source of .NET API information is the programmers reference:

- *Dashboard User's Guide*
- *ReadMe.txt*

Note:

You will find both of the above documents in the SDK .zip file

- *Avaya MultiVantage® Application Enablement Services Device, Media and Call Control .NET Programmer Reference*

Here you can find details about each package, interface, class, method, and field in the API.

Other Application Enablement Services documents include:

- *Avaya MultiVantage® Application Enablement Services Overview (02-300360)*
- *Avaya MultiVantage® Application Enablement Services Installation and Upgrade Guide for a Software Only Offer (02-300355)*
- *Avaya MultiVantage® Application Enablement Services Installation and Upgrade Guide for a Bundled Server (02-300356)*
- *Avaya MultiVantage® Application Enablement Services Administration and Maintenance Guide (02-300357)*
- *Avaya Application Enablement Services Device, Media and Call Control API XML Programmer Guide (02-300358)*
- *Avaya MultiVantage® Application Enablement Services Device, Media and Call Control XML Programmer Reference (XMLdoc)*
- *Avaya MultiVantage® Application Enablement Services Device, Media and Call Control API Web Services Programmer Guide ((02-300362)*
- *Avaya MultiVantage® Application Enablement Services Documentation Roadmap (02-300361)*
- *Avaya MultiVantage® Application Enablement Services OAM Help (HTML)*

About this document

- *Avaya MultiVantage® Application Enablement Services 3.1 TSAPI, JTAPI, and CVLAN Client Installation Guide* (02-300543)
- *Avaya MultiVantage® Application Enablement Services 3.1 TSAPI for Avaya Communication Manager Programmer Reference* (02-300544)
- *Avaya MultiVantage® Application Enablement Services 3.1 TSAPI Programmer Reference* (02-300545)
- *Avaya MultiVantage® Application Enablement Services 3.1 CVLAN Programmer Reference* (02-300546)
- *Avaya MultiVantage® Application Enablement Services 3.1 Java Telephony API (JTAPI) Programmer Reference (JTAPI v1.2 Specification)* (02-300547)
- *Avaya MultiVantage® Application Enablement Services 3.1 Java Telephony API (JTAPI) for Avaya MultiVantage Programmer's Guide* (02-300548)
- *Avaya MultiVantage® Application Enablement Services 3.1 ASAI Technical Reference* (02-300549)
- *Avaya MultiVantage® Application Enablement Services 3.1 ASAI Protocol Reference* (02-300550)

You can find all these documents online on the Avaya Support Center Web Site (<http://support.avaya.com>).

Communication Manager documents

Since this API gives you programmable access to Avaya Communication Manager features, you may wish to reference documents about that system.

The following documents from the Communication Manager documentation set provide additional information about administering Communication Manager for Device, Media and Call Control API. They are on the Avaya Support Centre Web Site (<http://www.avaya.com/support>).

- *Administrator's Guide for Avaya Communication Manager* (Issue 3.1 for CM 4.0), (03-300509)
- *Administration for Network Connectivity for Avaya Communication Manager* (Issue 12 for CM 4.0), (555-233-504_10)

ECMA documents

The *.NET Programmers Reference* contains much of what you need to know about CSTA services. For CSTA details not found in this document, please refer to the following documents. They are found in the Publications section of the ECMA web site (<http://www.ecma-international.org/>):

- *ECMA-269: Services for Computer Supported Telecommunications Applications (CSTA) Phase III*
- *ECMA-323: XML Protocol for Computer Supported Telecommunications Applications (CSTA) Phase III*
- *ECMA-354: Application Session Services*
- *ECMA Technical Report TR/72: Glossary of Definitions and Terminology for Computer Supported Telecommunications Applications (CSTA) Phase III*

Providing documentation feedback

Let us know what you like or do not like about this document. Although we cannot respond personally to all your feedback, we promise we read each response we receive.

Please email feedback to **document@avaya.com**

Thank you.

Chapter 1: API Services

This chapter provides an overview of what services this API supports. This API supports the following telephony services:

- device control
- media control
- call control

These services are provided through a .NET API interface. The interface will follow the event based asynchronous pattern. The exposed .Net classes will model all requests and response (or exceptions) asynchronously and will model all events and handlers using standard .Net conventions.

Exposed .Net classes

This API exposes the following .Net classes:

- Service Provider Class
- Device Class
- Phone Class
- Media Class
- Call Information Link Class
- Third Party Call Controller Class
- XML Processor Class
- Constant Classes

Service Provider Class

The Service Provider Class is used to establish and maintain a relationship between an application and a server for the purpose of exchanging application messages.

This class acts as a bootstrap class for connecting to the server, performing session management, retrieving other objects and disconnecting from the server. To learn how to use the Service Provider Class, see [The Dashboard tool](#) on page 37

The Service Provider Class provides the following asynchronous operations:

Table 1: Service Provider Class Requests 1 of 3

Method Name	Description
ConvertDialStringToE164	Convert one or more dial strings to E164 format.
ConvertE164ToDialString	Convert one or strings which are in E164 format to dial strings.
Get Auto Keep Alive Enabled	Lets the application know if auto keep alive is enabled.
Get Device	Given a device name (Id), returns a reference to the associated Device.
Get Device ID List	Get all the Device IDs that are active on the DMCC server.
Get Monitor List	Get a list of all the monitors which are active on this DMCC server.
Get New Device	Creates a new Device object and returns a reference to it.
Get Session ID List	Retrieve all the active sessions that are running in a DMCC server.
Reconnect	Sends a StartApplicationSession message to the server. If sessionId is not null then the server will try to reconnect a session that has timed out or become inactive.
Reset Application Session	Sends a ResetApplicationSession XML message to the server.
Session Management Start Monitor	Start the monitor for session management.
Session Management Stop Monitor	Stop a session management monitor.
1 of 3	

Table 1: Service Provider Class Requests 2 of 3

Method Name	Description
Shut Down	Allows the application to tell the Service Provider to shutdown. The Service Provider will in turn notify ThirdPartyCallControl, CallInformationLink, and all the Devices that have been created to also shut down. The Devices in turn will notify their Phone objects. The Phone Objects will in turn notify their Media Objects. Calling this method will also force the socket to the server to be closed. Therefore, once this method returns control to the application the application should not invoke any methods which will result in a message being sent to the server. If the application does do this then exceptions will be thrown back to the application. Note: The API provides a callback (OnObjectDeactivatedEvent) for all publicly exposed objects. This will allow the application the ability to easily be notified when an object has been shutdown (deactivated).
Start Application Session	Initiates an application session between an application and a server
Start Auto Keep Alive	Tells the API to start automatically sending the ResetApplicationSession messages every InternalInMs milliseconds. 1,000 milliseconds is 1 second.
Stop Application Session	Terminates an existing application session.
Stop Auto Keep Alive	Stops the ResetApplicationSession messages from automatically being sent to the server.
Transfer Monitor Objects	Transfer all the devices and monitors from one session to another session. The session which the monitors are being transferred from will be closed by the DMCC server.
Get Actual Protocol Version	Retrieves the API version for the server.
Get Call Information Link	Returns a reference to the CallInformationLink object.
Get Session ID	Returns the session name (id) as a string.
2 of 3	

Table 1: Service Provider Class Requests 3 of 3

Method Name	Description
Get Third Party Call Controller	Returns a reference to the ThirdPartCallControl object.
Get Xml Processor	Returns a reference to the XML Processor object.
3 of 3	

The Service Provider Class raises the following events:

Table 2: Service Provider Events

Event Name	Description
Missed At Least One Keep Alive Event	
Server Connection Down	Occurs if the socket to the AE Services server has gone down for some reason
Server Connection Not Active	Occurs if a message was received by the AE Services server but the session has timed out and been placed in the inactive state

Device Class

The Device Class will be used to communicate to the server in order to get a new DeviceID for a particular extension, The device object contains the phone object. There will be one Device Object for every device that is being used in the application. To learn how to use the Device Class, see [The Dashboard tool](#) on page 37.

The Device Class provides the following asynchronous operations:

Table 3: Device Class Requests

Method Name	Description
Get Device ID As String	Gets the device identifier that represents the device described by its extension number and the Communication Manager upon which it resides, as a string
Get Extension	Returns the extension associated with this device.
Get Phone	Returns the reference to the phone object for this Device.
Get Service Provider	Returns an instance of a Service Provider object.
Get Device ID	Gets the device identifier that represents the device described by its extension number and the Communication Manager upon which it resides
Release Device ID	Sends a ReleaseDeviceId XML message to the server.

Phone Class

The Phone Class provides physical device control. The control allows an application to manipulate and monitor the physical aspects of a device, which includes buttons, lamps, the display, and the ringer. The services simulate manual action on a device as well as provide the ability to request status of physical elements. The events provide notification of changes to the physical elements of the device.

All services that operate on a particular device use a device identifier to specify the device. This class will also provide a device identifier for a given extension on Communication Manager.

The Phone Class will also provide the ability to gain **Main**, **Dependent** or **Independent** control over a device and to specify the desired media mode for that device.

Main, **Dependent** and **Independent** control are described in the "Device and media control" section of the "Capabilities of the API" chapter of the *Avaya MultiVantage™ Application Enablement Services Overview*. For a list of device types that can be controlled with this API and for further distinction between the registration modes, see the "Controllable telephone types" section of the "Capabilities of the API" chapter of the *Avaya MultiVantage™ Application Enablement Services Overview*.

The desired media parameters are also specified using the Phone Class. The options for the Media parameters are described in the "Media control modes" section of the "Capabilities of the API" chapter of the *Avaya MultiVantage™ Application Enablement Services Overview*.

To learn how to use the Phone Class, see [The Dashboard tool](#) on page 37.

The Phone Class provides the following asynchronous operations:

Table 4: Phone Class Requests 1 of 2

Method Name	Description
Get Device	Returns the device that this phone object is in.
Get Media Mode	Returns the current media mode that the phone is in.
Get Media	Returns the media object which will perform all media operations on this phone.
Get Registered	Returns true if the phone is currently in the registered state.
Get Button Info	Gets the button information for either a specified button or all buttons on a device, including the button identifier, button function, associated extension (if applicable), and associated lamp identifier (if applicable)
Get Display	Gets a snapshot of the contents of the physical device's display
Get Hookswitch Status	Gets the hookswitch status of a specified device, either on hook or off hook
Get Lamp Mode	Gets the lamp mode status for either a specified button or all buttons on a device, including how the lamp is lit (flutter, off, steady, etc.), color and associated button
Get Message Waiting Indicator	Gets the message waiting status at a specified device, either on or off
Get Registration State	Get the current registration state for the phone.
Get Ringer Status	Gets the ringer status of the ringer associated with a device, including ring mode (ringing/not ringing) and the ring pattern such as normal ring or priority ring
Press Button	Simulates the depression of a specified button on a device
1 of 2	

Table 4: Phone Class Requests 2 of 2

Method Name	Description
Redirect Media	Instruct Communication Manager to direct the media to a different IP Address
Register Terminal	Registers a specific device with Communication Manager in order to gain main, dependent or independent control over the phone or extension and specifies the desired media mode. Applications can specify whether media encryption is desired while registering.
Set Hookswitch Status	Sets the hookswitch status of a specified device to either onhook or offhook
Start Monitor	Informs the API of what type of phone events the application wishes to be notified of. Phone events are events like lamp updates, display updates, ringer updates, etc
Stop Monitor	Cancels a previously initiated Start Monitor request
Unregister Terminal	Unregisters the specified device from Communication Manager in order to give up control of the device
2 of 2	

The Phone Class raises the following events:

Table 5: Phone Class Events

Event Name	Description
Display Updated	Occurs if the contents of a device's display has changed
Hookswitch Updated	Occurs if Communication Manager has changed the device's hookswitch status
Lamp Updated	Occurs if the lamp mode status of a particular lamp has changed
Ringer Status Updated	Occurs if the ringer attribute associated with a device has changed status
Unregister Terminal	Occurs if the specified device has been unregistered from Communication Manager

Media Class

The Media Class will provide an interface to allow media processing to be performed including play, record and tone collection.

This class will allow an application to record voice stream data coming into a device and to play messages to the device's outgoing voice stream.

This class will also allow your application to collect DTMF tones coming into a device, store them in a buffer, and report the tones based on application-specified retrieval criteria. The retrieval criteria can be one or more of the following:

- The specified number of tones has been detected
- The specified tone has been detected
- The specified amount of time has passed

If multiple criteria are specified, then the first condition that occurs terminates the retrieval and reports the string of DTMF tones collected. Both in-band and out-of-band tone collection are supported. Out-of-band tone collection is recommended.

When tones are retrieved and reported to the application, they are removed from the buffer. If the buffer fills up, the oldest tones are overwritten with the new detected tones.

The Tone Detection Events notifies an application whenever a DTMF tone has been detected coming into a device. Both in-band and out-of-band tone detection is supported. Out-of-band tone detection is recommended.

To learn how to use the Media Class, see [The Dashboard tool](#) on page 37 and [Recording and playing voice media](#) on page 74.

The Media Class provides the following asynchronous operations:

Table 6: Media Class Requests 1 of 2

Method Name	Description
Delete Message	Deletes a specified file from the connector server
Flush tone Collection Buffer	Reports the DTMF tones received since the last time the buffer was flushed and flushes the buffer
Get Device	Returns the parent Device object that the media object is a part of.
Resume Playing	Resume only playing, not recording
Resume Recording	Resumes only recording, not playing.
1 of 2	

Table 6: Media Class Requests 2 of 2

Method Name	Description
Start Monitor	Informs the API of what type of media events the application wishes to be notified of. Media events are events like media started, media stopped, recording starting/playing, recording stopped, tone detected, etc
Stop Monitor	Cancels a previously initiated Monitor Start request
Start Playing	Sends a Start Playing XML message to the server.
Stop Playing	Stops only the player, not the recorder.
Suspend Playing	Suspends only the player, not the recorder.
Start Recording	Sends a Start Recording XML message to the server.
Stop Recording	Sends a Stop Recording XML message to the server.
Suspend Recording	Suspends only the recorder, not the player.
Start Dubbing	Starts replacing an existing recording session with the specified file
Stop Dubbing	Stops replacement of an existing recording session
Set Tone Retrieval Criteria	Specifies the retrieval criteria
Start Tone Collection	Used to start collecting DTMF tones sent to a device and specifies the termination criteria.
Stop Tone Collection	Used to stop collecting DTMF tones sent to a device and report the tones that have been buffered. This flushes the buffer.

2 of 2

The Media Class raises the following events:

Table 7: Media Class Events

Event Name	Description
Playing	Indicates that a message is being played
Playing Suspended	Indicates that a message is suspended in play
Playing Stopped	Indicates that a play (or record) operation of a message on a device has been stopped or has completed.
Recording	Indicates that a message is being recorded
Recording Suspended	Indicates that a message is suspended during recording
Tones Retrieved	Occurs when DTMF tones are retrieved from the buffer. This event reports the retrieved tones to the application.
Tone Detected	Occurs when a DTMF tone has been sent to the device
Media Started	Indicates when the far-end RTP parameters have changed and an RTP session has been established. Also provides the media encryption keys if media encryption is enabled for the device.
Media Stopped	Indicates when the far-end RTP parameters have changed to null and the RTP session has been disconnected

Call Information Link Class

The Call Information Link Class will provide an interface to allow call information and Call Information Link (DAPI link) monitoring to occur.

This class allows applications to get detailed call information and to determine the status of the call information link. The call information link must be operational to get the call information. The call information link is one of the communication links between Communication Manager and the connector server.

The Call Information Link Class provides the following asynchronous operations:

Table 8: Call Information Link Class Requests

Method Name	Description
Get Call Information	Used to get detailed call information.
Get Link Status	Used to get the status of the call information link to the connector server.
Start Monitor	Informs the API of what type of call information events the application wishes to be notified of. There are two Call information events: link-up and link-down.
Stop Monitor	Cancels a previously initiated Monitor Start request

The Call Information Link Class raises the following events:

Table 9: Call Information Link Class Events

Event Name	Description
Link Up	Occurs when a link has come up and is now active. Occurs the first time the link is brought up, as well as every time the link is brought up after being down.
Link Down	Occurs when a link has gone down and is now inactive. Occurs when it is determined that Communication Manager is not responding or Communication Manager and Device, Media and Call Control API are out of sync. Response will indicate which link is down and whether the connector server will attempt to reconnect automatically.

Third Party Call Controller Class

The Third Party Call Controller Class provides an interface to allow basic 3rd Party call Control to be performed.

This class will provide single step conferencing capabilities to allow an application to add a device into an existing call.

This class will allow an application to obtain 3rd party information about the devices participating in a specified call. The information returned includes device identifiers, their connections in the call, and local connection states of the devices in the call as well as call related information.

This class provides information about calls associated with a given device. The information provided identifies each call the device is participating in and the local connection state of the device in that call.

The Call Control Services utilizes the TSAPI Service on the AE Services server. The use of the Call Control Services requires the setup of the connection and cti-link between the AE Services server and Communication Manager as well as a basic TSAPI license.

The use of the Snapshot Services requires the setup of the connection and cti-link between the AE Services server and Communication Manager as well as a basic TSAPI license

The Third Party Call Controller Class provides the following asynchronous operations:

Table 10: Third Party Call Controller Class Requests 1 of 3

Method Name	Description
Alternate Call	Performs a third party Alternate Call operation (put active call on hold and unhold held call).
Answer Call	Performs a third party Answer Call operation (answers an incoming call).
Clear Call	Perform a third party Clear Call operation (drops all parties on a call).
Clear Connection	Performs a third party Clear Call operation (drops all parties on a connection).
Conference Call	Performs a third party Conference Call operation (merges the held call into the active call).
Consultation Call	Performs a third party Consultation Call operation (puts the active call on hold and calls another device).
Deflect Call	Performs a third party Deflect Call operation (directs an incoming call to another destination).
Generate Digits	Performs a third party Generate Digits operation (inject touch tone digits into an active call).
1 of 3	

Table 10: Third Party Call Controller Class Requests 2 of 3

Method Name	Description
Get Do Not Disturb	Performs a third party Get Do Not Disturb operation (determines if Do Not Disturb, AKA Send All Calls, is active).
Get Third Party Device ID	Gets a third party device identifier information for the selected DeviceId. This information is necessary in order to perform other third party call functions.
Get Forwarding	Performs a third party Get Forwarding operation (determines if call forwarding is active, if so the call forwarding destination is provided).
Hold Call	Performs a third party Hold Call operation (puts an active call on hold).
Make Call	Performs a third party Make Call operation (originates a call for the specified third party Device ID).
Reconnect Call	Performs a third party Reconnect Call operation (drops the active call and unhold a call that is held).
Retrieve Call	Performs a third party Retrieve Call operation (takes a call off of hold and makes it active).
Set Display	Performs a third party Set Display operation (overwrites the current display of the specified Device ID. Note:Special administration on DMCC must be done to support this capability).
Set Do Not Disturb	Performs a third party Set Do Not Disturb operation (allows you to enable/disable Do Not Disturb AKA Send All Calls).
Set Forwarding	Performs a third party Set Forwarding operation (allows you to enable/disable Call Forwarding for the specified Device ID).
Single Step Conference Call	Adds a device to an existing call.
Single Step Transfer Call	Performs a third party Single Step Transfer Call operation (transfers an active call to another device).
2 of 3	

Table 10: Third Party Call Controller Class Requests 3 of 3

Method Name	Description
Snapshot Call	Provides information about the devices participating in a specified call.
Snapshot Device	Provides information on the status of calls at a specific device.
Start Monitor	Informs the API of what type of Third Party Call Control events the application wishes to be notified of. There are two Call information events: link-up and link-down.
Stop Monitor	Turn a monitor off.
Transfer Call	Performs a third party Transfer Call operation (transfers the call on hold to the active call and drops the party doing the transfer).
3 of 3	

The Third Party Call Controller Class raises the following events:

Table 11: Third Party Call Control Class Events

Event Name	Description
Conferenced	Event registration for Conferenced event.
Connection Cleared	Event registration for Connection Cleared event.
Delivered	Event registration for Delivered event.
Diverted	Event registration for Diverted event.
Established	Event registration for Established event.
Failed	Event registration for Failed event.
Held	Event registration for Held event.
Originated	Event registration for Originated event.
Retrieved	Event registration for Retrieved event.
Transferred	Event registration for Transferred event.
1 of 2	

Table 11: Third Party Call Control Class Events (continued)

Event Name	Description
Do No Disturb	Event registration for Do Not Disturb event.
Forwarding	Event registration for Forwarding event.
2 of 2	

XML Processor Class

The XML Processor Class provides 2 key features:

- It provides a method to allow a client to send their own XML messages to the server and a response handle to then process the XML response. It also provides an event handler to process any unhandled events coming from the server. This allows an application to take advantage of new server features without having to wait for a client with the support for that new feature.
- It provides the ability for the application to register two event handlers. One handler is called when any request is sent to the server. The event includes the XML message sent to the server. The 2nd handler is called when any request, exception or event is received from the server. This can be useful for debug purposes or for application developers working at the XML level who want to see what these "raw" message look like.

The XML Processor Class provides the following asynchronous operations:

Table 12: XML Processor Class Requests

Method Name	Description
Inject XML	Send XML directly to server. No verification on XML is done before it is sent.

The XML Processor Class raises the following events:

Table 13: XML Processor Class Events

Event Name	Description
Unhandled XML Message	Event registration for getting notified when XML messages are not processed by the API.
XML Data Inject XML Response	Event registration for responses to InjectXml request.
XML Message Sent	Event registration for getting notified when XML messages are sent to the server.
XML Message Received	Event registration for getting notified when XML messages have been received from the server.

Constant Classes

Constant classes are provided for the following:

- Button Function Constants
- Button ID Constants
- Lamp Mode Constants
- Ringer Pattern Constants
- Media Constants
- Registration Constants
- API Version Constants
- Service Name Constants

Please see [Appendix C: Constant Values](#) for more information on these constant classes.

Chapter 2: Getting started

This section describes what you need to do and what you need to know before you begin programming to this API, including:

- [Setting up the development environment](#) on page 29
- [Understanding basic CSTA concepts](#) on page 30
- [Events](#) on page 32
- [Signaling Encryption](#) on page 33
- [Media Encryption](#) on page 33
- [Accessing the client API reference documentation](#) on page 34
- [Learning from sample code](#) on page 34

Setting up the development environment

Applications will be developed using the Microsoft Visual Studio tool in order to build your application. To develop your .NET application using Visual Studio, you must make a reference to the ServiceProvider.dll file.

In Visual Studio:

- right click on **References** in the Solution Explorer window
- select **Add Reference**
- select the **Browse** tab
- browse to the folder where the ServiceProvider.dll file resides
- select the file and hit **OK**

Downloading the Application Enablement Services Device, Media and Call Control .NET API SDK

To download the Application Enablement Services Device, Media and Call Control .NET API SDK from the Avaya DevConnect website:

1. Go to www.avaya.com/devconnect.
2. Select **Member Login**.

3. Log in with your email address and password.
4. Download the SDK (dmcc-dotnet-sdk-x.zip) from the DevConnect website by navigating to the Application Enablement Services page and selecting the appropriate SDK from the Technical Resources section.

Note:

The Application Enablement Services page can be located through the Avaya Product Information/Documentation/SDKs link under the left-hand Do Your Research menu options.

The download location defaults to the desktop, but it does not matter where you download the files in your directory system. The SDK file is:

dmcc-dotnet-sdk-x.zip

where x is the load number.

5. Expand the SDK ZIP file using any application or tool that recognizes the ZIP file format. All of the SDK files are placed into a directory named dmcc-dotnet-sdk.

Setting up your test environment

Before running an application you will need to have an AE Services server machine setup. For instructions see the appropriate *Avaya MultiVantage™ Application Enablement Services Installation and Upgrade Guide* for the offer you have purchased (bundled server or software only).

Understanding basic CSTA concepts

CSTA stands for Computer-Supported Telecommunications Applications. It is a standard produced by ECMA, an international standards body (<http://www.ecma-international.org>). CSTA provides a standard for Computer-Telephony Integration (CTI). When fully implemented, CSTA allows an application to:

- monitor calls on dial strings, lines or trunks
- modify the behavior of calls
- make a call between two parties

The Avaya Application Enablement Services Device, Media and Call Control API implements a subset of CSTA. The API supports monitoring and making calls at the physical device level. Applications using this API have first party device control and media control and third-party call control.

The following sections describe what you need to know about the CSTA concepts of:

- [Devices](#)
- [Physical elements](#) and [Logical elements](#)
- [Calls](#)
- [Events](#)
- [Negative acknowledgements](#)

Devices

In the context of this API, a device refers to a software instantiation of a phone or dial string that is registered on Communication Manager. Such a device is also referred to as a softphone. A device has physical and logical elements.

Physical elements

The physical element of a device encompasses the set of attributes, features, and services that have any association with physical components of the device that make up the user interface of the device. Physical elements can be manipulated, such as pushing buttons or going offhook, or they can be observed, such as observing the ringer or whether a lamp is lit. The physical elements of a device include:

- Buttons
- Hookswitch
- Display
- Lamps
- Message waiting indicator
- Ringer

This API supports all of these physical elements.

Logical elements

A logical element is the part of a device that is used to manage and interact with calls at a device. This element represents the media stream channels and associated call handling facilities that are used by the device when involved in a call. The logical elements that this API supports are:

- DTMF tones coming into the device
- Media stream coming into and out of the device
- Do Not Disturb

- Call Forwarding

Calls

Calls are made from and received by a device. CSTA performs most telephony services against a connection that identifies a particular call.

Events

Events are asynchronous occurrences on a device that an application can choose to listen for and respond to. Examples of events are the `DisplayUpdatedEvent`, which indicates that the device's display has changed, or `TerminalUnregisteredEvent`, which indicates that the device has been unregistered by Communication Manager.

An application indicates its desire to be notified of events by using the `Monitor.Start` method on the various classes.

Every time you request the API to do something (for example, push a button) and that request results in a message being sent to the server, you will immediately get an *InvokeID* back. The *InvokeID* is the message number that is sent to the server and may be useful in debugging over on the server.

When the request has been acted upon, you will get an asynchronous message from the .NET API informing you of the results of the request. In order to get this message your application must have registered to get it. You will see how to do this under [Learning from sample code](#) on page 34.

Note:

Once you receive an event, release the event thread immediately and continue to do event processing on a different thread.

Negative acknowledgements

If an error is encountered, the details of the error will be in the "error" string attribute in the response event.

Signaling Encryption

AE Services offers the ability to encrypt the signaling channel between the AE Services server and Communication Manager. However, the option to encrypt this link is not under the direct control of your application. Instead, it is an option that is provisioned on Communication Manager, via the "set network-region" page. See the "Setting up a network region for Device and Media Control" subsection of the "Administering Communication Manager for AE Services" chapter of the *Avaya MultiVantage™ Application Enablement Services Administration and Maintenance Guide*.

The encryption of the signaling link is automatically negotiated between the AE Services server and Communication Manager during the device registration, and the resultant encryption used is dependent on the options set for the network region. The following encryption types are supported:

- Challenge
- Pin-Eke

When registering the device, the negotiated signaling encryption type is included as a parameter in the RegisterTerminalResponse message.

Media Encryption

You have the ability to encrypt the RTP (media) stream between the application and the other endpoint of the call.

The option to encrypt the RTP stream is provisioned on the Communication Manager, this time via the "set codec" page. See the "Creating the Device and Media Control codec set" subsection of the "Administering Communication Manager for AE Services" chapter of the *Avaya MultiVantage™ Application Enablement Services Administration and Maintenance Guide*. However, in this case, the application does have some (albeit limited) control over the type of encryption used. For AE Services R3.1, the media encryption will be one of the following types:

- Advanced Encryption Scheme (AES)
- no media encryption

When registering the device, the application may specify which media encryption schemes that it supports. For more details see [Choosing the media encryption](#) on page 72.

This list of application-supported media encryption schemes is matched to the list of encryption schemes provisioned on Communication Manager's "set codec" page for the device's codec set. Communication Manager will pick an encryption scheme that is common to both lists, if possible.

Note that the encryption scheme chosen by Communication Manager will be indicated in the `MediaStartEvent` message. See [MediaStartEvent Handling](#) on page 84.

Accessing the client API reference documentation

You may need to reference the *.NET Programmer's Reference* provided with this API. It is available on the Avaya Support site (www.avaya.com/support) as both a viewable HTML document and a downloadable zip file.

This html documentation will also ship with the SDK. This documentation describes all of the interfaces and their parameters. In addition, Visual Studio will provide help with the parameters as you are writing the code.

Using the Device, Media and Call Control Dashboard

In addition, the SDK ships with a "Dashboard" application which documents all of the capabilities of the API and provides the ability to get detailed help about every interface. This Dashboard application can greatly reduce the learning curve. The Dashboard tool allows you to easily try out all of the capabilities of the .NET SDK without having to write any code.

For information on getting started with using the Dashboard tool, see [The Dashboard tool](#) on page 37.

Learning from sample code

The SDK ships with many code snippets which can be imported directly into Visual Studio. These code snippets are available to provide sample code for virtually every interface into the SDK and also provide an example of handling all of the events.

To import the code snippets into Visual Studio:

- click on **Tools**
- click on **Code Snippets Manager**
- select you language (Visual C# or Visual basic)
- select **Add**
- Browse to the Method and Events Folder (provided by the SDK)

- select **Open**
- select **Ok**

These code snippets are now available for use in your applications. To utilize these in your code, right click in your source code and select **Insert Snippet**. You may then specify which snippet to insert into your code.

Another key learning tool is the set of sample applications that are provided with this API. These sample applications are located in the following directory:

cmapi-net-sdk/examples/src/samples

Table 14: Sample code

Application or Class	Language written in	File path names under cmapi-net-sdk/examples/src/samples	Description
Simple Record	C#	simpleRecord	This application provides a simple recording capability. It allows you to record conversation for a given extension. You may record the calls either on the server or on your PC.
Simple RecordVB	Visual Basic	simpleRecordVB	This application is identical to Simple Record except it was written in Visual Basic.
Soft phone in a browser	HTML and JScript	Sample web page is in: jScript/softphone The html document name is: leSoftphone.html	This is an HTML page which demonstrates a simple softphone which runs in a browser. The softphone can run in telecommuter mode or in no media mode.
Dashboard in a Browser	HTML and JScript	JScript\dashboard	This is an HTML page which provides basic dashboard capabilities.

Chapter 3: Writing a client application

This chapter describes how to write an application using the Application Enablement Services.

You will need to understand:

- what features of Device, Media and Call Control you will need for your application
- what capabilities Device, Media and Call Control provides.

The .NET SDK ships with a tool that will assist you in both aspects.

The Dashboard tool

The name of this tool is **Dashboard.exe** and it is located in the `dmcc-dotnet-sdk` folder. The Dashboard tool allows you to easily try out all the capabilities of the .NET SDK without having to write any code. Once you have determined what you need out of Device, Media and Call Control it will not be difficult to implement in the code (assuming you have a working knowledge of Visual Studio).

The information in this chapter assumes a developer has zero knowledge of Device, Media and Call Control.

We will outline the steps necessary to write a simple application. In general, most, if not all, applications will need to do the following:

- Start a session with the AE Services server
- Get one or more devices that you are interested in monitoring.
- Register those Devices to the Communication Manager.
- Listen for events that you are interested in (e.g., lamp updates).
- Process those events.

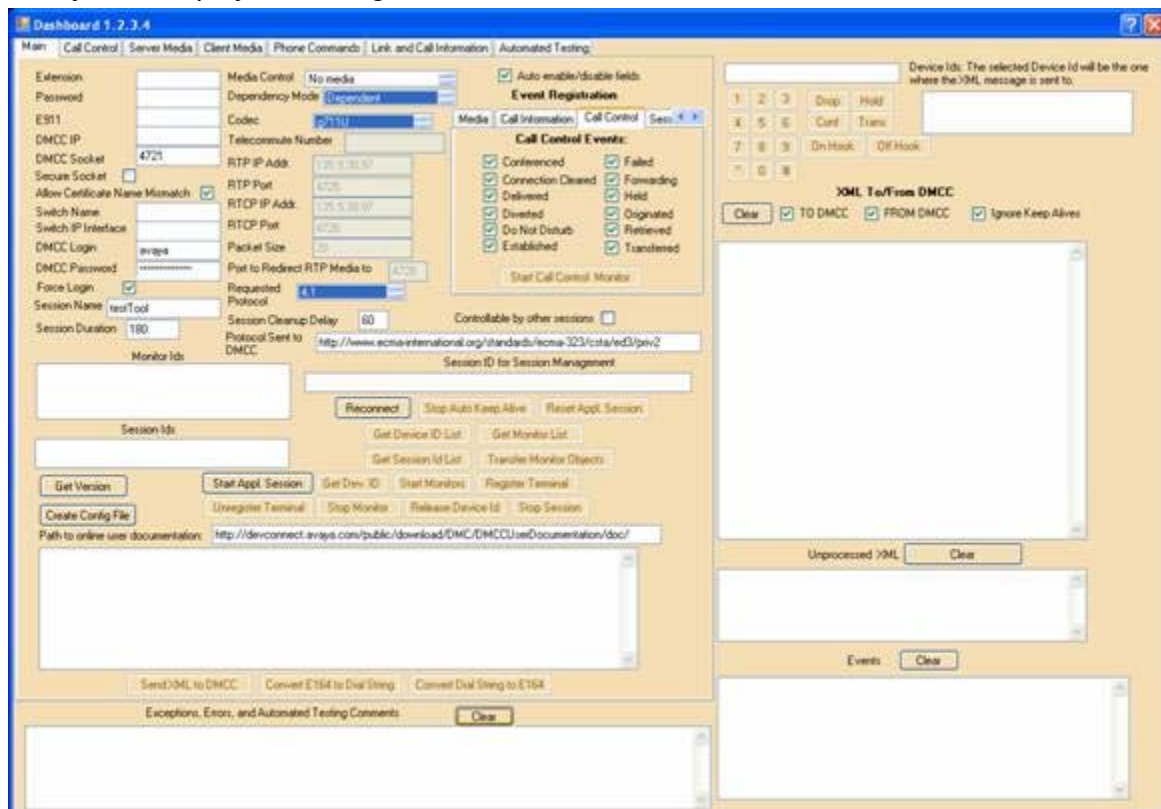
For our sample application, we will assume the developer wants to do the following:

- monitor all incoming calls for extension 5382834
- log the duration of the calls

The first thing you must do is start up the Dashboard application.

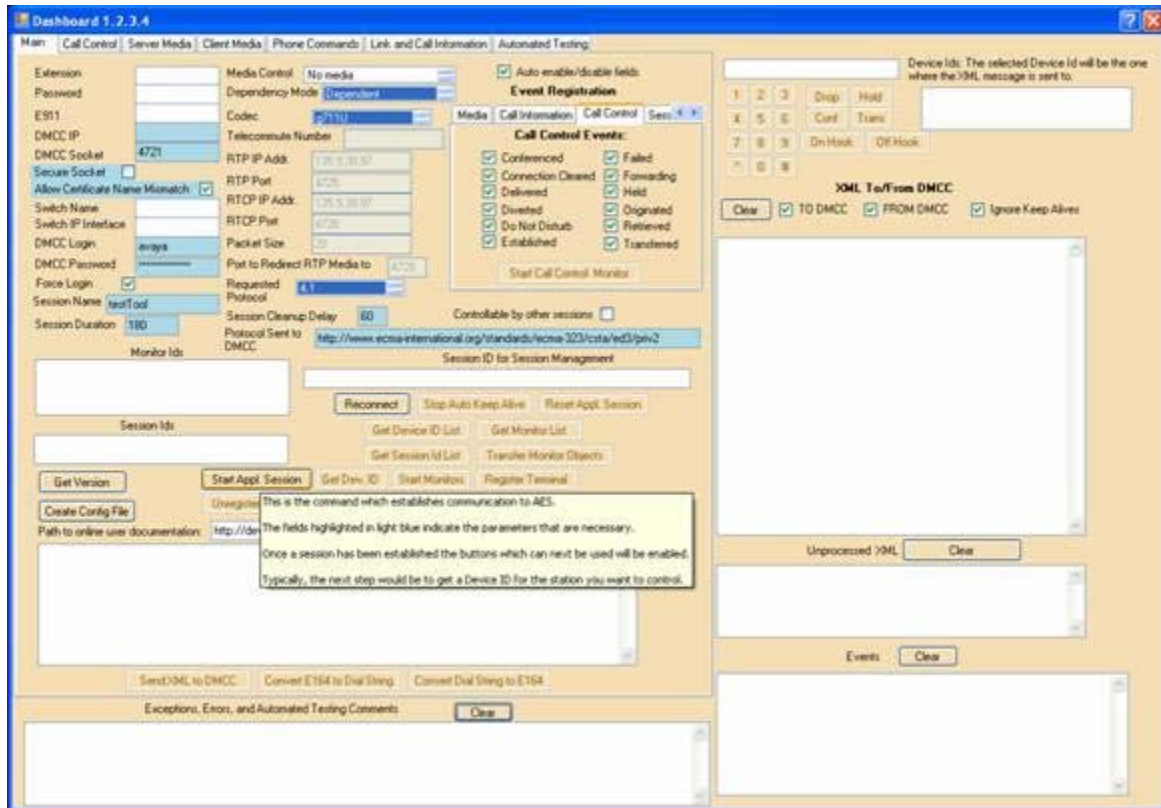
Writing a client application

The system displays following:



Notice that most of the buttons are inactive. The only buttons that are active are the ones that are valid given the current state of the Dashboard. In this case, the only button that is active is the Start Application Session button.

If you hold the mouse over the Start Application Session button several of the input fields will turn a light blue color.



The input fields that turned light blue in this case are the parameters that are required by the Start Application Session method.

Note:

In addition, we also see a tool tip that comes up when the mouse is held over a button or an input field. The tool tip provides additional information about that button or the input field.

We now know exactly what information is needed to start an Application Session. The required information is:

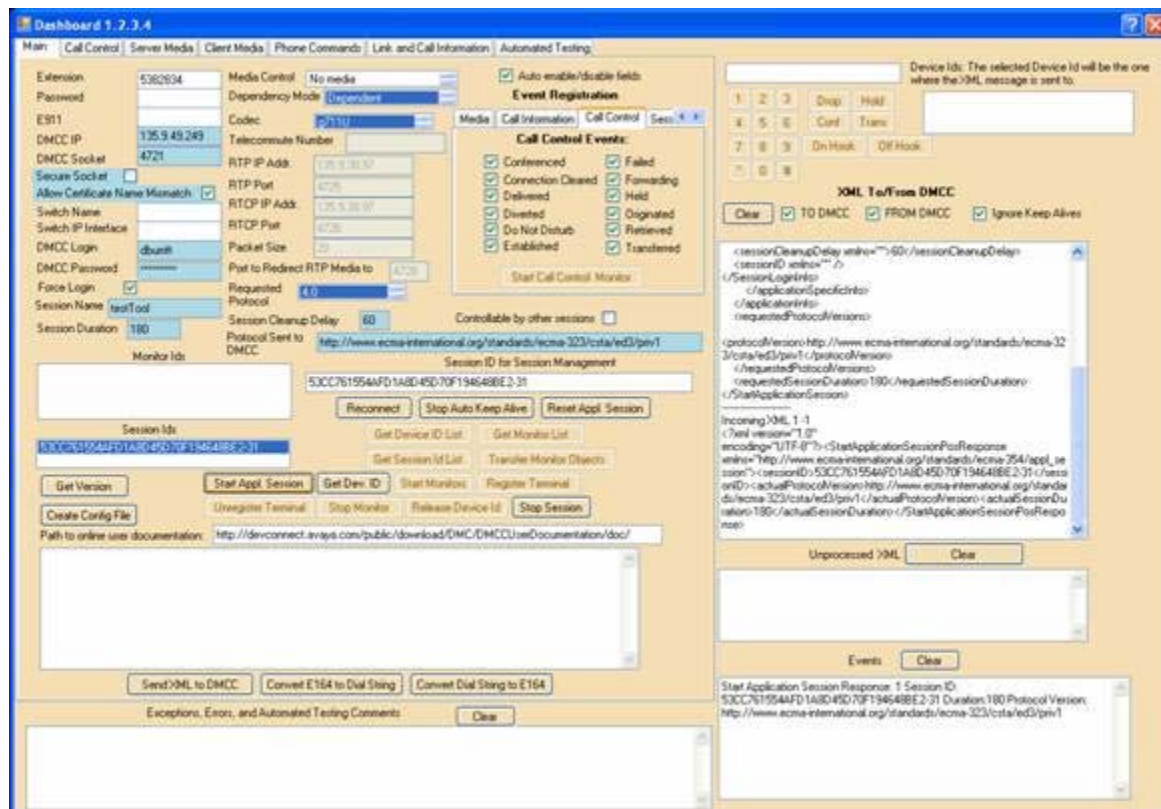
- The IP Address (or DNS name) to the AE Services server
- The port that you want to connect to, typically 4721 or 4722 (4722 is the default port for encrypted information)
- You need to indicate if this port is a secure socket (usually checked if port 4722 is used)
- You need to indicate if you want to allow a certificate name mismatch (only necessary if a secure socket is used)
- You need to have a valid login name and password on the DMCC server
- You need to specify a name for the session

Writing a client application

- You need to specify how long the session (in seconds) should last if no keep alives are returned. (180 seconds is a reasonable number)
- You need to specify (in seconds) how long the session is to be kept active in the server AFTER the server tears the connection down due to no keep alives being sent. This will allow you to attempt to reconnect the session if you are notified that the session was terminated by the server. (60 is a reasonable value for this field).
- You need to specify what protocol version you want to communicate to the DMCC server on.

Input the information into the Dashboard and push the **Start Application Session** button.

The following screen shot shows the result:

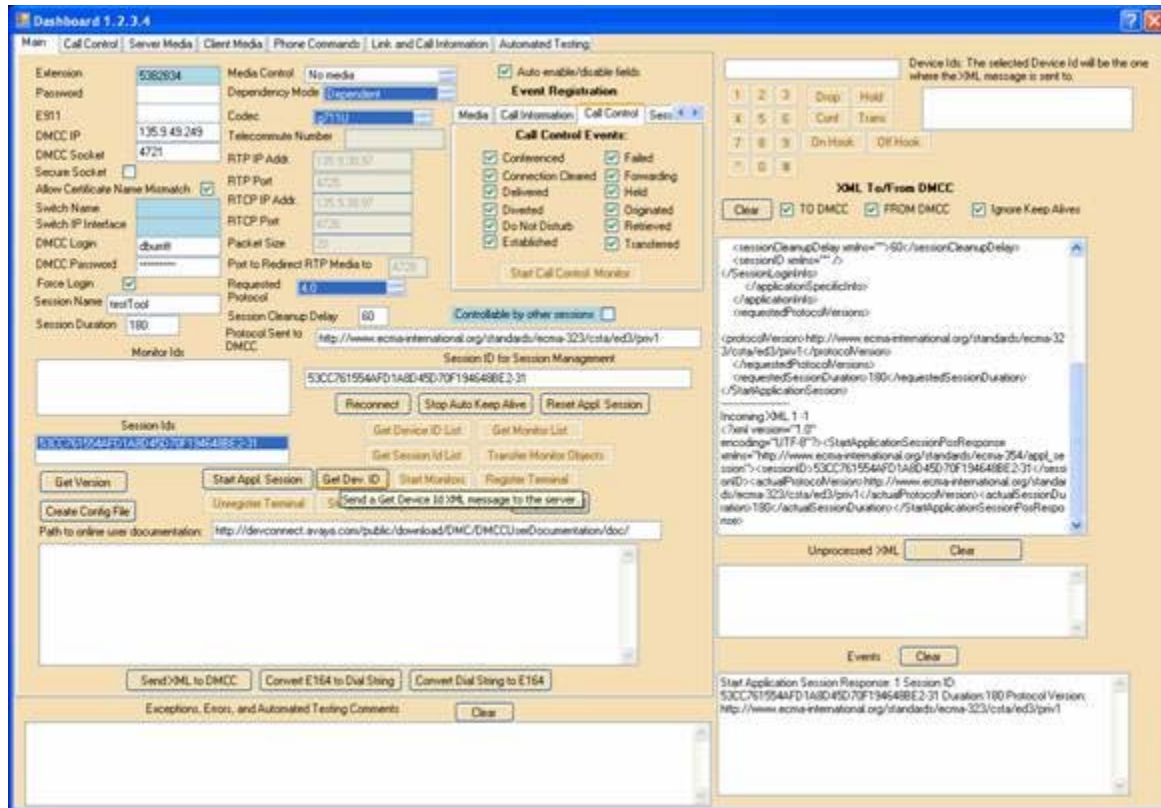


A session has been created and:

- the Session Id is in the Session Id textbox area
- all of the outgoing and incoming XML messages which are sent/received to the AE Services server are displayed
- the results of your request are in the Events box
- If an error has occurred you will see the details in the Events area or possibly in the Exceptions, Errors, and Automated Testing Comments textbox.

You now have a valid session established. The the next thing you need to do is get a Device ID.

Notice that the **Get Dev Id** button is now available. If we hold the mouse over the **Get Dev. Id** button we see what information we need to fill in.



Specifically we need:

- Extension
- Switch Name (or IP Address)
- Do we want other sessions to also control this station

Writing a client application

Fill in the required information and push the Get Dev. Id button:

Dashboard 1.2.3.4

Main | Call Control | Server Media | Client Media | Phone Commands | Link and Call Information | Automated Testing

Extension: 5362934
Password:
ESR1:
DMCC IP: 135.9.43.249
DMCC Socket: 4721
Secure Socket: ☐
Allow Certificate Name Mismatch: ☒
Switch Name: satum
Switch IP Interface:
DMCC Login: dbunill
DMCC Password:
Face Login: ☒
Session Name: testTool
Session Duration: 180
Monitor Ids:
Session Ids:
Get Version:
Create Config File:
Path to online user documentation: <http://devconnect.avaya.com/public/download/DMCC/DMCCUserDocumentation/doc/>

Media Control: No media
Dependency Mode: Unpublished
Codec: G.711
Telecommute Number:
RTP IP Addr: 135.9.43.57
RTP Port: 4726
RTCP IP Addr: 135.9.43.57
RTCP Port: 4726
Packet Size: 20
Port to Redirect RTP Media to:
Requested Protocol: 1.0
Session Cleanup Delay: 60
Controllable by other sessions: ☐
Protocol Sent to DMCC: <http://www.ecma-international.org/standards/ecma-323/csta/ed3/priv1>
Session ID for Session Management: 53CC761554AFD1A8D45D70F194548BE231
Reconnect:
Stop Auto Keep Alive:
Reset Appl. Session:
Get Device ID List:
Get Monitor List:
Get Session Id List:
Transfer Monitor Objects:
Start Appl. Session:
Get Dev. Id:
Start Monitor:
Register Terminal:
Unregister Terminal:
Stop Monitor:
Release Device Id:
Stop Session:
Send XML to DMCC:
Convert E164 to Dial String:
Convert Dial String to E164:
Exceptions, Errors, and Automated Testing Comments:
Clear:

Auto enable/disable fields:
Event Registration:
Call Control Events:
☒ Confirmed
☒ Connection Cleared
☒ Delivered
☒ Diverted
☒ Do Not Disturb
☒ Established
☒ Failed
☒ Forwarding
☒ Held
☒ Originated
☒ Retrieved
☒ Transferred
Start Call Control Monitor

Device Ids: The selected Device Id will be the one where the XML message is sent to:
1 2 3 Drop Hold
4 5 6 Cont Trans
7 8 9 On Hook Off Hook
0
XML To/From DMCC:
Clear
☒ TO DMCC
☒ FROM DMCC
☒ Ignore Keep Alives
Outgoing XML 17:
<?xml version="1.0" encoding="utf-8"?>
<GetDeviceId
xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsi="http://www.w3.org/2001/XMLSchema"
<switchName>atum</switchName>
<extension>5362934</extension>
</GetDeviceId>
Incoming XML 17-1:
<?xml version="1.0" encoding="UTF-8"?><GetDeviceIdResponse
xmlns="http://www.avaya.com/csta"><deviceTypeOrNumber>other</deviceTypeOrNumber>
<deviceId>5362934</deviceId>
</GetDeviceIdResponse>
Unprocessed XML:
Clear
Events:
Clear
Start Application Session Response: 1 Session ID: 53CC761554AFD1A8D45D70F194548BE231 Duration: 180 Protocol Version: <http://www.ecma-international.org/standards/ecma-323/csta/ed3/priv1> Get Device Id Response (Device Phone): 17 Device Id: 5362934 satum: 135.9.43.152.0

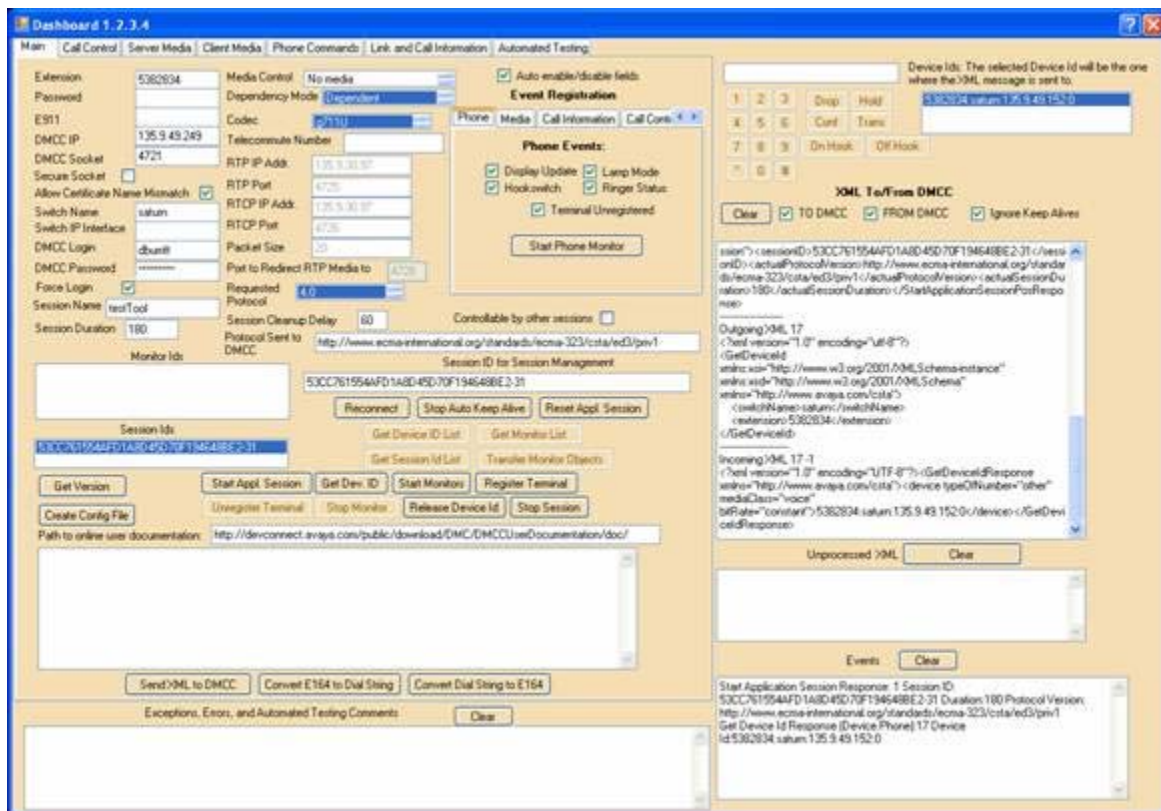
We now have a Device ID and:

- the Device Id is in the Device ID textbox area
- the results are in the Events box

Now that we have a Device ID, the next thing we need to do is detect that a user is on a call. We have decided we want to monitor all phone events to see what is coming in. To accomplish this we need to create a monitor for phone events.

You do this with the **Event Registration** area of the Dashboard. This is where you register for the events you are interested in. The **Event Registration** area is divided into logical tabs.

To monitor phone events, we will select the **Phone** tab.

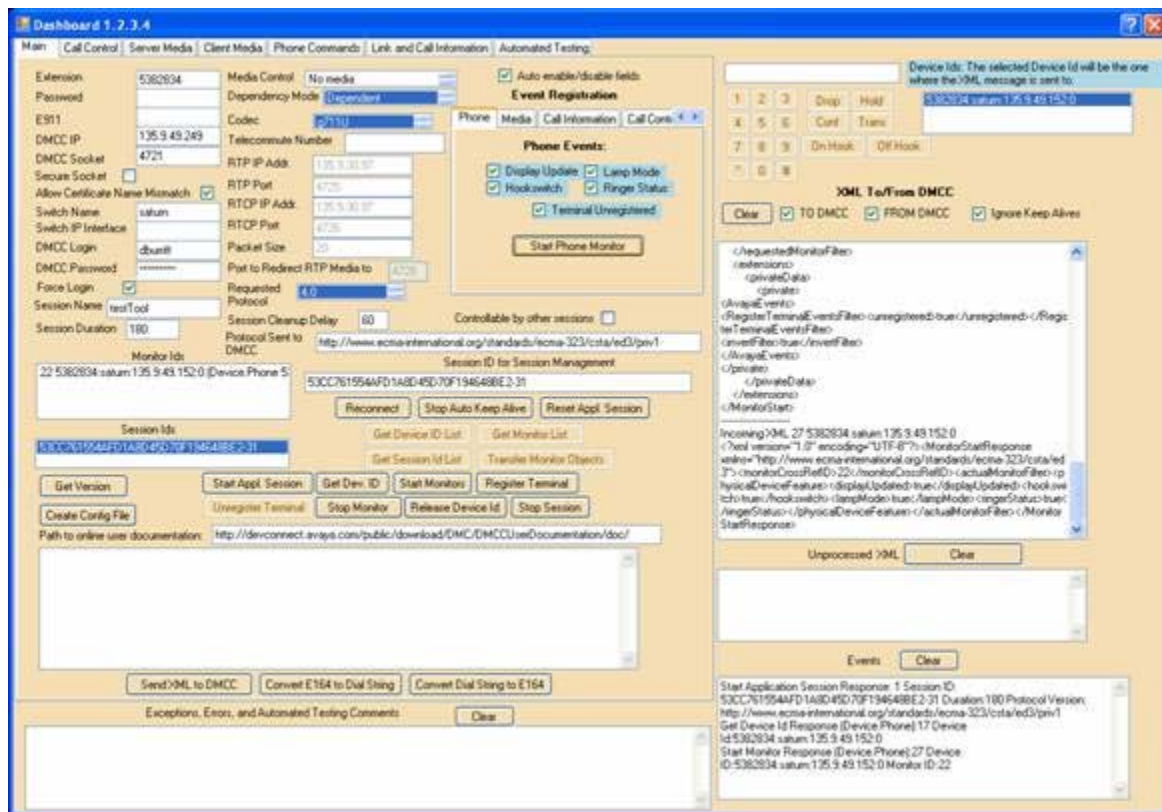


We now see that we can monitor display updates, hookswitch updates, lamp mode updates, ringer status updates, and get notified if the terminal is unregistered.

Since we are not sure which ones we want, we will monitor all phone events.

Writing a client application

So we push the **Start Phone Monitor** button.



We have started a Phone Monitor and:

- the Phone Monitor is in the Monitor IDs area
- the results are in the Events box

Now that we have the monitor started, we need to register the terminal.

Notice that the **Register Terminal** button is now available. If we hold the mouse over the **Register Terminal** button we see what information we need to fill in.

Specifically we need:

- Password
- E911 Information (if any)
- Force Login
- Media Mode
- Dependency Mode

Fill in the required information and push the **Register Terminal** button:

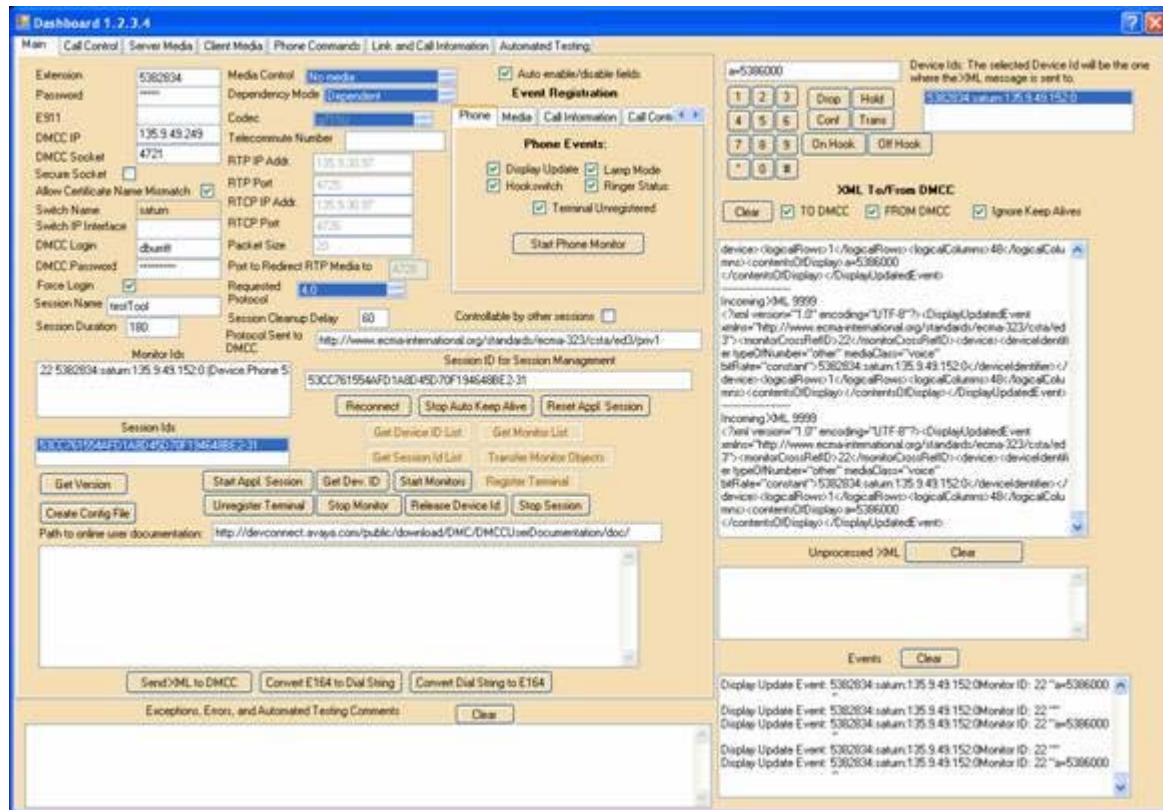
The screenshot shows the 'Dashboard 1.2.3.4' interface. The 'Main' tab is selected, showing various configuration fields. The 'Register Terminal' button is highlighted in the bottom right section. The 'Event Registration' section shows 'Auto enable/disable fields' checked. The 'Phone Events' section shows 'Display Update', 'Lamp Mode', 'Hookswitch', 'Ringer Status', and 'Terminal Unregistered' checked. The 'XML Te/From DMCC' section shows 'TO DMCC' and 'FROM DMCC' checked. The 'Unprocessed XML' section shows a list of XML events, including 'Terminal Registered Response' and 'Display Update Event'.

The device is now registered and

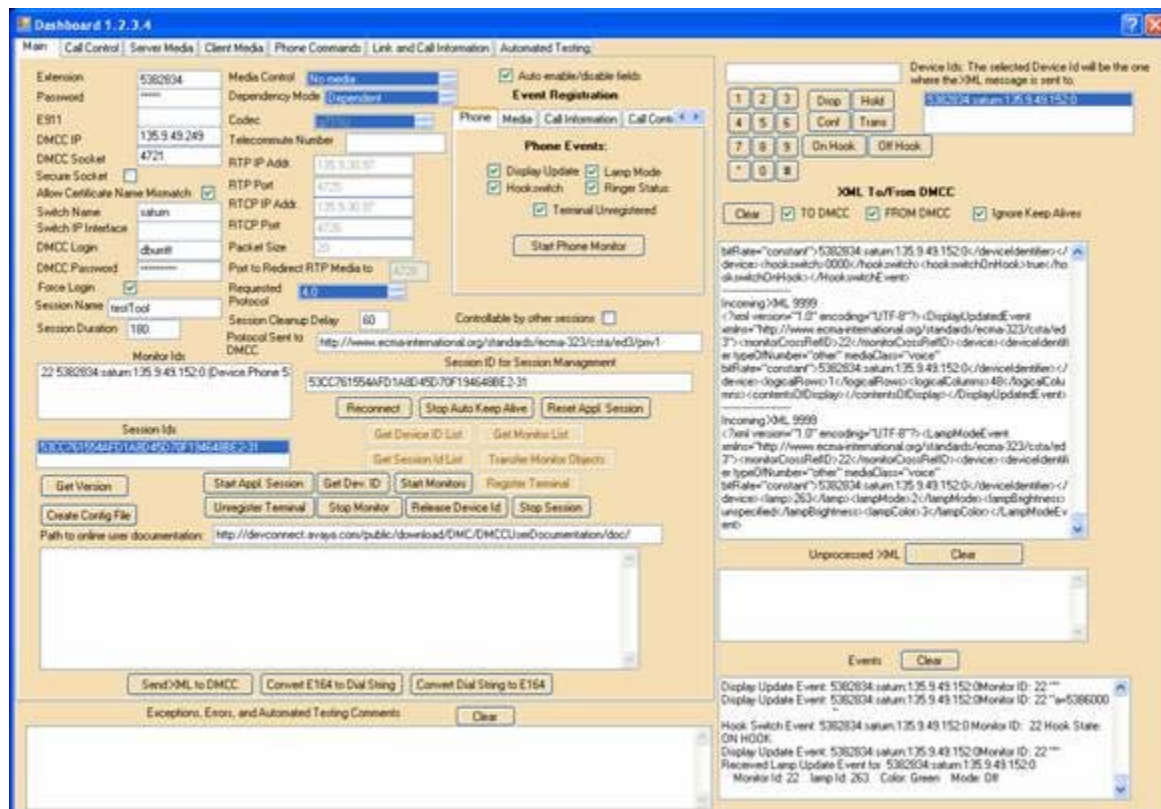
- some display and lamp update are now in the Events area
- the dial pad buttons (0 ñ 9, *, and #) are now active
- the **Drop**, **Hold**, **Conf**, **Trans**, **On Hook**, and **Off Hook** buttons are now active

Writing a client application

So, let's make an outgoing call to 5386000 and see what happens.



Now we will hang up:



In the Events textbox we see a lot of display updates, lamp updates, and hook switch updates.

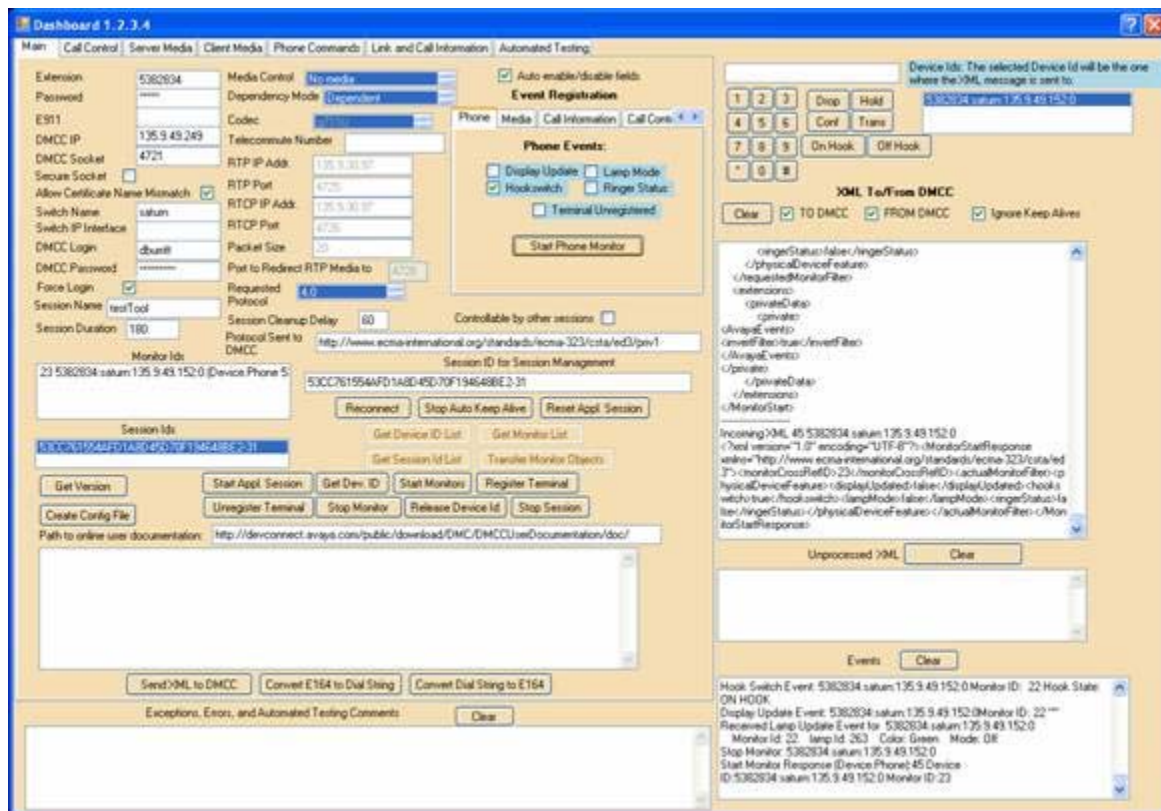
We now make and receive a few calls and observe the Events textbox.

By scrolling the Events textbox area we notice that we are getting an **OFF HOOK** event every time the phone makes or answers a call and an **ON HOOK** event every time a user disconnects a call. We decide, for this example, that monitoring the hookswitch status is how we want to decide if the user is on a call or not.

To do this:

- Stop the existing phone monitor by selecting the monitor in the Monitor Ids textbox and then pushing the **Stop Monitor Button**
- Start another one that just monitor hook switch changes by unchecking all the events in the Phone Events area except for **Hookswitch** and then push the **Start Phone Monitor** button.

Writing a client application



A new monitor was created.

Our steps to this point have been:

- Start an application session to the AE Service server
- Get a Device Id for extension 5382834
- Start a monitor for that Device Id which will only monitor Hookswitch changes
- Register the terminal

Once you have done the above your application should be able to record the call duration.

When your application is done, you must:

- Unregister the terminal
- Stop the monitor
- Release the Device ID
- Stop the Application Session

Expanded help feature

The Dashboard tool offers an expanded help feature. To use this feature

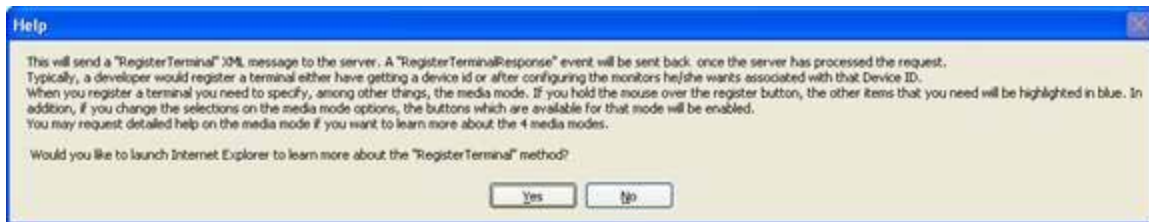
1. click on the ? button (top right)
2. click on any button (for example, the **Register Terminal** button) that you wish to have more detailed help with

The more detailed help will be presented.

Note:

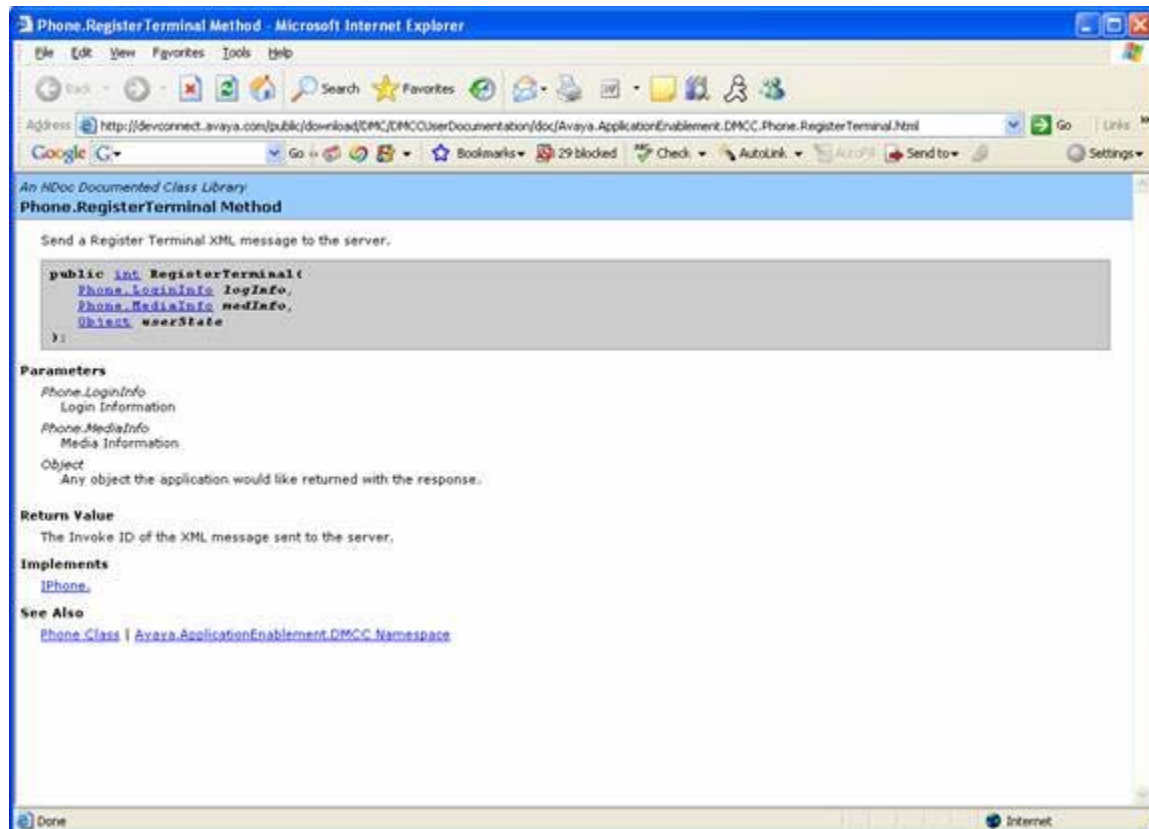
In addition, in most cases you will be prompted to also launch a browser which will take you directly to the documentation on the Avaya DevConnect site for that particular method.

If you do this with the **Register Terminal** button the following dialog will appear:



Select **Yes** to get more information about the **RegisterTerminal** method.

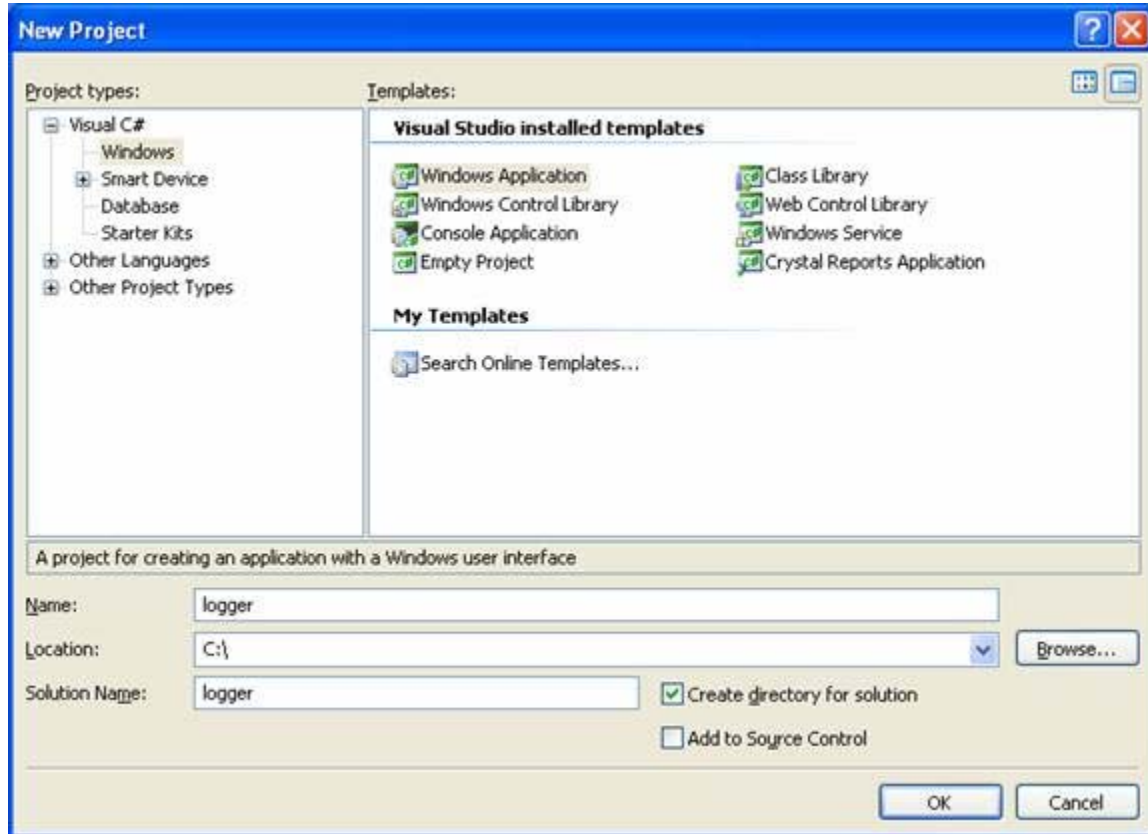
Internet Explorer will bring up the *Programmers Reference* with the information about the **RegisterTerminal** method. From there you can click on the links to learn more about the parameters or other related methods.



Writing Code

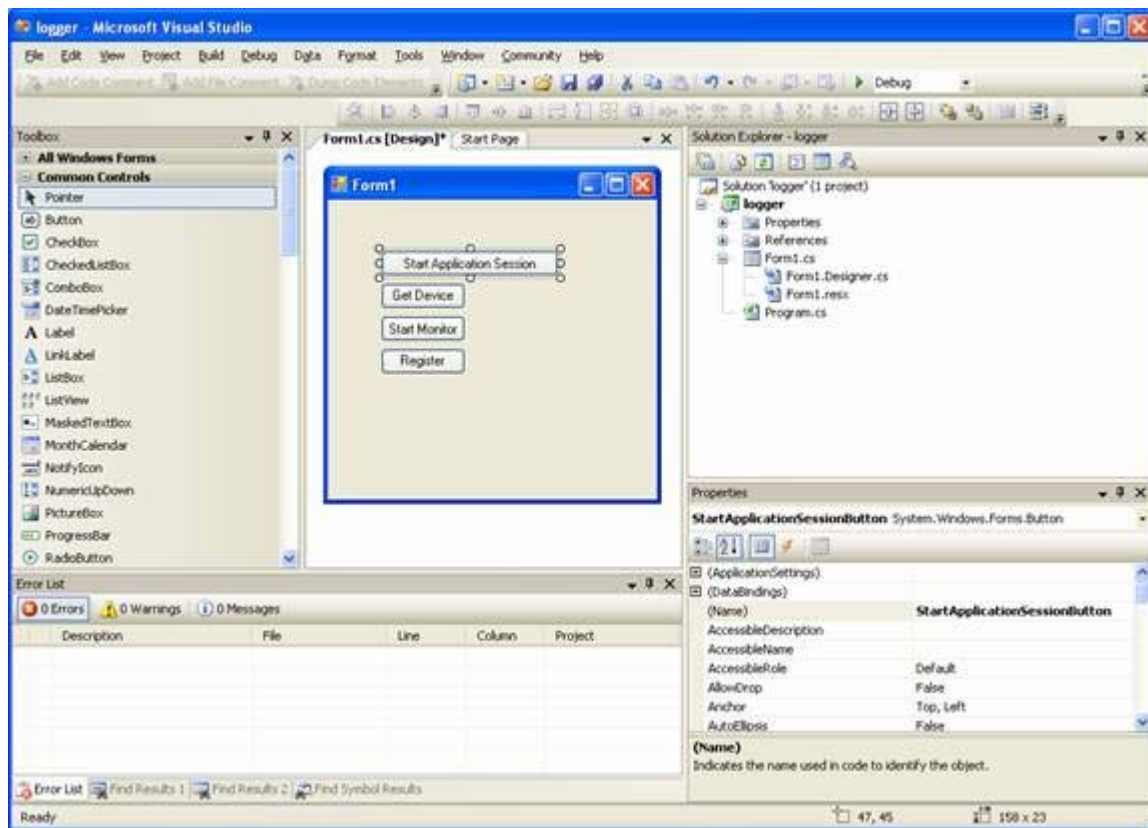
This section covers starting an application session. The other steps are virtually identical.

Start Visual Studio. We will assume you are familiar with Visual Studio so start a new **Windows Application** project and for this example call your application **logger**.



Add 4 buttons to the form and label them:

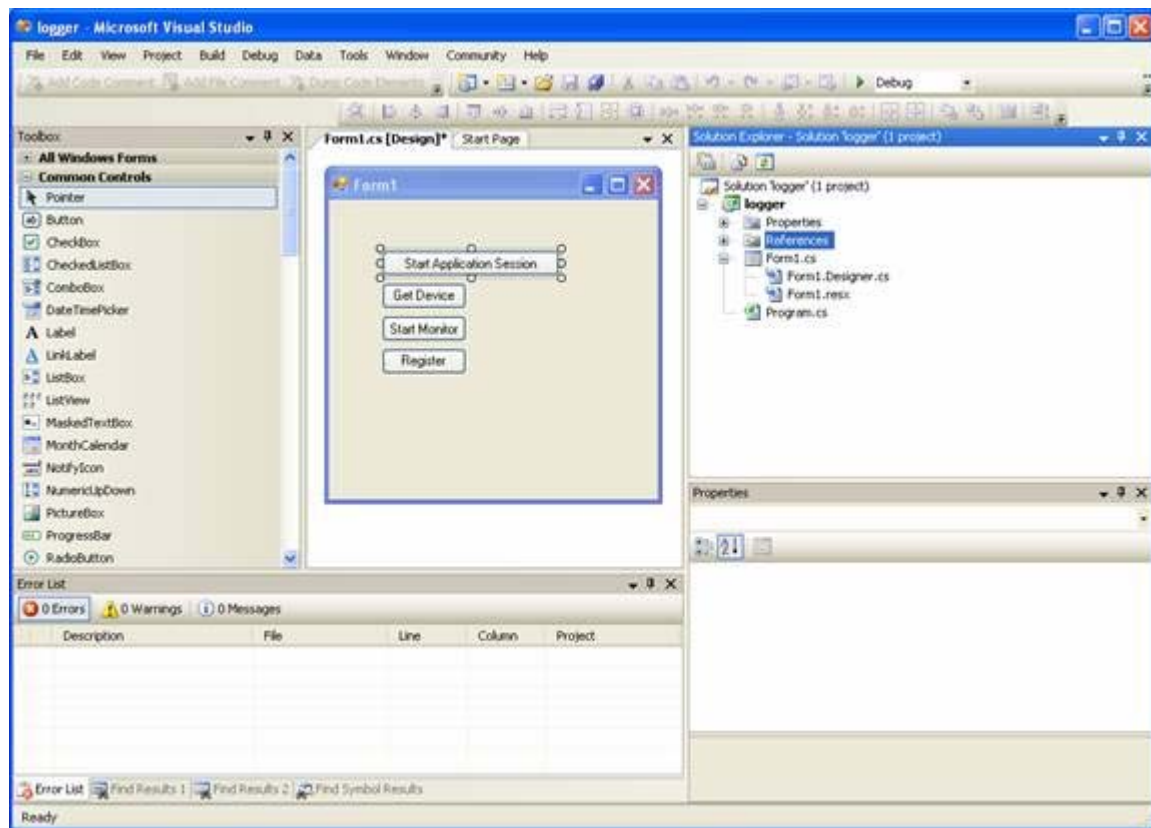
- Start Application Session
- Get Device Id
- Start Monitor
- Register Terminal



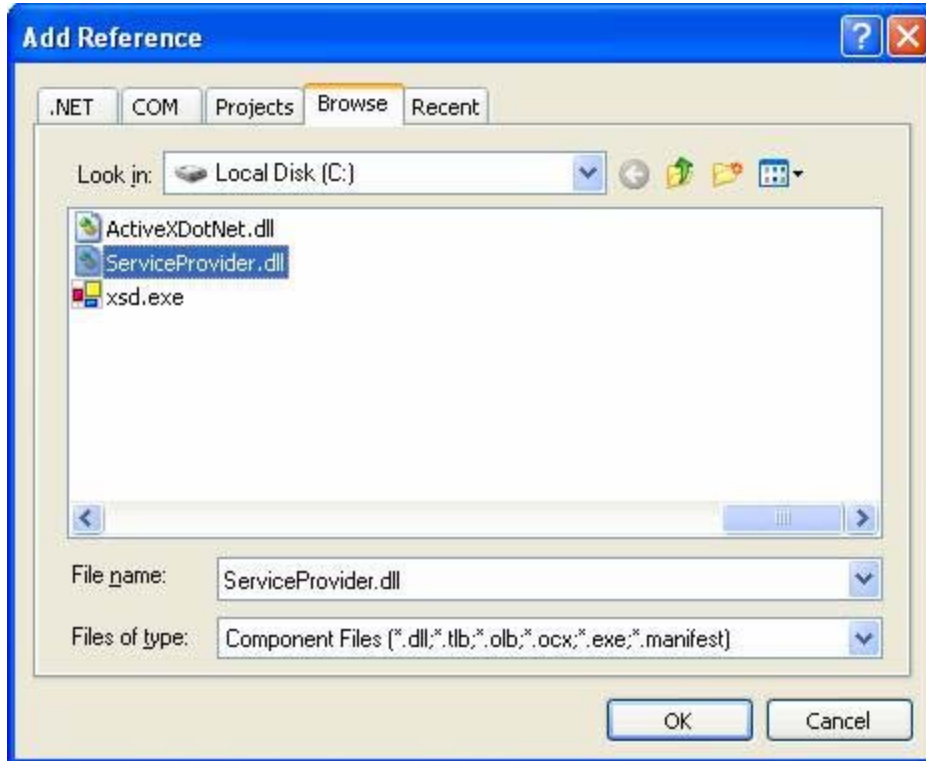
Before we add any code, we need to add a reference to the **ServiceProvider.dll** file.

Writing a client application

Right click on **References** and select **Add Reference**

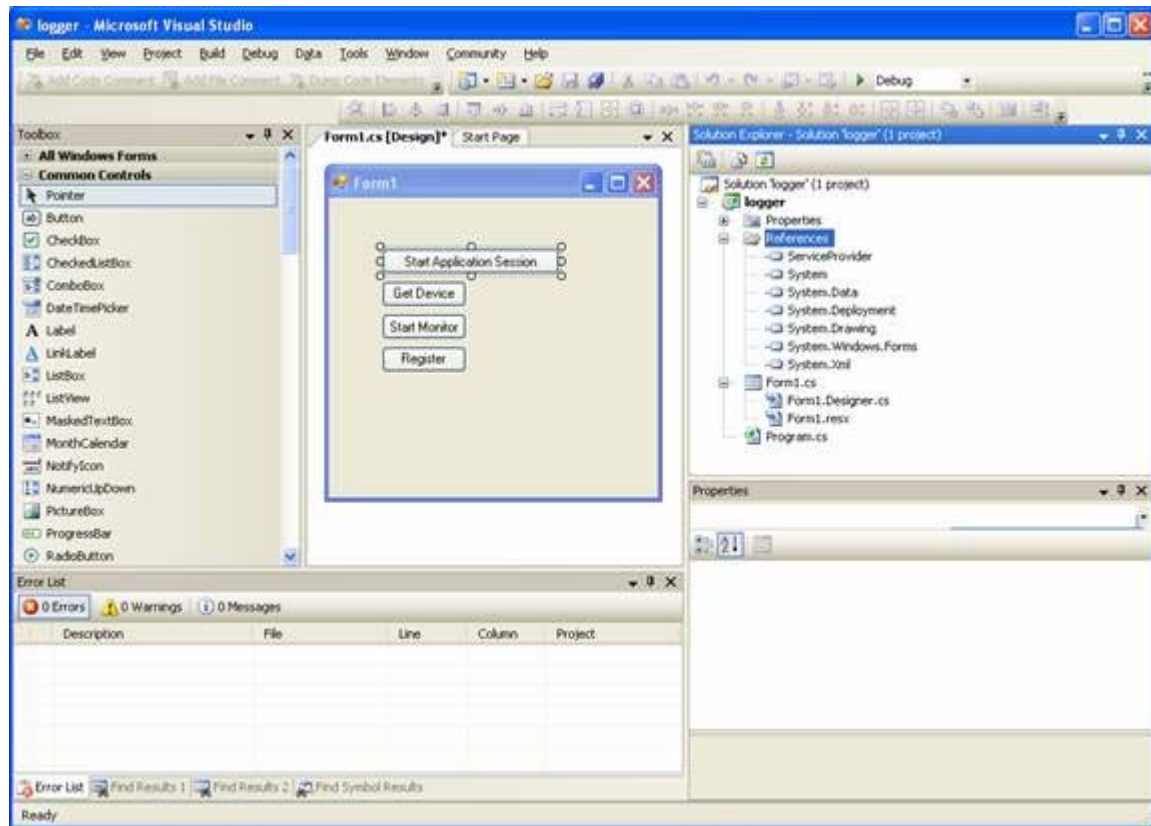


A dialog will show up, select the **Browse** tab and go to the folder where the **ServiceProvider.dll** file resides (it comes with the .NET SDK .zip file).



Writing a client application

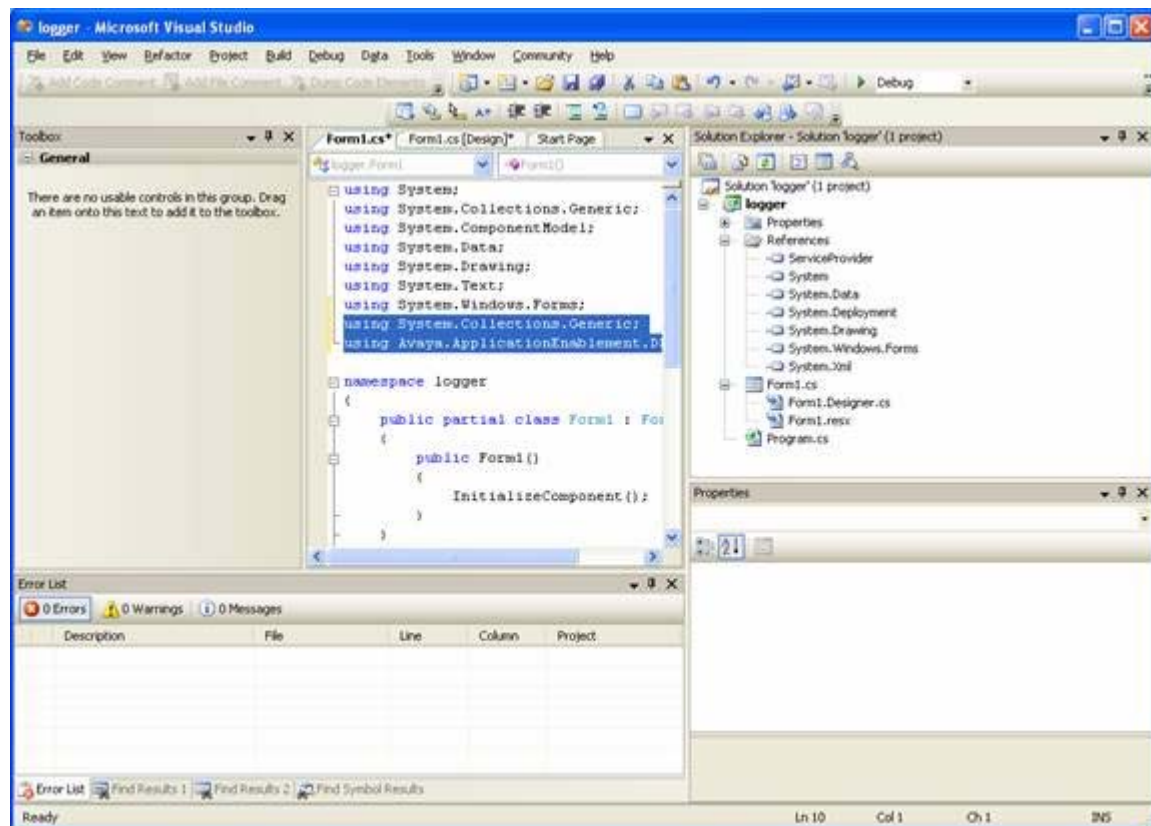
Select **OK** and you should now see it in the list of references.



Now right click on **Form1.cs** and select **View Code**.

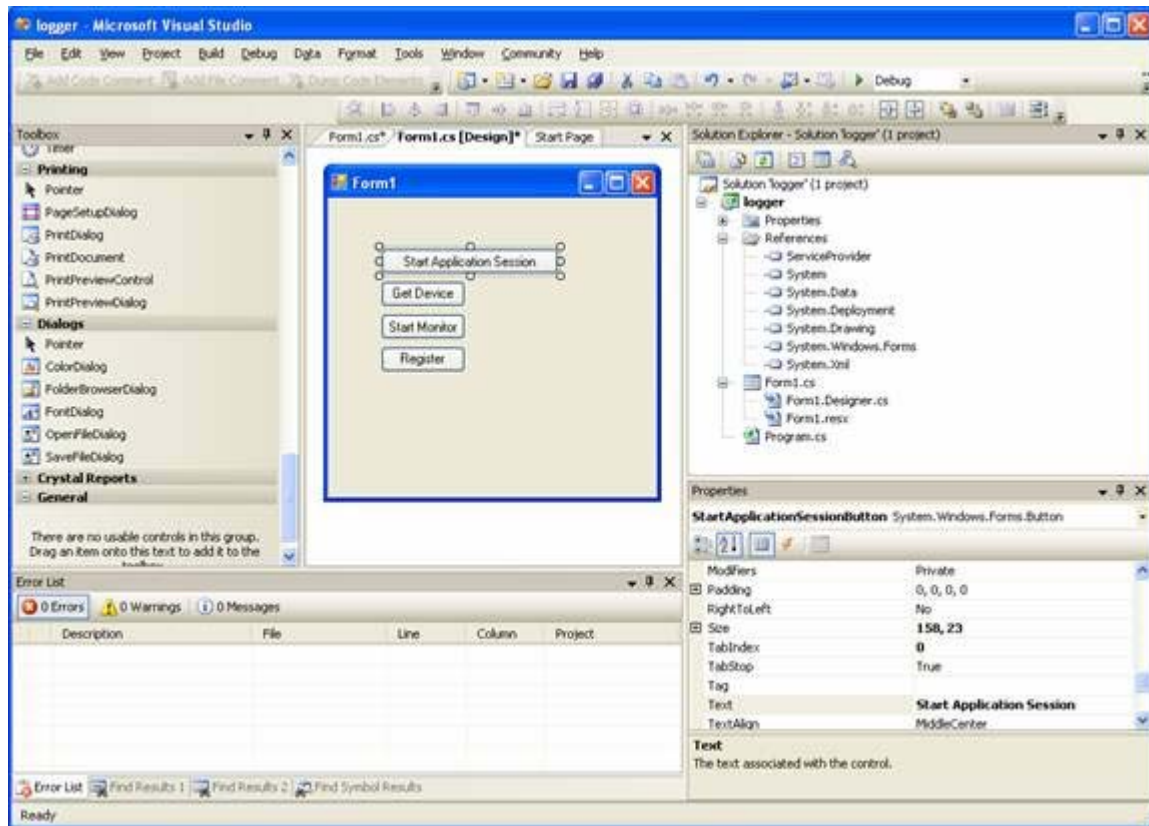
Insert the following below the **using** directives:

```
using System.Collections.Generic;  
using Avaya.ApplicationEnablement.DMCC;
```

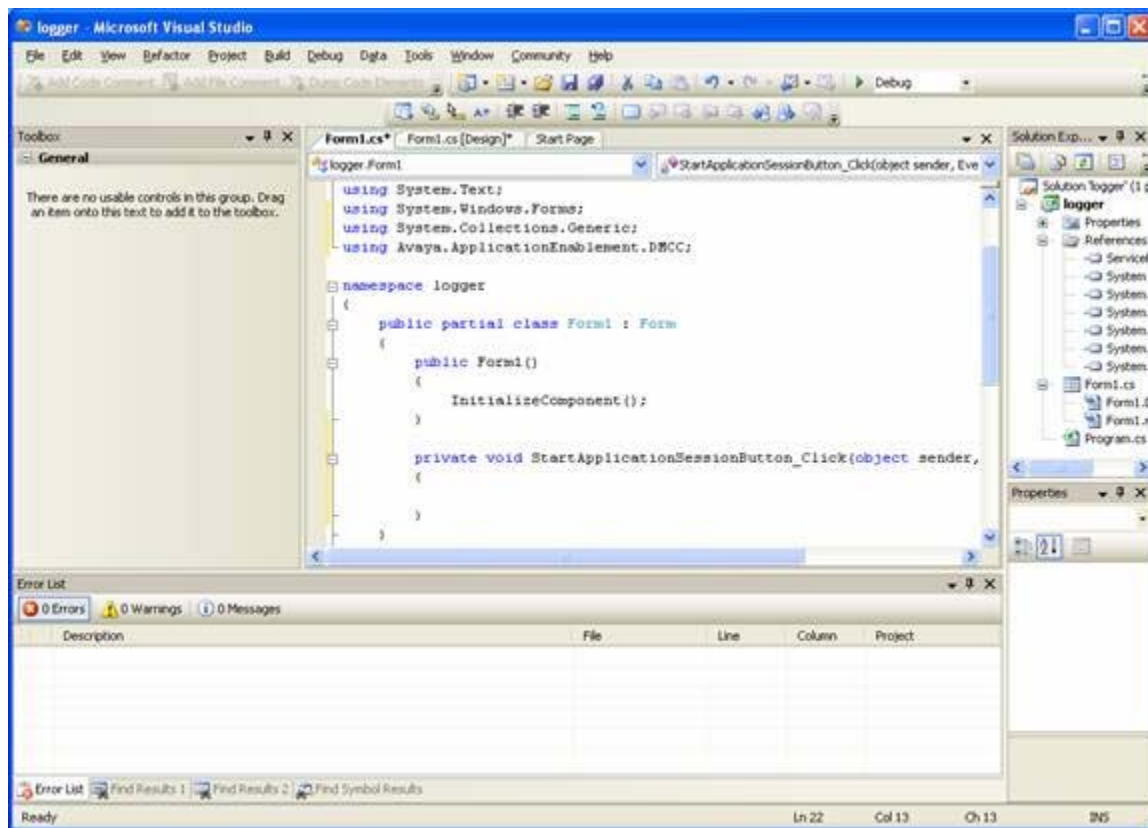


Writing a client application

Go back to design view by double clicking on the **Form1.cs** file in the Solution Explorer.



Create the handler for the **Start Application Session** button by double clicking on it.



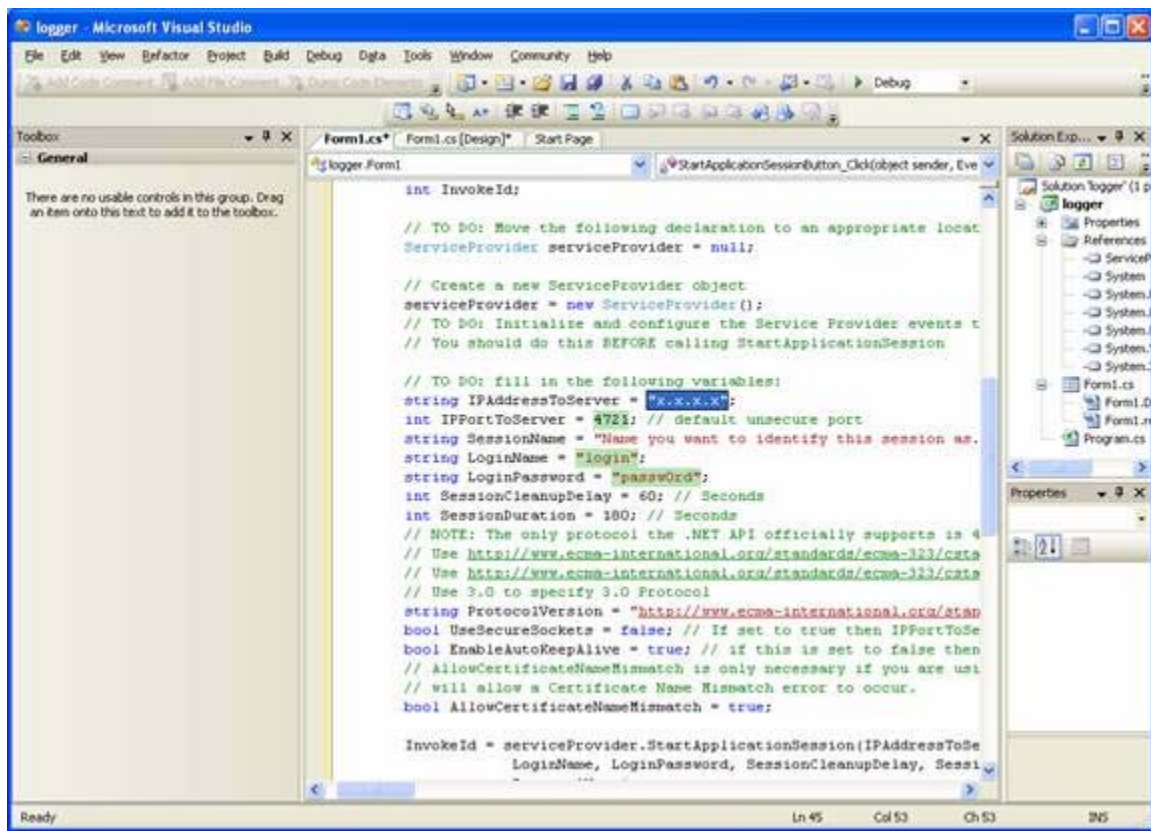
Add the code for starting an application session

- Right click in the `StartApplicationSessionButton_Click()` method
- select Insert Snippet --> DMCCCodeSnippetsC# --> Methods --> StartApplicationSession

Note:

This assumes you have already imported the code snippets which ship with the SDK.

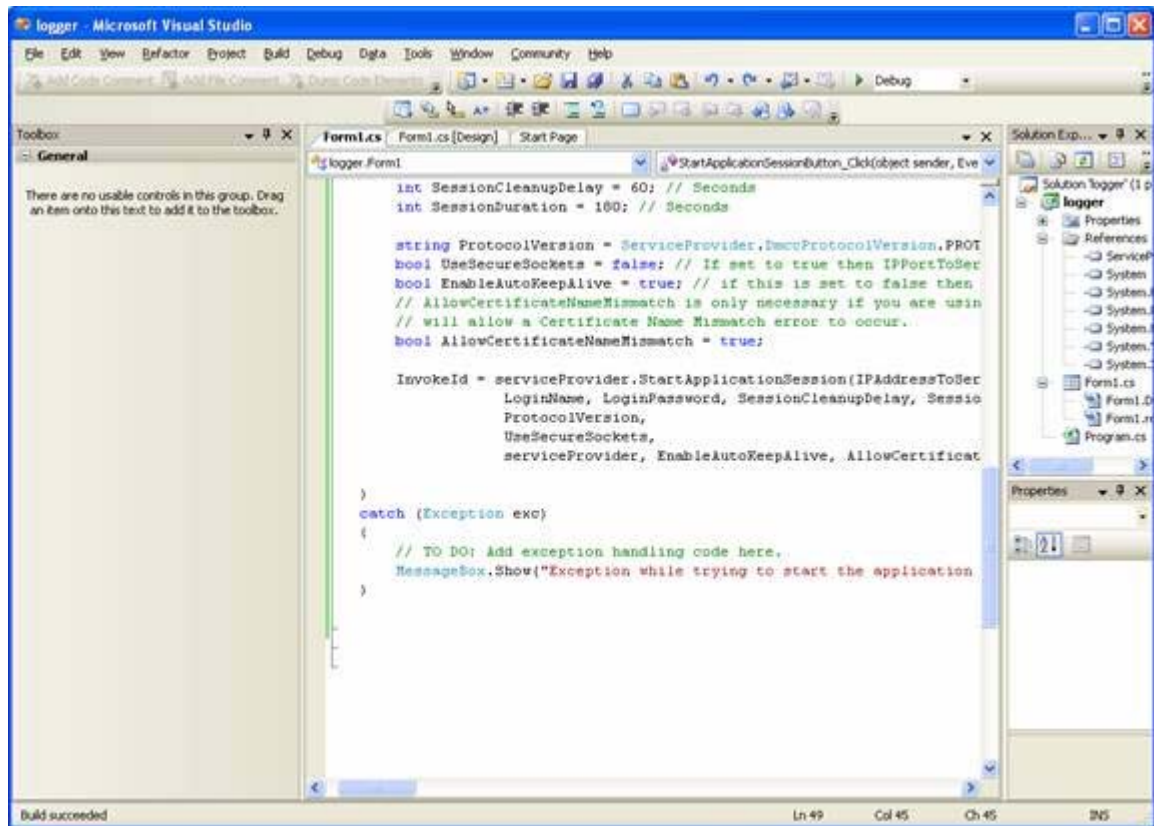
Writing a client application



Code will automatically be brought in to your workspace.

There will be comments on what you need to do in order to make it work and fields will be highlighted on what needs to be changed.

Follow the instructions and then attempt to build.



The build was successful.

Add the code to handle the response for the Start Application Session.

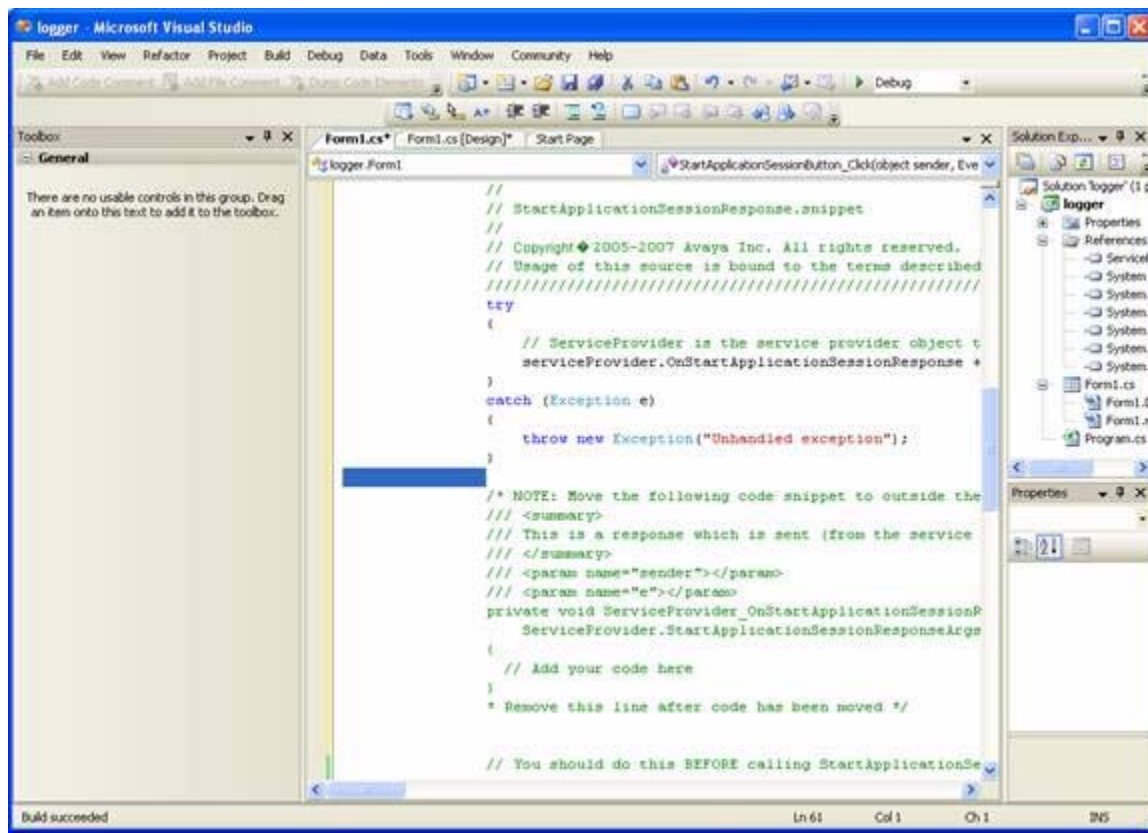
Use code snippets to create the majority of the code.

As the comments for Start Application Session indicate, we want to insert the event manager code right after we create it r (i.e., do a New).

Put the cursor right after the code that creates it:

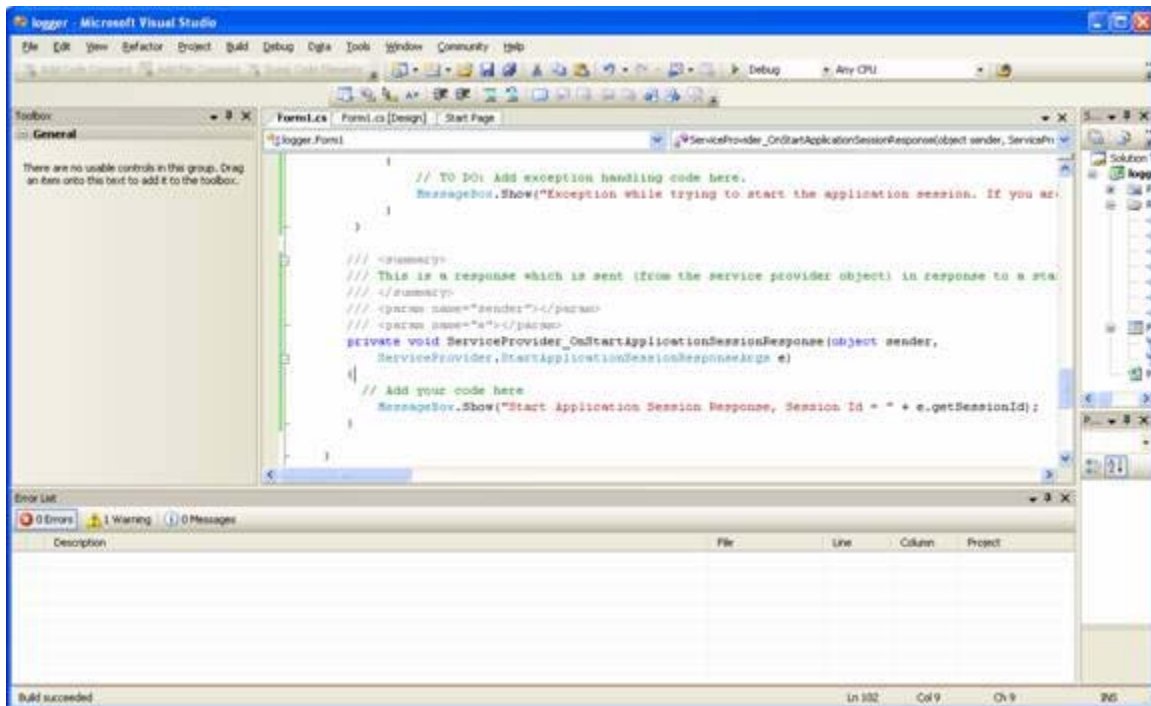
- Right click
- select Insert Snippet --> DMCCCodeSnippetsC# --> Events --> StartApplicationSessionResponseEvent

Writing a client application



The code is brought in, modify as the comments direct you to.

Modify the response handler to give you a pop-up message when it arrives.



Run the application.

Press the **Start Application Session** button.



Writing a client application

You will get a pop-up giving you the Session ID.



You have finished putting in the code for Start Application Session. By using the included Code Snippets, very little code had to be written from scratch.

You need to follow these same steps to write the code for Get Device, Start Monitors, and Register.

In this context we have ignored error detection and correction. However, this small example should provide you with the tools you need to start writing Device, Media and Call Control .NET applications.

Note:

If you want to write the code in Visual Basic, follow the same steps using the supplied Visual Basic code snippets instead.

We have only covered a very small part of the Dashboard's capabilities.

The Dashboard exposes all the DMCC functionality which is available to you from .NET including, but not limited to:

- Playing wave files
- Recording Wave files
- All Third party call control features
- Receiving RTP media
- Tone Detection
- Link Status
- Getting Button Information
- Pushing Buttons
- Making Calls
- Answering Calls
- Conference and Transfer
- Monitoring XML messages going to/from the DMCC server
- Ability to Inject and XML message that you want

The combination of the Dashboard and code snippets should allow you to not only quickly learn about what DMCC can do but also be able to write applications very quickly.

Writing applications that run in a browser

You can write Device, Media and Call Control .NET applications that will run in an Internet Explorer browser.

Note:

This API is not compatible with non Microsoft browsers.

Extensive JavaScript libraries have been provided to make it relatively easy to write thin clients.

An HTML version of the Dashboard ships with the SDK. It allows you to try out the various interfaces in Internet Explorer. You can also view sample JavaScript code from the browser dashboard.

The HTML version of the dashboard is located in the **jscript/dashboard** folder and the html file is called **dashboard.html**.

There is also an HTML softphone sample application in the **js/softphone** folder.

By looking at the source code for the softphone and using the dashboard you should be able to start writing thin client applications with minimal effort.

Registering devices

To monitor or control a device, an application must register the device with Communication Manager. This tells Communication Manager whether you want **Main**, **Dependent** or **Independent** control of the device and what kind of media access you want. Only devices that are softphone enabled on Communication Manager's **Station** form can be registered.

You must check the registration state of a device with a `Phone.GetRegistrationState` request before sending a register request if it is controlled by more than one clients session.

Note:

Registering a device with Communication Manager is only necessary for device and media control, not call control.

AE Services will verify that the user invoking any DeviceID based service request is authorized to control that DeviceID. An error will be returned if an unauthorized user attempts to control a device.

Communication Manager 5.0 allows up to three Device, Media and Call Control station clients to register to one extension. This extension must be administered as a DCP or Avaya H323 IP Softphone. All three endpoints that are registering to an extension can request separate media streams. Each endpoint registration must go through separate AE Servers.

Registration is performed through the `registerTerminal` request of the Phone Class found in the `Avaya.ApplicationEnablement.DMCC.Phone` namespace. In the `RegisterTerminal` request you must at a minimum specify which device to register and the

Writing a client application

password that is administered for the device on the **Station** form. There are also a number of options to choose from. For more details on setting up a RegisterTerminal request, see the *Programmer's Reference*.

The only registration-related event supported is the Phone class's OnTerminalUnregistered event. This event is sent if the AE Services server automatically unregisters the device. The typical cause is when the network or Communication Manager is unresponsive.

If a device becomes automatically unregistered, it is up to the application to re-register. Avaya recommends retrying every 30 seconds.

Some important decisions you will need to make when registering a device include:

- What registration dependency mode to choose
- What registration media mode to choose
- What codecs to choose
- What media encryption to choose

Controllable telephone types

For Device and Media Control, any DCP or H323 IP Softphone that can be enabled for IP Softphone can be controlled by the AE Services server. For details on how to softphone enable a device, see the appropriate *Avaya MultiVantage™ Application Enablement Services Installation and Upgrade Guide for the offer you have purchased (bundled server or software only)*.

In the case of DCP and H323 IP softphones, it is not necessary to have a physical set present. Device, Media and Call Control endpoints can register in any registration dependency mode. An endpoint registered in **Main** mode is usually associated with a physical set, but a DMCC endpoint can also register as **Main**.

For Call Control, any Communication Manager supported endpoint can be controlled by the AE Services server

You cannot choose **Independent** or **Dependent** dependency mode with SIP endpoints, so you cannot exercise device monitoring/control over them.

Registration modes

Dependency modes

At registration time, the application specifies the desired registration dependency mode. This indicates who controls the device's physical elements and media. The dependency choices are:

- **Main dependency mode**

There can be only one **Main** registrant associated with an extension.

A device that registers in this mode does not depend on registration of any other endpoints using the same extension.

Once in a call this end point can talk and listen.

Only an endpoint registered as **Main** can block the transfer of talk capability to an endpoint register as **Dependent** or **Independent** via the "share-talk" button.

Note:

"share-talk" button push is processed only if an endpoint registers using any media mode other than **No Media**.

- **Independent dependency mode**

A device that registers in this mode can originate and receive calls and can talk and listen even when the **Main** device is not registered.

If an end point registers as **Main** after an endpoint registers as **Independent** on the same extension, talk capability is transferred to the endpoint registered as **Main**.

An endpoint registered as **Independent** cannot block transfer of talk capability by pressing the "share-talk" button.

- **Dependent dependency mode**

A device will be allowed to register in this mode only if another device is already registered to Communication Manager using the same extension in **Main** mode.

A request to register using **Dependent** mode will fail unless another endpoint is already registered to that extension in **Main** mode.

Communication Manager will unregister a device registered in **Dependent** mode if the **Main** endpoint unregisters.

When on a call, endpoints registered as **Dependent** will be in listen-only mode.

An endpoint registered as **Dependent** cannot block transfer of talk capability by pressing the "share-talk" button.

Note:

For further information on the "share-talk" button see [The "share-talk" button](#) on page 70.

When to use:

- **Main**

Use this mode when you need to be able to answer calls, make calls, talk, or listen, regardless of the presence of other registrants.

This mode can be used with any Media Mode including **No Media**.

- **Independent**

Use this mode when you wish registration to succeed regardless of whether a **Main** registrant already exists.

This mode can be useful when two instances of an application register an extension that doesn't have an associated end-user telephone. One instance of the application would register in **Main** mode and the other in **Independent** mode. Communication Manager will allow either application to answer calls, make calls, talk (one at a time) or listen, immediately after they register.

- **Dependent**

Use this mode when you wish registration to succeed only if a **Main** registrant already exists and you want to unregister when the **Main** registrant unregisters.

This mode can be useful if you wish to listen to a call whose device is already registered as **Main**.

This mode would also be useful if you wish to monitor/control a physical device.

Media modes

When a device is registered by an application, the application has access to the real-time protocol (RTP) media stream coming into and going out from the softphone.

There are four media modes available when a device is registered in **Main** dependency mode: server media, client media, telecommuter, and No Media.

When the device is registered in either **Dependent** or **Independent** dependency mode, then only server media, client media or No media is available to control the media stream.

The following table, [Registration Dependency and Media modes](#), illustrates what media modes are allowed with each dependency mode.

Table 15: Registration Dependency and Media modes

	Client	Telecommuter	Server	No Media
Main	Allowed ^a	Allowed ^a	Allowed ^a	Allowed
Dependent	Allowed	Not Allowed	Allowed	Allowed ^b
Independent	Allowed	Not Allowed	Allowed	Allowed

a. Corresponds to Exclusive Control in previous releases

b. Corresponds to Shared Control in previous releases. In this mode, the user of a telephone is not notified of actions initiated by an application except through resulting status changes to the device's lamps and display. Also in this mode, the application is not notified of actions initiated by a user of the telephone except by status changes to device's hookswitch, lamps, ringer, and display.

When multiple endpoints are registered to the same extensions, all endpoints see activity on other endpoints when status changes to the device's hookswitch, lamps, ringer, and display. An endpoint in this case would not be aware of the following action taken by another endpoint:

- If endpoint **A** presses a digit to make a call, other endpoints (for examples, **B** and **C**, assuming 3 endpoints registered to an extension) will not see the digits sent by **A** to the AE Services server.
- Speaker button press, Head Set button press and taking Head Set off the cradle are seen as off-hook by Communication manager. And therefore other endpoints registered to the same extensions will only see off-hook event.
- Feature button presses are undetectable unless the feature button has a lamp that toggles when the button is pressed.

The RTP parameters that an application can control or state preferences for at registration time are:

- Local RTP and RTCP addresses
- Coder/decoder (codec): G.711 A-law, G.711 Mu-law, G.729, G.729A

Choosing a media mode

The following table [Choosing a media mode](#) shows the media modes available, when to use each, and how to request each:

Table 16: Choosing a media mode

Media modes	When to use	How to set
Server media mode	This mode is used when the application wants the AE Services server to handle media processing. The AE Services server handles media with Voice Unit Services and Tone Detection Services or Tone Collection Services. Voice Unit Services is used to record and play messages. Tone Detection Services or Tone Collection Services are used to detect out-of-band DTMF tones. In server media mode, the media stream terminates on the AE Services server. A <code>MediaControlListener</code> may be used in this mode to detect changes to the far-end RTP/RTCP parameters.	Set <code>MediaInfo.MediaControl</code> to <code>Media.MediaMode.SERVER_MODE</code> in the <code>MediaInfo</code> argument that is passed in the <code>Phone.RegisterTerminal()</code> method.
Client media mode	This mode is used when application wants to process the media itself. The RTP stream can be terminated on any machine that can be controlled by the application. A <code>MediaControlListener</code> may be used in this mode to detect changes to the far-end RTP/RTCP parameters. <code>ToneDetectionServices</code> or <code>ToneCollectionServices</code> can be used to detect out-of-band DTMF tones.	Set <code>MediaInfo.MediaControl</code> to <code>Media.MediaMode.CLIENT_MODE</code> in the <code>MediaInfo</code> argument that is passed in the <code>Phone.RegisterTerminal()</code> method.
1 of 2		

Table 16: Choosing a media mode (continued)

Media modes	When to use	How to set
Telecommuter mode	This mode is used when an application wants the media to go to a real telephone. The real telephone can be an internal dial string to Communication Manager or a PSTN telephone number.	Set <code>MediaInfo.MediaControl</code> to <code>Media.MediaMode.TELECOMMUTER</code> in the <code>MediaInfo</code> argument that is passed in the <code>Phone.RegisterTerminal()</code> method.
No Media	This mode is used when the application does not want an RTP stream to be setup to it by Communication Manager.	Set <code>MediaInfo.MediaControl</code> to <code>Media.MediaMode.NONE</code> in the <code>MediaInfo</code> argument that is passed in the <code>Phone.RegisterTerminal()</code> method.
2 of 2		

The following table [Media Control Capabilities](#) shows the media control capabilities of the AE Services server for both server media mode and client media mode.

Table 17: Media Control Capabilities

Media control capabilities	server media mode	client media mode
Record media from a call into a WAV file	provided by the server	provided by the application
Dub a recording with the contents of another compatible WAV file	provided by the server	provided by the application
Play a voice announcement or tone from a prerecorded WAV file	provided by the server	provided by the application
Play a list of prerecorded announcements from separate WAV files	provided by the server	provided by the application
Stop, pause, or resume outstanding play or record operations	provided by the server	provided by the application
Detect out-of-band DTMF tones	provided by the server	provided by the application
1 of 2		

Table 17: Media Control Capabilities (continued)

Media control capabilities	server media mode	client media mode
Detect inband DTMF tones	provided by the server	provided by the application
Manage media related event listeners	provided by the server	provided by the application
Originate and terminate RTP streams on any machine that the application has control of		provided by the application ^a
Media encryption	provided by the server	provided by the application ^a
2 of 2		

a. Media encryption is provided by the application unless the application is using the Avaya provided client media stack in which case it will be provided by the server.

The "share-talk" button

Although Communication Manager 5.0 allows up to three Device, Media and Call Control station clients to register to one extension, for each extension only one "Talk" time slot is used. If there are three endpoints registered with that extension, only one at a time will be able to talk, but all three can listen.

If your application wants the ability to share the talk capability, you will use the "share-talk" button. The "share-talk" button must have been administered in Communication Manager. For information on how to administer the share-talk button on Communication Manager, please see the *Avaya MultiVantage® Application Enablement Services Administration and Maintenance Guide*.

Communication manager will process a "share-talk" button push only if the media mode of that endpoint is not **No Media** and the extension is in a call.

Once in a call, an endpoint registered as **Main** can press this button to block any endpoint registered as **Dependent** or **Independent** from taking over the talk capability. The **Main** endpoint can then unblock it by pushing this button again.

If a **Main** endpoint has not blocked the talk capability, a **Dependent** or **Independent** endpoint can press this button to acquire the "Talk" capability. The **Dependent** or **Independent** endpoint can press this button a second time to move the talk capability back to the **Main** endpoint.

Interpretation of the 'share-talk' button lamp state

- By an endpoint registered as **Main**:

- Steady On

The **Main** endpoint currently has the Talk capability. If **Main** presses the button while in this state, the Talk capability will be blocked (see Flutter).
- Flutter

The **Main** has blocked the talk capability from being taken over by a **Dependent** or **Independent** endpoint. **Main** can unblock by pressing this button one more time and the lamp will transit back to "Steady On".
- Off

A **Dependent** or **Independent** endpoint has taken over the talk capability. If a **Main** endpoint wants to talk it can take over the talk capability at any time by pressing the button (lamp will transit back to "Steady On" after the button push).
- By an endpoint registered as **Dependent** or **Independent**:
 - Steady On

The **Dependent** or **Independent** endpoint currently has the Talk capability. When this transition happens Communication Manager will turn the "share-talk" button lamp off at other endpoints associated with this extension. While in this state, a **Dependent** or **Independent** endpoint can transfer the talk capability back to **Main** by pushing the button.
 - Flutter

The **Main** has blocked the talk capability from being taken over, The **Dependent** or **Independent** endpoint cannot obtain the talk capability.
 - Off

A **Dependent** or **Independent** endpoint has no Talk capability, however it can take over the Talk capability if it desires.

Choosing a codec

A codec is the algorithm used to encode and decode audio media. For devices choosing media modes of either client or server media, the application can optionally specify at registration time what codecs are preferred for the device. The codec options are:

- G.711 A-law (g711A)
- G.711 Mu-law (g711U)
- G.729 (g729)
- G.729 Annex A (g729A)
- G.723 (supported as an option for Client Media mode only)

Specify the set of codecs your application supports and list them in preferential order in the `Phone.MediaInfo` parameter of `RegisterTerminal` request. If you do not specify a set of codecs in the `Phone.MediaInfo` parameter of the `RegisterTerminal` request, the

Writing a client application

connector server will default to G.711 A-law as the first choice and G.711 Mu-law as the second choice. If Communication Manager cannot satisfy your request for specific codecs, then calls will still go through, but there will be no media.

Note:

For server media mode you cannot specify a mixture of G.711 and G.729 codecs for a single device. This is because there is no conversion offered by the server.

For more information about selecting and administering network regions and their codecs, see Chapter 1 of the *Avaya MultiVantage™ Administration and Maintenance Guide*.

Choosing the media encryption

For devices choosing media modes of either client or server media, the application can optionally specify, at registration time, what media encryption is preferred for the device's media stream.

The media encryption options are:

- Advanced Encryption Standard (AES)
- none (i.e. no encryption of the media stream)

Specify the set of encryption options your application supports and list them, in preferential order, in the `MediaInfo` parameter of the `RegisterTerminal` request. If you do not specify a set of encryption options in the `MediaInfo` parameter, the connector server will default to "none" (no media encryption).

Note:

You will find information on call capacities your application can expect to be able to handle in the "Capacities for calls in Device, Media and Call Control applications" section of the "Capacities for types of applications" chapter of the *Avaya MultiVantage™ Application Enablement Services Overview*. This section includes information on how the choice of registration dependency modes, media modes, codecs and encryption can affect the performance of your applications.

Telephony Logic

The following sections will give general instruction how to perform common things around the telephone. We encourage you to use the Dashboard application to investigate further. The actual .NET source code is available by using the Visual Studio Code Snippets that are in the SDK.

Knowing what buttons are administered

If your application needs to press any buttons or determine which lamps have changed state, you will need to know what buttons are administered on the device. Buttons are assigned to devices during station administration via the Communication Manager system access terminal (SAT) interface.

Your application must use the `Phone.GetButtonInfo()` method to ask about a particular button or to ask to get all button information. Look under the Dashboard "Phone Commands" to test this functionality.

Detecting an incoming call

When a call comes into a device, these three changes occur to the physical device:

- The phone rings.
- A green call appearance lamp flashes.
- The display changes to show caller information.

To get this information, you must set a phone monitor to monitor these events. You can try this in the dashboard and use the Visual Studio code snippets to look at sample source code.

Determining that far end has ended the call

If all far-end parties drop on a call, these changes occur on the local device:

- The call appearance green lamp turns off.
- The display is updated based on the current state of the extension. For example:
 - Returns to an idle state showing extension, date and time information
 - Begins showing information about a ringing call at the extension
 - Is not updated and continues to show information relative to an active display feature such as Directory Lookup

Getting ANI information for a call

You use the conference display button to get ANI information for a call.

In order to use the conference display button to get ANI information for the call, the conference display button (conf-dsp) will have to have been administered for that extension in Communication Manager.

Once an incoming call is received by your application for the extension, your application should press the conference display button of the extension repeatedly to get the ANI information (through the `DisplayUpdatedEvent` of Phone Services) for each party on the call.

Recording and playing voice media

If your application needs to record incoming media or play a message on a call, then at registration time:

- You must choose to either handle the media yourself (client media) or have the AE Services server do it for you (server media). See [Choosing a media mode](#) on page 68.

This section describes how to use the Media Object to have the AE Services server record and play media for you. The supported services are described in the [Media Class](#) on page 20. The details about using the services are described in the Programmers Reference. To see sample code using the Media Class, see the sample application called `SimpleRecord` referenced in [Learning from sample code](#) on page 41.

Some basic rules of Voice Unit Services are:

- **Wave files**

All digital audio files that are created or played are in the Wave Resource Interchange File Format (RIFF). The standard Wave file structure is used for all encoded media types. See <http://www.sonicspot.com/guide/wavefiles.html> for a description of the Wave structure.

G.729 formatted files, however, use non-standard field values in some of the standard format chunk fields. They are:

- Compression code value: 131 (0x0083)
- Block align value: 10 (0x000A)
- Bits per sample value: 1 (0x0001)

An external G.729 converter is required to convert a G.729 Wave file into a standard RIFF Wave file that can be played.

- **Files on connector server**

All Wave files are assumed to be on the AE Services server machine in the directories specified in the OAM interface under the media properties as the player directory and the recorder directory.

Note:

These two directories do not have to be the same. These directories are the root directory of the relative paths specified in the `playMessage` and `recordMessage` requests.

- **Encoding algorithms**

Files to be played can be encoded in these formats:

- PCM 8-bit or 16-bit
- G.711 A-law
- G.711 Mu-law
- G.729
- G.729A

Recordings can be made of calls in these formats

- PCM 8-bit or 16-bit
- G.711 A-law
- G.711 Mu-law
- G.729

Note:

While in server mode, you can find out what codecs have been chosen from the `MediaStartedEvent`, which will contain a list of the codecs.

Note:

Codecs must be specified at registration time and while in server media mode you cannot specify a mixture of G.711 and G.729 codecs for a single device. This is because there is no conversion offered by the server.

- **Conversions between encoding algorithms**

Files to be played can be converted from any PCM type to any G.711. No other conversions are supported for playing.

Messages to be recorded can be converted from G.711 to PCM. No other conversions are supported for recording.

- **Using with Tone Detection or Tone Collection Services**

The Voice Unit Services player and recorder may be setup to detect DTMF tones at the same time Tone Detection or Tone Collection Services is being used. However, there is no

guarantee which service will detect a tone first. See [Possible race conditions](#) on page 92 for more specifics.

Media Operations

media operations such as Recording, Dubbing, playing wave files are all done through the Media Object. Use the dashboard tool to learn more about these capabilities and use the Visual Studio code snippets to help you write the actual code.

Determining when far-end RTP media parameters change

You will need to set a monitor for the Media Started and M3dia Stopped events. When the RTP media changes you will be notified by these events. Use the Dashboard tool to experiment and use the Visual Studio code snippets to help you write the code.

Recovery

The application session protocol that was implemented for the 3.0 release has several recovery benefits for the Device, Media and Call Control application. The keepalive messages that are sent periodically by the API to the AE Services server provide a way for the application to verify that the AE Services server is operational even if there is no other activity from the application. In addition, the session cleanup delay provides a mechanism for the application to reestablish its session after a short network interruption without having to reestablish their state (e.g. register devices again). This section tells you how to take advantage of these new features to design a more robust Device, Media and Call Control application.

The best way to monitor for problems with a session is to monitor for `ServerConnectionDownEvents`, `ServerConnectionNotActiveEvents`, and `MissedAtLeastOneKeepAliveEvents` within the Service Provider object.

The best way to monitor for problems with a session is to add a `ServiceProviderListener` to the application's `ServiceProvider` instance. The following is a list of events the application can receive via this listener, and the proper recovery actions to take in the case of each event.

- `ServerConnectionDownEvent`. This event is sent if the socket to the AE Services server has gone down for some reason. The recovery action to take for this event is to call `reconnect` on the application's `ServiceProvider` instance. This method will attempt to open a new socket to the AE Services server, and to reestablish the existing session with a new socket.

If successful, the application should be able to resume operations without reestablishing its state although your application may have missed some events. To be sure that you know

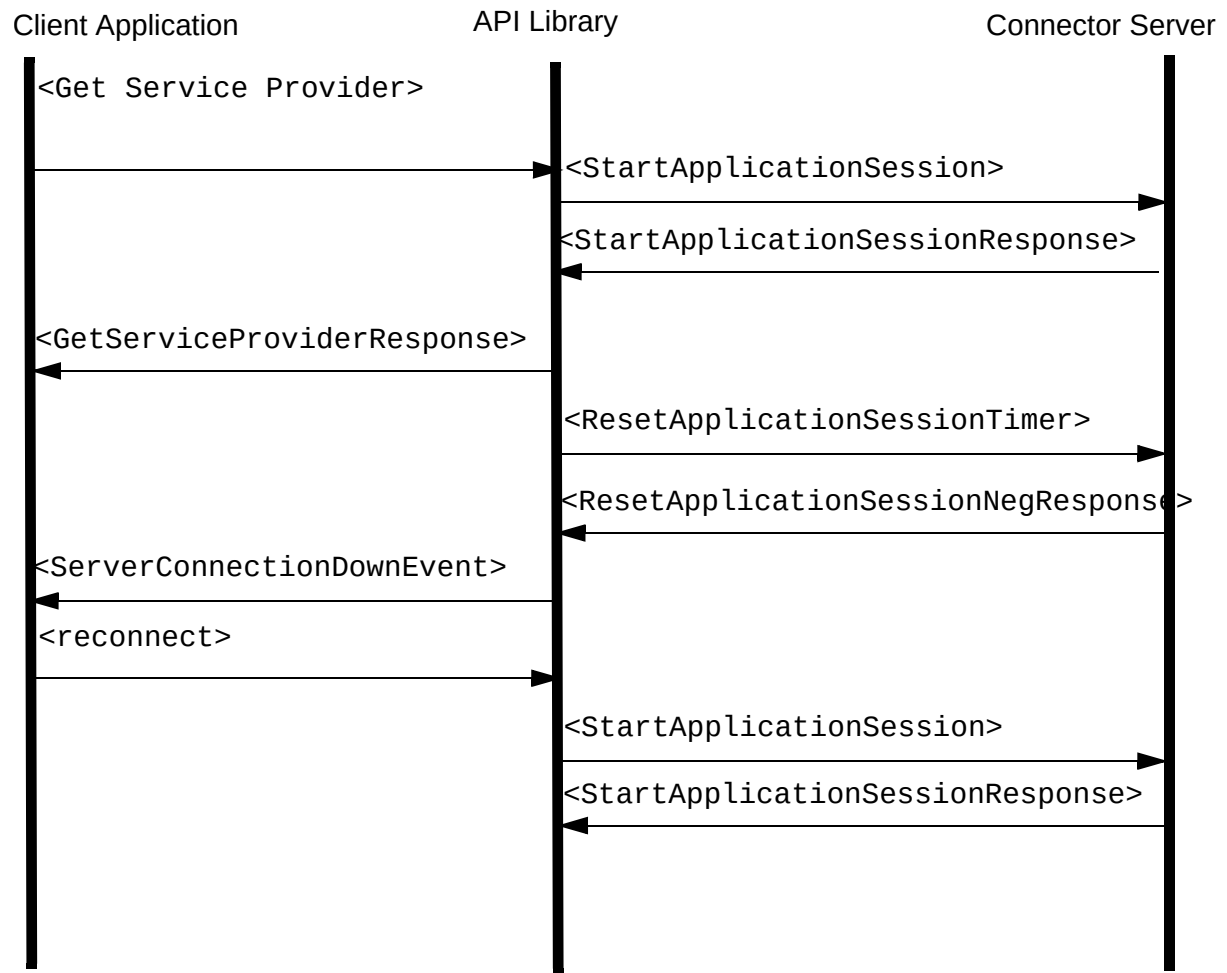
the current state of the lamps, hookswitch and display, you will have to query for those states using `getLampState`, `getHookswitch` and `getDisplay` to update the states.

- **ServerConnectionNotActiveEvent.** This event is sent if a message was received by the AE Services server but the session has timed out and been placed in the inactive state. A session enters the inactive state if the Application Session Duration expires before a `ResetApplicationSessionTimer` message is received. Upon receiving this event, an application should take the same recovery action as for the `ServerConnectionDownEvent`.
- **MissedAtLeastOneKeepAliveEvent.** This event comes in if the server has missed at least one keep alive.

Note:

By default, the .NET Service Provider handles the automatic sending of Keep Alives to the AE Services server for you. However, you can turn this capability off via the `StopAutoKeepAlive()` method in the Service Provider object (use `StartAutoKeepAlive()` method to start the processing back up).

The following figure shows the sequence of events that occurs between the client application, the API library and the connector server during establishment of an application session, and an instance of a `ServerConnectionDownEvent` occurring during subsequent session management.

Figure 1: Session Management Recovery

Cleanup

If the cleanup session expires resources are reclaimed. It is important to free resources when they are no longer needed. This is most likely to occur when your application:

- detects the end of a call
- is finished with a device
- is about to exit

Cleanup should occur in this order:

1. Stop collecting tones

At the end of a call, you can choose to stop collecting DTMF tones for the device. Alternatively, you can let the collection and the retrieval criteria continue across calls. In that case you might just flush the buffer at the end of each call.

When finished with a device, stop the tone collection for that device.

When the application is about to exit, stop tone collection on all devices.

2. Stop recording or playing

At the end of a call, you can choose to stop recording or playing or let the recording or playing continue across calls on the device.

When finished with a device, stop both recording and playing on that device.

When the application is about to exit, stop both recording and playing on all registered devices.

Both recording and playing can be stopped on a device using `Media.stopRecording` method and `Media.stopPlaying`

3. Unregister the device

When finished with a device, unregister it using the `Phone.unregisterTerminal` method.

When the application is about to exit, unregister each registered device.

4. Stop the monitor(s)

When your application no longer needs to receive events for a device, stop the monitors that were started using the `StopMonitor()` methods for the appropriate services.

5. Release the device identifier

When finished with a device identifier, release it using the `Device.releaseDeviceID` method.

When the application is about to exit, release each device identifier.

6. Stop the Application Session

Stop the application session by calling the `StopApplicationSession` method on the application's `ServiceProvider` instance.

Media Encryption

Application Enablement Services offers the user the ability to encrypt the voice RTP streams between the CMAPI softphone and the far end of the call when the application chooses to use server media mode.

For Application Enablement Services, the only encryption scheme available is the Advanced Encryption Standard (AES) media encryption.

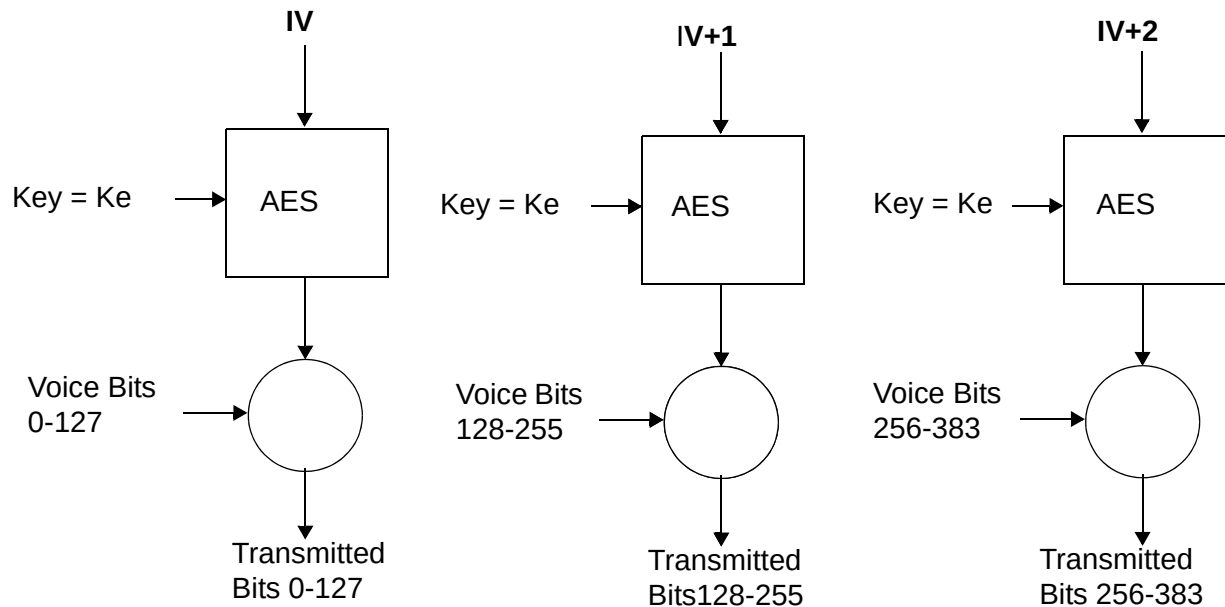
The AES Encryption Scheme

1. Encrypting the Voice Stream

To transmit voice information over a digital medium, the analog voice signal is sampled at discrete intervals. Each sample is represented as an 8-bit number or 8-bit byte. Samples (bytes) are transmitted sequentially as a stream of bits. If a G.711 codec is used to generate the samples and if 20 milliseconds worth of data is sent in each Internet Protocol (IP) packet, then each packet will contain 160 bytes which equals 1280 bits. (8000 samples/second for 20 milliseconds).

To send the voice data, the stream of voice bits is exclusive OR'd with a second stream of bits before being transmitted. This second stream of bits is generated cryptographically and the resulting transmitted stream is said to be "encrypted". The two bit streams are processed in fixed "chunks" of 128 bits as illustrated in [Figure 2](#)

Figure 2: Encryption of the Voice Stream



A 128 bit initialization vector (IV) is encrypted with key KE (128 bits) using the AES encryption algorithm to produce 128 bits of output. Those 128 bits are exclusive OR'd with

the first 128 bits of the voice packet. The initialization vector is incremented by one, and the process repeated for the next 128 bits. Ten repetitions are required to send one 20 millisecond packet which contains 1280 bits of voice data. This means that the AES encryption engine is run 10 times to send one packet. This mode of using AES is called counter mode (because the IV acts as a counter).

On the receiving end, the same process is used to recover the original voice data. The receiver must have the exact same key (KE) and the same initialization vector (IV).

2. Generating Key Material

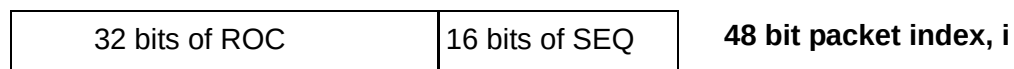
In order to generate the encryption key, initialization vector, and other keys to be seen shortly, the following operations are performed. Note that these computations are performed anew for each RTP packet.

Let the packet index “i” be defined as:

$$i = (32\text{-bit ROC}) \parallel (16\text{-bit SEQ for RTP})$$

where ROC is the roll over counter, SEQ is the 16-bit sequence number from the RTP packet and $\parallel$ indicates concatenation. This is shown in [Figure 3](#).

Figure 3: Structure of the Packet Index



Let

$$r = i \text{ DIV } \text{key_derivation_rate}$$

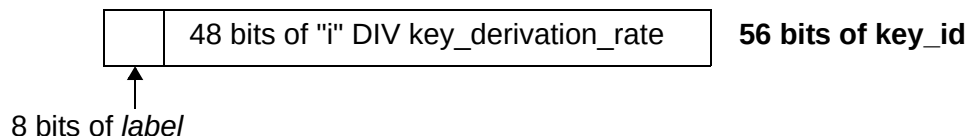
where DIV denotes integer division rounded down with the convention that dividing by 0 equals 0. The SRTP algorithm supports changing the keys periodically, even while the voice stream is active. The key derivation rate is the rate of this change. A value of zero indicates that the keys are not changed periodically. When the rate is zero, “r” is also zero (48 bits). Note that in the first computation of “r”, the value of SEQ used in the computation of “i” is the initial value at the beginning of the media stream.

Let

$$\text{key_id} = \text{<label>} \parallel r$$

where <label> = 0x00 for RTP packet encryption and 0x02 for the salting key used to generate the IV as illustrated in [Figure 4](#).

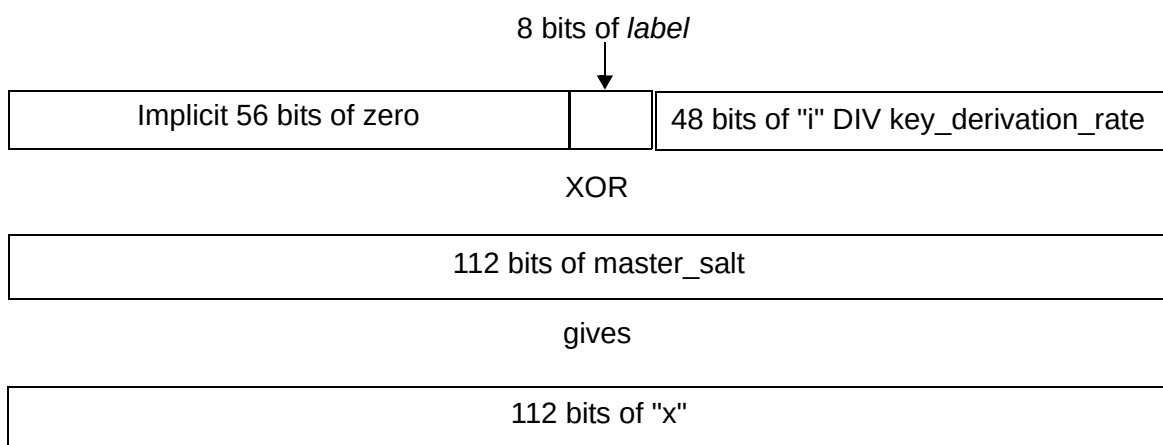
Figure 4: key_id Structure



Let $x = \text{key_id} \text{ XOR } \text{master_salt}$

This is shown in [Figure 5](#):

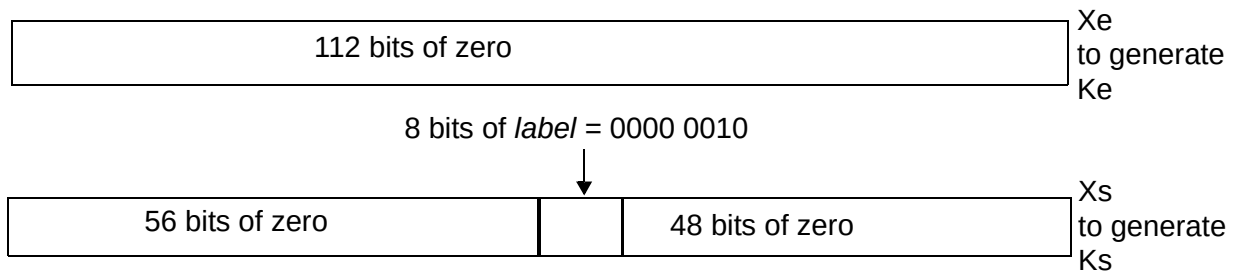
Figure 5: Computation of "x"



The keys for encryption and IV generation are the result of encryption of ($x * 2^{16}$) with a key called the master key. (The master key is distributed by the Media Gateway Controller for each voice IP media link as the link is established.)

The values of "x" for Avaya's implementation are shown in figure 6. Note that two values of "x" are computed, one (xe) is used to compute the value of KE and a second value of "x" (xs) is used in the computation of the IV.

For Avaya's implementation: Key_derivation_rate = 0 Master_salt = 0

Figure 6: Values of "x" for Avaya's Implementation

3. Creating the Initialization Vector

The Initialization Vector (IV) changes for each packet (1280 voice bits for 20ms G.711) according to the following equation: $IV = (SSRC * 2^{64}) \text{ XOR } (KS * 2^{16}) \text{ XOR } (i * 2^{16})$

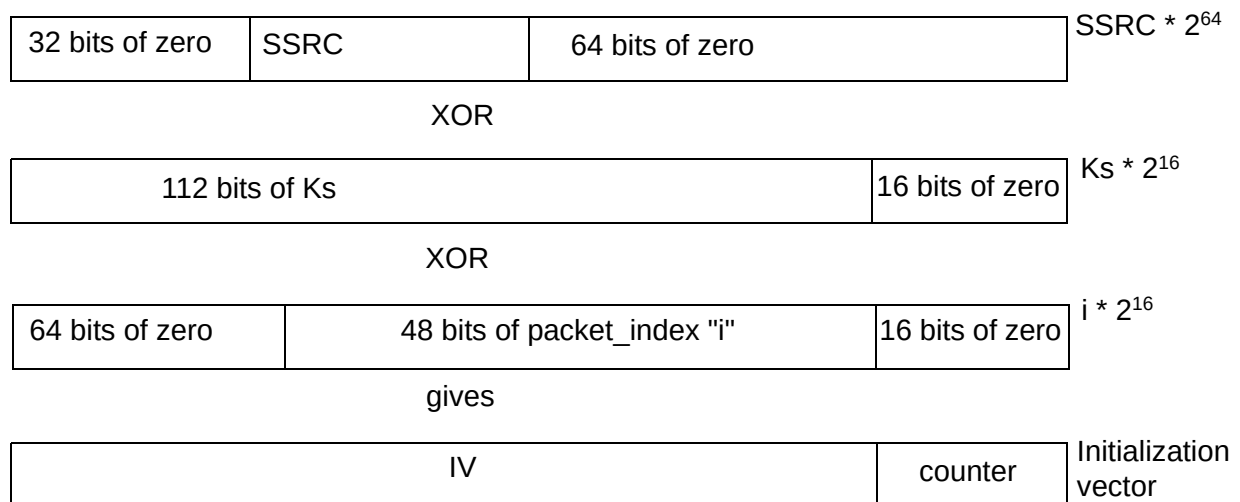
where KS is known as the session salting key and SSRC is the synchronization source from the RTP packet currently being encrypted. Note that "i" contains the packet sequence number SEQ and therefore **the IV must be recalculated for each RTP packet.**

The 112 bit KS is computed from x_s using the pseudo random function (PRF) as follows.

$$KS = \text{PRF}_{112}(\text{master_key}, x_s * 2^{16})$$

The pseudo random function is defined to be AES in counter mode with its output stream truncated as necessary (left most bits are retained).

The process for IV generation is shown in [Figure 7](#).

Figure 7: 128-bit IV Generation

4. Creating the Encryption Key KE

The process for generating the session key (KE) uses the pseudo random function (AES in counter mode) to produce a 128 bit value as follows: $KE = PRF_{128} (master_key, xE * 2^{16})$

Specifying the Devices' Encryption Capability

You may control whether your CMAPI softphone will support media encryption or not by specifying the supported encryption types in the local `MediaInfo` structure:

```
// specify either AES or no encryption for this device (let CM choose)

String [] encryption = {MediaConstants.AES, MediaConstants.NOENCRYPTION};

MediaInfo localMediaInfo = new MediaInfo();

localMediaInfo.setEncryptionList(encryption);

station.register (password, false, localMediaInfo, new MyAsyncRegistrationCallback());
```

This specifies that the CMAPI softphone will support both AES encryption and no media encryption. In this case, the decision to encrypt the media stream is left up to the Communication Manager (as specified in the CM's "change ip-codec" form).

Alternatively, you may force AES media encryption to be chosen by specifying a supported encryption type of only AES:

```
// must use AES encryption

String [] encryption = {MediaConstants.AES};
```

and, of course, you may force no encryption to be chosen by specifying:

```
// encryption not supported

String [] encryption = {MediaConstants.NOENCRYPTION};
```

MediaStartEvent Handling

If you have chosen to receive and handle the media stream as part of your application (you have chosen client-media mode), you will receive the media encryption information in the `MediaStartEvent`. In addition to the usual "RTP address", "RTP port", "codec" etc., the `MediaStartEvent` will also contain an "Encryption" object containing the encryption protocol and keys chosen by Communication Manager.

When you start to process the RTP stream, you need to pass the encryption information to your RTP stream read/write methods to enable them to do the media encryption/decryption. If you

are using Avaya's client-media-stack, you may call the Audio "start" method (as usual) passing the encryption information as an extra parameter:

```
MediaEncryption encryption = new MediaEncryption();
encryption.setProtocol(startEvent.getEncryption().getProtocol());
encryption.setTransmitKey(startEvent.getEncryption().getTransmitKey());
encryption.setReceiveKey(startEvent.getEncryption().getReceiveKey());
encryption.setPayloadType(startEvent.getEncryption().getPayloadType().intValue());
audio.start(rtpAddress, rtcpAddress, codec, packetSize, encryption);
```

Media Encryption Information

The Encryption object in the `MediaStartEvent` contains the following information:

- Encryption protocol
- Separate media encryption transmit and receive keys
- Payload type

For release 3.1, the encryption keys and the payload type are only required if the encryption protocol is "MediaConstants.AES".

For SDK compatibility reasons, the transmit and receive keys are formatted as Strings. For example, the transmit key may be in the form:

```
String transmitKey = "{38,4F,0B,34,DF,00,2A,BE,F0,C7,55,80,1D,1D,33,A8}"
```

and similarly for the receive key. The curly braces and commas are actually part of the String. In order to be used by the encryption/decryption routines, the keys need to be converted to byte arrays of the form (for example):

```
byte[] txKey = { 0x38, 0x4F, 0x0B, 0x34, 0xDF, 0x00, 0x2A, 0xBE, 0xF0, 0xC7, 0x55, 0x80, 0x1D, 0x1D, 0x33, 0xA8 };
```

If you use Avaya's client-media-stack, the Audio "start" method will automatically do the String to byte[] conversion for you.

Encrypting and Decrypting the RTP Stream

The encryption transmit and receive keys, along with the roll over counter (ROC) plus the RTP header's SSRC and sequence number, are used to calculate the Initialization Vector.

Roll Over Counter (ROC)

The ROC (initially set to zero) is a 32-bit unsigned integer which records how many times the 16-bit RTP sequence number (SEQ) has been reset to zero within the same SSRC (after

Writing a client application

incrementing up through 65,535) Unlike the sequence number (SEQ), which your secure RTP implementation (SRTP) extracts from the RTP packet header, the ROC is maintained by the header of each RTP packet. The ROC must also be knowledgeable of the SSRC that is included in the header of each RTP packet. The SSRC is a 32 bit randomly chosen value in an RTP packet that is used to represent the synchronization source (RFC1889). From one MediaStartEvent to the next MediaStopEvent, the SSRC will remain the same. If the SSRC changes this will be an indication that a new RTP stream has started. When this situation occurs the ROC must be reset to zero.

```
// Increment the ROC whenever the sequence number rolls over
incomingReadSSRC = rtpHdr.ssrc;
incomingSeqNum = rtpHdr.seqNum;
if (currentReadSSRC == incomingReadSSRC) {
    if (incomingSeqNum > currentSeqNum) {
        // Do nothing
    } else if (incomingSeqNum < (currentSeqNum - 100)) {
        // Sequence number has probably rolled over
        ++readROC;
    } else {
        // out of sequence RTP packet - ignore it
        return 0;
    }
} else if (incomingReadSSRC == prevReadSSRC) {
    // very late RTP packet from previous call - ignore it
    return 0;
} else {
    // New SSRC (i.e. new call) - reset ROC
    readROC = 0;
    prevReadSSRC = currentReadSSRC;
    currentReadSSRC = incomingReadSSRC;
}
currentSeqNum = incomingSeqNum;
```

and similarly for writeROC.

Creating the Encryption Keys Using the Pseudo Random Function

The pseudo random function PRF_n(key,x) produces a bit string of length “n” from a string “x” which is encrypted using the encryption key named “key”. The AES Symmetric algorithm mode is “ECB” with no padding.

```
private status final byte Xe[] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,}
// Set up and initialize JCE Engine
SymmetricEncryptionEngineIF see =
    SymmetricFactory.getInstance().createFactory(SymmetricFactory.AES);
see.initializeEngine("ECB", "NoPadding");
// Generate the symmetric key using the JCE Engine and the readMasterKey from the
// MediaStart event and then use the AVAYA XE value to calculate the Ke-Rx value
see.generateKey(readMasterKey);
Ke-Rx = see.encryptBlock((Xe);
```

And, similarly for Ke-Tx based on the writeMasterKey.

Once the media receive and transmit encryption keys (Ke-Rx and Ke-Tx) are created, they will be used within the AES algorithm to encrypt and decrypt the RTP stream.

Creating the Initialization Vectors (IV)

The ROC, together with the media encryption keys from the MediaStartEvent, the SSRC and the RTP header sequence number are used to calculate the IV for each direction (transmit & receive):

```
private static final byte Xs-Rx[] = {0,0,0,0,0,0,0,2,0,0,0,0,0,0,0,0};
byte[] Ks-Rx = new byte[14];
ByteBuffer ssrcBuffer = ByteBuffer.allocate(16);
ByteBuffer KsBuffer = ByteBuffer.allocate(16);
ByteBuffer iBuffer = ByteBuffer.allocate(16);

// Convert the RTP header sequence number to bytes
byte[] seqHex = convert2Bytes(seq);
// Clear the local 16-byte buffers used in the IV calculation
ssrcBuffer.clear();
KsBuffer.clear();
iBuffer.clear();

// Softphone PRF version
// Set up and initialize JCE Engine
SymmetricEncryptionEngineIF see =
    SymmetricFactory.getInstance().createFactory(SymmetricFactory.AES)
see.initializeEngine("ECB", "NoPadding");

// Generate the symmetric key using the JCE Engine and the readMasterKey from the
// MediaStart event and then use the Acaya Xs value to calculate the Ks-Rx value
see.generateKey(readMasterKey);
Ks-Rx = see.encryptBlock(Xs);
KsBuffer.put(Ks-Rx, 0, Ks-Rx.length-2);
```

And, similarly for Ks-Tx based on the writeMasterKey

```
// Grab the SSRC from the RTP header and populate the SSRCbuffer
ssrcBuffer.position(4);
ssrcBuffer.putInt(ssrc);

// Setup and populate the iBuffer - this will currently
// make the roll over counter to be zero always
iBuffer.position(8);
iBuffer.putInt(readROC);
iBuffer.put(seqHex[2]);
iBuffer.put(seqHex[3]);

byte[] ssrcBytes = ssrcBuffer.array();
byte[] ksBytes = KsBuffer.array();
byte[] iBytes = iBuffer.array();
byte[] iv-Rx = new byte [16];
iBuffer.put(seqHex[2]);
iBuffer.put(seqHex[3]);

// XOR all 3 buffers
```

Writing a client application

```
for (int ii = 0; ii < iv-Rx.length; ii++) {  
    iv-Rx[ii] = (byte)(ssrcBytes[ii] ^ ksBytes[ii] ^ iBytes[ii]);
```

and similarly for calculating the write buffer IV (iv-Tx).

Decrypting the Media Payload

In the draft SRTP specification, the encryption algorithm is defined as AES in counter mode (CTR and NoPadding). The generated IVs, along with the Ke-Rx or Ke-Tx, can be used to secure the RTP stream during each transmit and receive operation.

```
// get the RTP packet from the read socket  
ByteBuffer dst = rtpPacket.getPacket();  
  
// Point to payload start and obtain the encrypted payload  
int hLength = 12; (RTP header size)  
pLength = dst.limit() - hLength; (RTP payload size)  
  
dst.position(hLength);  
byte[] buf2 = new byte[pLength];  
dst.get(buf2, 0, pLength);  
  
// Decrypt payload  
byte[] buf3 = decryptEngine.decryptBlock(buf2, iv-Rx, Ke-Rx);
```

The decryptBlock will look something like the following:

```
// decryptBlock sample code  
Public byte[] decryptBlock(buf2, iv-Rx, Ke-Rx){  
    // Set up cipher engine  
    SymmetricEncryptionEngineIF see =  
SymmetricFactory.getInstance().createFactory(SymmetricFactory.AES);  
    see.initializeEngine("CTR", "NoPadding");  
  
    // Pass to JCE etc. . . .  
    see.decrypt(buf2, iv-Rx, Ke-Rx);  
}
```

Test Data

In order to validate your code, we present here some test data against which you may run your decipher code. Following that is the expected output.

- Input Data

```
// From the MediaStart event, we get  
byte[] readMasterKey = { (byte)0xaa,(byte)0xaa,(byte)0xaa,(byte)0xaa,  
                        (byte)0xaa,(byte)0xaa,(byte)0xaa,(byte)0xaa,  
                        (byte)0xaa,(byte)0xaa,(byte)0xaa,(byte)0xaa,  
                        (byte)0xaa,(byte)0xaa,(byte)0xaa,(byte)0xaa  
};
```

```
// From the RTP packet, we get
int ssrc = 987011809;
int seq = 3;
int roc = 0;

// And the data to decipher is:
cipherData = {(byte)0xbb,(byte)0xbb,(byte)0xbb,(byte)0xbb };
```

● Expected Output

```
Ke-Rx:
(byte)0xba, (byte)0xeb, (byte)0xc6, (byte)0x18, (byte)0xa5, (byte)0x5c, (byte)0x35,
(byte)0x1f, (byte)0x25, (byte)0xce, (byte)0xdf, (byte)0x37, (byte)0xbf, (byte)0x70,
(byte)0xf3, (byte)0x90
Ks-Rx:
(byte)0x98, (byte)0x12, (byte)0xf4, (byte)0x3c, (byte)0x17, (byte)0xc5, (byte)0xd4,
(byte)0x0e, (byte)0xe3, (byte)0x8f, (byte)0x09, (byte)0xe1, (byte)0x7f, (byte)0xa8,
(byte)0xba, (byte)0xb7
IV-Rx:
(byte)0x98, (byte)0x12, (byte)0xf4, (byte)0x3c, (byte)0x2d, (byte)0x11, (byte)0x4e,
(byte)0xef, (byte)0xe3, (byte)0x8f, (byte)0x09, (byte)0xe1, (byte)0x7f, (byte)0xab,
(byte)0x00, (byte)0x00

// And the deciphered data should be:
plainData = {(byte)0x05, (byte)0x43, (byte)0x2a, (byte)0x3d };
```

Security considerations

Your application development organization has the responsibility of providing the appropriate amount of security for your particular application and/or recommending appropriate security measures to your application customers for the deployment of your application. Therefore you should be aware of the security measures that AE Services Device, Media and Call Control API already takes and what risks are known.

AE Services Device, Media and Call Control API provides these security measures:

- The station password is required to register a device.
- Filenames specified for recorded files must be relative to the configured directory, their directories must already exist, and recordings cannot overwrite an existing file.
- Only files within the configured recorder directory can be deleted using the `Media.DeleteMessage()` method.

Application Enablement Services offers the user the ability to encrypt the voice RTP streams between the IP softphone and the far end of the call. See [Media Encryption](#) on page 79 for more information.

Writing a client application

If you are using encryption, AE Services Device, Media and Call Control API provides these additional security measures:

- The signaling and bearer channels are encrypted.
- XML messages transmitted between the client application and the AE Services server software are encrypted.
- The station password is passed encrypted.
- Username and password are encrypted.
- Username and password are authenticated by the Device, Media and Call Control service.



WARNING:

If you do not use encryption on the client link, the signaling and bearer channels are unencrypted, there is no encryption of XML messages transmitted between the client application and the AE Services server software, the station password is passed unencrypted and the username and password are sent in the clear.

For a complete discussion of the security guidelines for AE Services, see *White-paper on Security in Application Enablement Services for Bundled and Software only solutions*. This white paper is available on the Avaya support site along with the customer documents.

Chapter 4: Debugging

This chapter describes:

- [Debugging](#)
- [Getting support](#)

Debugging

This section assumes that you have verified the operation of your connector server using the verification application that comes with the connector server RPM file. Please refer to the appropriate *Avaya MultiVantage™ Application Enablement Services Installation and Upgrade Guide* for the offer you have purchased (bundled server or software only) for troubleshooting procedures if this verification application does not function properly.

In debugging your application, you will rely on:

1. **Error Messages** returned in response events. Every request you make to the .NET API which results in a message being sent to the AE Services server should result in a response message coming back. You should always have an event handler for every message you send and when you get the event, verify that the error attribute of the response is an empty string. If it is not an empty string, the error attribute will contain information on why the response failed.

You should always catch exceptions when invoking methods in the API. The most common exception is trying to send a message to the AE Services server after the socket has been closed.

The .Net SDK ships with a **DMCC.config** file. This file is the configuration file for the lognet error logger. By changing the log levels you can enable all XML messages sent to/from your application and the AE Services server, to be logged to a file. In addition, you can also enable logging of internal error conditions that may have occurred in the API. The **DMCC.config** file has comments in it instructing you how to do this. In order for your application to load and use the **DMCC.config** file the file must be located in the same folder as your applications .exe file. The **DMCC.config** file also has a logger just for the dashboard and information on how to enable it is in the file.

If you want to use the Device, Media and Call Control's .NET **log4net** capabilities in your application, you need to add the following code to your application:

```
private static DMCC_log4net.ILog Logger =  
DMCC_log4net.LogManager.GetLogger("NAME_OF_YOUR_LOGGER");
```

Where NAME_OF_YOUR_LOGGER is how you want you logger identified in the **DMCC.config** file.

Then in the **DMCC.config** file you need to add the following:

```
<logger name="NAME_OF_YOUR_LOGGER">
  <!-- <appender-ref ref="B" /> -->
  <level value="DEBUG" />
  <appender-ref ref="RollingLogFileAppender"/>
  <appender-ref ref="ConsoleAppender" />
</logger>
```

The remainder of this section helps you with:

- [Possible race conditions](#) on page 92
- [Improving performance](#) on page 92

Possible race conditions

- If a thread is playing/recording a message on a phone and another thread attempts to play/record a message to the same phone then the second attempt will fail.
- If an application is using the Media object to request that playing/recording terminate when a specified DTMF tone is detected AND is also configured to listen for DTMF tones then there is no guarantee of the order for the Play/Recording Stopped Event and the Tone Detected Event.

Improving performance

Many different factors may potentially affect the performance of your system. The system has three main parts that may be affected:

- The AE Services server
- Communication Manager
- The network

An excessive load on any of these may slow down the overall system. Here are other factors to check that also may affect your system performance.

On the AE Services server:

- Ensure that your AE Services server machine meets the minimum requirements specified in the appropriate *Avaya MultiVantage™ Application Enablement Services Installation and Upgrade Guide* for the offer you have purchased (bundled server or software only).
- Avoid running any other applications on the AE Services server machine.
- Check that the AE Services server's Linux operating system resources are tuned correctly for your application needs. The server software makes no assumptions concerning your application needs and therefore does not tune the kernel for you. See the Linux documentation found at <http://www.redhat.com/docs/manuals/linux/RHL-9-Manual>.

- Update the Linux kernel with the latest updates available.

On Communication Manager:

- Ensure that Communication Manager is properly configured for your network and business needs. Misconfigured Communication Manager elements can lead to performance issues.

On the network:

- Ensure that your network traffic is properly balanced. One way to do this professionally is to ask Avaya to perform a network assessment. There is also a *VoIP Readiness Guide* available from the Avaya Support Centre (<http://www.avaya.com/support>). For more information about improving the performance of your network, see the “Network Quality and Management” section of *Administration for Network Connectivity for Avaya Communication Manager* (555-233-504).

Getting support

Development support is only available through Avaya's DevConnect Program at this time. As an Innovator/Premier/Strategic level member of the DevConnect Program, technical support questions can be answered through the DevConnect Portal at www.avaya.com/devconnect

As a Registered member of the program, support is not available. If you require support as a Registered member, you can apply for a higher level of membership that offers support and testing opportunities through the DevConnect Portal. Membership at the Innovator/Premier/Strategic level is not open to all companies. Avaya determines membership status above the Registered level.

The DevConnect web site offers discussion forums for a variety of Device, Media and Call Control areas. There is one that is dedicated to the .NET API. If you have a question regarding the .NET API, feel free to post the question on the forum. Before posting, please look at the other topics which have already been posted to see if your question has already been asked and answered.

Appendix A: The J-Scripts Library

Application Enablement (AE) Services Device, Media and Call Control .NET API offers a thin client prototype which uses a modified .NET SDK inside Internet Explorer. The modified .NET SDK – call it the *ActiveX ServiceProvider* – combines into one class all of the methods of all of the classes of the unmodified SDK. In order to use the *ActiveX ServiceProvider*, a web application has to interact with it using Jscript.

The purpose of this appendix is to define the interface and implementation of a JScript library with which you can develop web applications using the *ActiveX ServiceProvider*.

In this interface the method signatures are almost identical to the methods of the .NET SDK, making this interface general-purpose and easy to use.

J-Script

We have developed the Jscript library with the assumption that it will be imported into HTML documents by way of the src attribute of a script element, as in:

```
<script src="../../sp.js" type="text/Jscript"></script>
```

This helps maintain a strict separation of concerns; our Jscript library publishes an interface and the developer writes code using that interface, all in separate files. It simplifies your HTML files by allowing you to remove large blocks of JavaScript code from them – that is, it helps keep content and behavior separate.

We will also use namespaces for each of the Jscript objects (not classes) described below. Namespaces prevent name collisions and, when used with an anonymous function can “seal” a JavaScript object so that its private properties are invisible outside of the scope of the object. The following code is an example of creating `com.avaya.dmcc` and populating it with a constructor:

```
var com;
if (!com){
    com = {}
} else if (typeof com != object) {
    throw new Error("com is not an object")
}
if (!com.avaya) {
    com.avaya = {}
} else if (typeof com.avaya != object) {
    throw new Error("com.avaya is not an object")
}
if (!com.avaya.dmcc) {
    com.avaya.dmcc = {}
} else if (typeof com.avaya.dmcc != object) {
    throw new Error("com.avaya.dmcc is not an object")
}
/* Begin anonymous function */
(function() {
```

```
function ServiceProvider() { /* empty */ }
ServiceProvider.prototype.startApplicationSession =
    function(/* args list */) {
        /* body of function */
    }
ServiceProvider.prototype.stopApplicationSession =
    function(/* args list */) {
        /* body of function */
    }
/* Populate the namespace */
com.avaya.dmcc.ServiceProvider = ServiceProvider
})(); // End anonymous function and invoke it
sp = new com.avaya.dmcc.ServiceProvider();
sp.startApplicationSession();
```

Note:

JavaScript is an object- or prototype-based object-oriented language. We saw an example of prototypes occurs in the code above, where the instance functions `startApplicationSession` and `stopApplicationSession` are created using the prototype for the `ServiceProvider` object.

ActiveX Service Provider

The implementation of this project depends on the completion of a proxy class that will allow the .NET SDK to be used inside Internet Explorer. Its methods will be annotated using Microsoft Interop Services annotations to make them COM-visible. This in turn will allow us to invoke those methods using Jscript. The web application's Jscript depends on the Jscript library, which depends on *ActiveX ServiceProvider*. By association, *ActiveX ServiceProvider* has a dependency on the unadulterated .NET *ServiceProvider*.

To create an instance of the *ActiveX ServiceProvider* inside Internet Explorer, the web application developer will need to include an HTML object element in their web page(s). The Jscript library assumes that the value of the element's `id` attribute is **DMCC**; it will not function if the object element is missing from the page, the `id` attribute is missing from the object element, or the `id` attribute has a different value. The element looks like this:

```
<object id="DMCC" . . . ></object>
```

Also, the Jscript library will not function if the *ActiveX ServiceProvider* object has not been instantiated, so the object element must appear before the script element in the HTML document.

Static event handlers

It is not possible to instantiate COM objects dynamically (using `new`) in Jscript when you need to be able to handle events propagated through Interop Services.

JavaScript and Jscript natively handle asynchronous browser events, such as page loads, button clicks, and so on, but those events originate in the browser or from the user. Consequently, we must statically define handlers for events in the Jscript library. Using the

Jscript extension to JavaScript for handling events via Interop Services, the signature of a static handler looks like:

```
function DMCC::MediaStartedEvent(  
    deviceId, monitorId,  
    rtpAddress, rtpPort,  
    rtcpAddress, rtcpPort, packetSize)
```

Since only one static handler exists for a event, the static handler will be responsible for dispatching events and response events to the object to which they ultimately belong.

Basically, the declaration of this function binds the function to a C# delegate in the *ActiveX ServiceProvider*. Note that the prefix of the function name – **DMCC::** – refers to the value of the *id* attribute of the object element in the web page. This binds the function *mediaStartedEvent* to the instance of the *ActiveX ServiceProvider* instantiated by Internet Explorer and acts as a proxy between the *ActiveX ServiceProvider* and the Jscript callback provided by the web application developer.

Static handlers will be defined for each event and each response event.

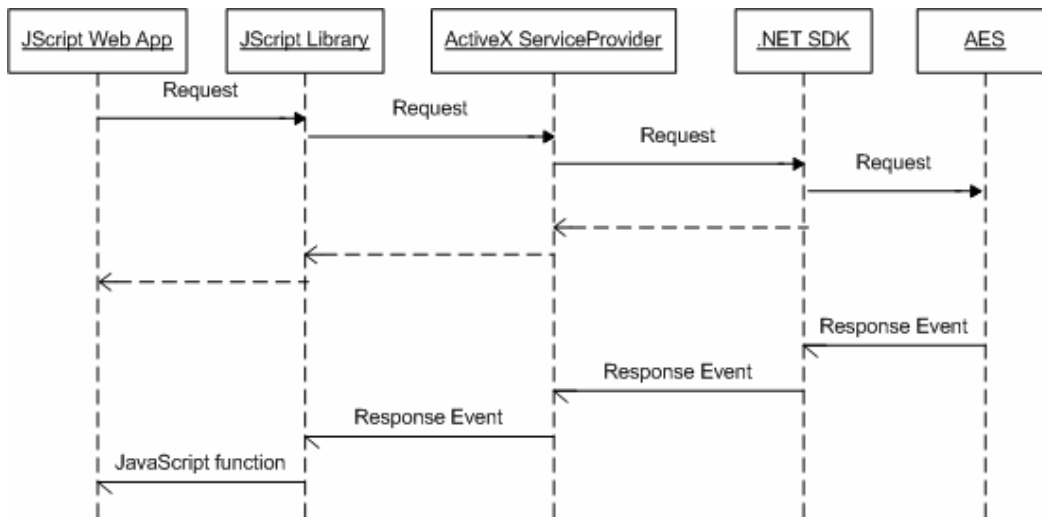
Requests, response events, and events

In the implementation of the Jscript library, requests are non-blocking, and a response to a request is considered a response event. A response event is similar to but slightly different than a regular event.

Consider a request to register a terminal. When invoking *registerTerminal*: the developer passes in the usual arguments (*media mode*, *codec*, etc.) as well as a function that the library will call when the response event occurs.

The Jscript library forwards the parameters to the corresponding method in the *ActiveX ServiceProvider* (without the developer-provided function), which in turn, creates an *Invoke ID*, converts the request to XML, sends it to the server and returns the *Invoke ID* to the Jscript library. The Jscript library places the *Invoke ID* along with the developer-provided function into a hash. A response event occurs when the server's response is received, at which time the *ActiveX ServiceProvider* invokes the static handler, *RegisterTerminalResponseEvent*, passing it the *Device ID* string, the *Invoke ID*, and some other parameters. Using the *Invoke ID*, the static handler retrieves the developer –provided function, deletes it from the hash, and invokes it with all of the arguments that were passed to itself.

[A Request and Subsequent Response Event](#) illustrates this process.

Figure 8: A Request and Subsequent Response Event

This process is similar for regular events. To register interest in an event, the developer calls `startMonitor()` using a reference to a Phone, Media, CallControl, or CallInformation object. Instead of passing in a boolean value to indicate interest in a particular type of event, however, the developer will pass in a function to be called when an event of that type occurs.

The primary differences between response events and regular events are:

- an *Invoke ID* is not returned when the developer calls `startMonitor()`
- the function that the developer provides for an event type remains in the library's internal hash for that event type until it is deleted when the developer calls `stopMonitor()`.

The normal value of the *Invoke ID* is in correlating responses with requests. The function-based approach just described manages this for the developer, so the value of the *Invoke ID* to the user is unknown. Each request method in the Jscript library will nonetheless return the *Invoke ID*, should the user wish to do something with it.

The following code are examples of registering for and handling request events and events.

When the `StartApplicationSessionRequestEvent` occurs, the function `sasCallback` is invoked.

```

sp = new com.avaya.dmcc.ServiceProvider();
sasCallback = function(error, protoVersion, sessionDuration, invokeId) {
    /* web application code */
}
sp.startApplicationSession(/* normal argument list */, sasCallback);
  
```

Similarly, when the Phone's `StartMonitorResponseEvent` occurs, the Jscript library invokes the web application's `responseFunc`, and when a `DisplayUpdatedEvent` occurs it invokes the web application's `displayUpdatedHandler`. If the developer wishes to register interest in other events, they will assign a handler to the other properties of the Phone Events

object. The names of the properties of the Jscript Phone Events object will be the same as the corresponding Boolean fields of the C# Phone Events class, except in camelCase.

```
responseFunc = function(invokedId, deviceId, monitorId, userState, error) {  
    /* web application code */  
}  
displayUpdatedHandler = function(deviceId, displayContents) {  
    /* web application code */  
}  
events = new PhoneEvents();  
events.displayUpdated = displayUpdatedHandler;  
phone.startMonitor(events, 0, responseFunc)
```

Figure 5: Registering an Event Handler

J-Script object

The Jscript library will have objects with the following names: ServiceProvider, Device, Phone, Media, ThirdPartyCallController, and CallInformation.

Note:

An XMLProcessor object is not included in this design, but can be added if requested.

The following tables show the methods and events belonging to each object. The methods that do not take a callback function as the final argument are *italicized*.

The names follow the JavaScript naming conventions:

- Pascal case for types
- camel case for identifiers and methods.

An object will not have a method named after an event. The object will have a method for registering a function (which is anonymous from the perspective of the Jscript library) to handle events of that type.

Note:

If this list is not exhaustive, any additional methods or events not shown here will be added when they are added to the *ActiveX ServiceProvider*.

Service Provider Methods and Events

Table 18: Service Provider

Methods	Events
startApplicationSession	serverConnectionDownEvent
stopApplicationSession	serverConnectionNotActiveEvent

Table 18: Service Provider

Methods	Events
reconnect	missedAtLeastOneKeepAliveEvent
getNewDevice	
startAutoKeepAlive	
stopAutoKeepAlive	
shutdown	
getSessionId	
getAutoKeepAliveEnabled	
getDeviceId	
releaseDeviceId	
getThirdPartyDeviceId	

Device Methods and Events

Table 19: Device

Methods	Events
startMonitor	displayUpdatedEvent
stopMonitor	lampUpdatedEvent
registerTerminal	ringerStatusEvent
unregisterTerminal	terminalUnregisteredEvent
redirectMedia	hookSwitchUpdatedEvent
getHookswitchStatus	
setHookswitchStatus	
pressButton	
getButtonInfo	
getDisplay	

Table 19: Device

Methods	Events
getLampMode	
getMessageWaitingIndicator	
getRingerStatus	

Phone Methods and Events

Table 20: Phone

Methods	Events
startMonitor	mediaStartedEvent
stopMonitor	mediaStoppedEvent
mediaDeleteMessage	mediaPlayingEvent
mediaSetToneRetrievalCriteria	mediaPlayingStoppedEvent
mediaStartDubbing	mediaPlayingSuspendedEvent
mediaStopDubbing	mediaRecordingEvent
mediaStartRecording	mediaRecordingStoppedEvent
mediaStopRecording	mediaRecordingSuspendedEvent
mediaSuspendRecording	mediaToneDetectedEvent
mediaResumeRecording	mediaTonesRetrievedEvent
mediaStartPlaying	
mediaSuspendPlaying	
mediaResumePlaying	
mediaStartToneCollection	
mediaStopToneCollection	
mediaFlushToneCollectionBuffer	

Media Methods and Events

Table 21: Media

Methods	Events
startMonitor	conferencedEvent
stopMonitor	deliveredEvent
snapshotDevice	connectionClearedEvent
singleStepConferenceCall	doNotDisturbEvent
alternateCall	establishedEvent
answerCall	failedEvent
clearCall	forwardingEvent
clearConnection	heldEvent
conferenceCall	originatedEvent
consultationCall	retrievedEvent
deflectCall	transferredEvent
generateDigits	
holdCall	
makeCall	
getDoNotDisturb	
getForwarding	
reconnectCall	
retrieveCall	
setDisplay	
setDoNotDisturb	
setForwarding	
singleStepTransfer	
snapshotCall	
transferCall	

Third Party Call Controller Methods and Events

Table 22: Third Party Call Controller

Methods	Events
startMonitor	linkUpEvent
stopMonitor	linkDownEvent
getCallInformation	
getLinkStatus	

Call Information Methods and Events

Table 23: Call Information

Methods	Events

J-Script library configuration

All JScript-related things are stored in the .NET SDK zip file in dmcc-dotnet-sdk/JScript. Inside that are three folders:

- dashboard
containing all the code of the JScript dashboard
- doc
containing the simple reference documentation for the JScript library
- src
containing the JScript library code

Server-side Instructions

Modify the object element that appears near the top of **dashboard.html**.

In particular, you must:

- Change `host.example.com` to the name of the machine on which the web server is running.
- Change `path/to` to the path to the directory containing `ServiceProvider.dll`. (`ServiceProvider.dll` must reside in the same directory on your web server as **dashboard.html**.)
- Do not change the value of the object element's `id` attribute; doing so will cause the dashboard and any DMCC JScript library-based application to fail to function.

Client-side Instructions

In order for the dashboard page to function in Internet Explorer, you must do the following:

- Start a DOS window
- Go to **C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727** (or whatever 2.x .NET framework is installed)
- Run `.\caspol.exe -lg`, noting the 2 digit Zone number for LocalIntranet (e.g., 1.2), hereafter referred to as *Z.N*.
- Run `.\caspol.exe -m -ag Z.N -url http://host.example.com/* FullTrust`, where *Z.N* is the two digit zone number from the previous step and `host.example.com` is the name of the machine on which the dashboard page resides. This is necessary because the .NET control opens sockets.
- In Internet Explorer go to **Tools --> Internet Options --> Security**
- Select **Local Intranet**
- Click **Sites**
- Select **Advanced**
- Enter `http://host.example.com`
- Select **Add**

You may also have to configure additional security settings in Internet Explorer. They can be set by going to **Tools --> Internet Options --> Security--> Local Intranet --> Custom Level**.

The setting that affect whether or how this page (including `ServiceProvider.dll`) functions are:

1. Run components signed with Authenticode
2. Download signed ActiveX controls
3. Run ActiveX controls and plugins
4. Script ActiveX controls marked safe for scripting

The dashboard page has only been tested with Internet Explorer 6 on Windows XP.

Appendix B: Communication Manager Features

Here is a list of key features in Communication Manager that you may wish to take advantage of in developing your applications. This is not an exhaustive list - just a subset of features most likely to be used in applications. For descriptions of these features, see any of the Communication Manager administration guides.

- AAR/ARS Partitioning
- Abbreviated Dialing
- Abbreviated and Delayed Ringing
- Abbreviated Programming
- Administration Without Hardware
- Authorization Codes
- Automatic Alternate Routing (AAR)
- Automatic Call Distribution (ACD) Features
 - Announcements
 - Automatic Answering With Zip Tone
 - Multiple Call Handling
 - Service Observing
 - Observe Digital Sets/IP Phones
 - Observe Logical Agent IDs
- Automatic Call Distribution (ACD) Features:
 - Basic Hunt Group Call
 - Agents in Multiple Splits
 - Agent Login/Out
 - Display - Agent Terminal
- Automatic Callback (on Busy)
- Automatic Callback on Don't Answer
- Automatic Route Selection (ARS)
- Bridged Call Appearance (single-line, multi-appearance)
 - Hunt Group Redirect Coverage
 - Multiple Coverage Paths
 - Temporary Bridged Appearance
 - Remove Temporary Bridged Appearance

Communication Manager Features

- Call Coverage Features
 - Call Coverage Consult with Conference/Transfer
- Call Coverage Features
 - Consult
 - Coverage Paths
 - Send All Calls
 - Temporary Bridged Appearance
- Call Forwarding / Busy Don't Answer @ Call Vectoring
 - VDN of Origin Announcements
- Call Forwarding by Service Observer
- Call Forwarding by Service Observed
- Call Forwarding Features:
 - Call Forwarding All Calls
 - Call Forwarding - Busy and Don't Answer
 - Call Forwarding - Don't Answer
 - Call Forwarding - Off Net
- Call Park
- Call Pickup
 - Directed Call Pickup
 - Remove Temporary Bridged Appearance
 - Remove Auto-Intercom
- Call Vectoring:
 - VDN of Origin Display
- Consult
- Coverage of Calls Redirected Off Net
- Drop (button operation)
- Group Paging
- Hold (single-line, multi-appearance)
- Hold - Automatic
- Hold (single-line, multi-appearance)
- Hold - Automatic [from IR1V4 WCC]
- Hunting/Hunt Groups

- IP Trunks
- Last Number Dialed
- Leave Word Calling - Switch
- Malicious Call Trace
- Message Waiting Indication
- Music-on-Hold Access
 - Held Calls
 - Conference-Terminal Calls
 - Transferred Trunk Calls
- Personalized Ringing
- Personal Station Access
- Priority Calling
- Recorded Announcement
- Terminal Translation Initialization (TTI)
- Tone on Hold
- Transfer (single-line, multi-appearance)
- Voice Terminal Display
 - Calling Number Display (SID/ANI/Extn ID)
 - Called Number Display (internal & DCS)
 - Stored Button Display

Appendix C: Constant Values

Here are tables describing the values for the XML message parameters which take a constant value and that are switch specific. These values are specific to Avaya's Communication Manager switch. These include:

- Phone Object Enumerations and Constants
 - Lamp State - enumeration which defines all of the lamp states
 - Lamp Color - enumeration which defines all of the lamp colors
 - Button Function - enumeration which defines all of the button functions
 - Button ID - enumeration which defines all of the button ID's
 - Registration States - enumeration which defines all of the registration states
 - Dependency Modes - enumeration which defines all of the dependency modes
 - Ringer Pattern Constants
 - Registration Constants
- Media Mode Enumerations
 - Media Mode - enumeration which defines all of the media modes
- Service Provider Enumerations
 - Convert E164 to Dial String - enumeration which defines
 - Convert Dial String to E164 - enumeration which defines
- Third Party Call Control Enumeration
 - Single Step Conference Call - enumeration which defines all of the Single Step Conference Call states
 - Call Forwarding - enumeration which defines all of the Call Forwarding states

Phone Object Enumerations and Constants

Table 24: Lamp Mode Enumeration

Name	Description
Broken Flutter	Usually indicates a feature activation has been denied.
Flutter	Usually indicates a feature activation has been denied or a button with no feature associated with it has been pushed.

Table 24: Lamp Mode Enumeration

Name	Description
Off	Lamp is off.
Steady	Lamp is on.
Wink	Lamp is winking (fast blink). Usually indicates a feature is pending activation.
Inverted Wink	Lamp is winking in reverse. Usually indicates some kind of denial.
Flash	Lamp is flashing. Usually indicates a call is on hold or a feature is pending activation.
Inverted Flash	Lamp is flashing in reverse. Usually indicates some kind of denial.
Undefined	Some other lamp state. You should never get this.

Table 25: Lamp Color Enumeration

Name	Description
Red	Red lamp
Green	Green lamp
Undefined	Some other color. You should never get this

Table 26: Button Function Enumeration

Name	Description
ABR_PROG	Abbreviated Dial Program
ABR_SPCHAR	AD Special Character
ABRDG_APPR	Analog Bridged Appearance
ABRV_DIAL	Abbreviated Dial
ABRV_RING	Button to force Abbreviated / Delayed Ring Transition
AC_ALARM	Administered Connection alarm button
1 of 8	

Table 26: Button Function Enumeration

Name	Description
ACA_HALT	ACA referral call activate button
ACCOUNT	CDR Account Code Button
ADMIN	Enter Terminal Self-Admin Mode
AFTER_CALL	ACD - After Call Work
ALRT_AGCHG	Move agent while staffed alert
ALT_FRL	Alternate FRL button
ANI_REQUEST	Incoming ANI Request (Russian trunks only)
ASSIST	ACD - supervisor assist button
ASVN_HALT	SVN Auth Code Halt
ATD_QCALLS	Attended group - number of queued calls
ATD_QTIME	Attended group - oldest time
AUDIX_REC	Audix One-Step Recording
AUT_MSG_WT	Automatic message waiting indication
AUTO_CBACK	Auo call back
AUTO_ICOM	Auto-dial ICOM
AUTO_IN	ACD - Auto-In
AUTO_WKUP	Automatic Wakeup entry mode
AUTODIAL	Autodial button
AUX_WOR	
BRDG_APPR	Bridged appearance of primary extension
BTN_RING	Button Ring Control entry mode
BTN_VIEW	Enhanced View Button
BUSY_IND	Station busy indicator
CALL_APPR	General call appearance, no aux data
CALL_DISP	Call the displayed number
CALL_FWD	Call forwarding button
2 of 8	

Table 26: Button Function Enumeration

Name	Description
CALL_PARK	Park/Unpark button
CALL_PKUP	UM call pickup
CALL_SCRN	
CALL_TIMER	Call Timer
CALLR_INFO	Caller Information
CAS_BACKUP	CAS (branch) back-up mode lamp
CDR1_ALARM	SMDR primary printer alarm
CDR2_ALARM	SMDR secondary printer alarm
CFWD_BSYDA	Call Forward/Busy Don't answer
CFWD_ENH	Check-in entry mode
CHECK_IN	Check-in entry mode
CHECK_OUT	Check-out entry mode
CLK_OVERID	Clocked override-time of day routing
CONFERENCE	Conference
CONF_DSP	Display conference parties
CONSULT	Consult/return
COV_CBACK	Coverage LWC call back
COV_MSG_RT	Coverage retrieve LWC message
CPN_BLK	Per call CPN blocking activate
CPN_UNBLK	Per call CPN unblocking activate
CRSS_ALERT	Crisis Alert to Digital Station or Attendant.
DATA_EXT	Data extension button
DATE_TIME	TOD/DATE display mode
DELETE_MSG	Delete LWC message
DIAL_ICOM	Autodial, aux is uid of destination
DID_REMOVE	DID remove entry mode
3 of 8	

Table 26: Button Function Enumeration

Name	Description
DID_VIEW	DID view entry mode
DIR_PKUP	Directed call pickup
DIRECTORY	Directory listing for name search
DISP_CHRG	Display Charge button
DISP_NORM	NORMAL/LOCAL mode button
DN_DST	User do not disturb button
DROP	Drop the last party on a conference call
EC500	EC500 Feature plus timer
EXCLUSION	Manual exclusion
EXT_DN_DST	Do not disturb - ext.
EXTND_CALL	OPTIM Extend call
FE_MUTE	Conf Select Far End Mute
FLASH	Flash button for station on CAS MAIN call OR a call using Trunk Flash
GOTO_COVER	Goto coverage button
GRP_DN_DST	Do not disturb - grp.
GRP_PAGE	Group Paging button
HEADSET	Headset in use
HOLD	Hold
HUNT_NS	Hunt night service button
IN_CALL_ID	
INSPECT	Inspect display mode
INT_AUT_AN	Internal Automatic Answer button
LAST_NUMB	Last number dialed
LIC_ERROR	License error
LIMIT_CALL	
4 of 8	

Table 26: Button Function Enumeration

Name	Description
LINK_ALARM	Link alarm button
LOGOUT_OVR	SVN login security violation
LSVN_HALT	
LWC_CANCEL	Cancel LWC
LWC_LOCK	Lockout LWC
LWC_STORE	LWC store message
MAJOR_ALRM	Major alarm button
MAN_MSG_WT	Manual message waiting button
MAN_OVERRIDE	Manual override-time of day routing
MANUAL_IN	ACD - Manual-In
MCT_ACT	MCT: Malicious Call Trace Activate
MCT_CONTR	MCT: Malicious Call Trace Control
MF_DA_INTL	International directory assistance - Mexico
MF_OP_INTL	International CO operator - Mexico
MJ_MN_ALRM	Major/Minor alarm button
MWI	Message Waiting Indicator
MM_BASIC	Multimedia Voice/Data activate/deactivate
MM_CALL	Multimedia Call Mode activate
MM_CFWD	Multimedia Call Forward
MM_DATAACNF	Multimedia Data Conference activate
MM_MULTNBR	Multimedia Multi Address activate
MM_PCAUDIO	Multimedia PC-Audio activate
MMI_CP_ALM	MMI pack or port (video) alarm
MSG_RETR	Principal retrieve LWC message
MWN_ACT	Message notification on mode
MWN_DEACT	Message notification off mode
5 of 8	

Table 26: Button Function Enumeration

Name	Description
NEXT	Step through LWC message
NIGHT_SERV	Night Activate
NO_HLD_CNF	No hold conference
NH_CONSULT	
NOANS_ALRT	No Answer Alert button for Redirect On No Answer (RONA) timeout for split.
NORMAL	Normal display mode
OFF_BD_ALM	Off-board DS1 board alarm button
PER_COLINE	Personal CO line, aux is grp id
PG1_ALARM	SA8312 PAGE1 alarm
PG2_ALARM	SA8312 PAGE2 alarm
PMS_ALARM	PMS alarm button
POST_MSGS	Posted messages
PR_AWU_ALM	Wakeup journal printer alarm
PR_PMS_ALM	PMS journal printer alarm
PR_SYS_ALM	BCMS printer link alarm button
PRINT_MSGS	
PRIORITY	Priority calling button
Q_CALLS	Hunt group - number of queued calls
Q_TIME	Hunt group - oldest queued time
RELEASE	ACD - Release
RING_STAT	Ringer status
RINGER_OFF	Ringer cut button for stations
RS_ALERT	System reset alert
RSVN_HALT	SVN remote access security violation
SCROLL	SCROLL mode button
6 of 8	

Table 26: Button Function Enumeration

Name	Description
SEND_CALLS	Send all calls button
SEND_TERM	Terminating extension group SAC button
SERV_OBSRV	Service observing button
SIGNAL	Manual signalling
SHARE_TALK	
SSVN_HALT	SVN station security call activate button
STA_LOCK	Station Lock
START_BILL	Start Billing
STORED_NUM	STORED-NUMBER display mode
STROKE_CNT	Single-Digit Stroke Counts
TEAM	
TERM_X_GR	
TIMER	ELAPSED-TIME display mode
TOGGLE_SWAP	Conf/Transfer Toggle Swap
TRANSFER	Transfer
TRK_AC_ALM	Facility acc test trunk access alert
TRK_ID	Trunk ID button
TRUNK_NAME	Trunk name when DCS (also DCS CAS MN)
TRUNK_NS	Trunk night service button
USR_ADDBSY	Add FBI to station
USR_REMBSY	Remove FBI from station
UUI_INFO	UUI-info button
VC_CP_ALM	VC pack or port (audio) alarm
VERIFY	Busy verification button
VIP_CHKIN	VIP check-in
VIP_RETRY	Reschedule VIP wakeup as regular wakeup
7 of 8	

Table 26: Button Function Enumeration

Name	Description
VIP_WAKEUP	Place VIP wakeup call
VOA_REPEAT	VDN of Origin Annc. Repeat Button
VOICE_MAIL	Engen fixed voice mail
VU_DISPLAY	Vu-Stats Station Displays
WARN_ALRM	Warn alarm
WHISP_ACT	Activate Whisper Page
WHISP_ANBK	Answerback for Whisper Page
WHISP_OFF	Whisper Page Off
WORK_CODE	Multi-Digit Stroke Count
8 of 8	

Table 27: Button Id Enumeration

Name	Description
ALL	Use when you want to get information about ALL the buttons
DIGIT_1	The digit 1 on the dialpad.
DIGIT_2	The digit 2 on the dialpad.
DIGIT_3	The digit 3 on the dialpad.
DIGIT_4	The digit 4 on the dialpad.
DIGIT_5	The digit 5 on the dialpad.
DIGIT_6	The digit 6 on the dialpad.
DIGIT_7	The digit 7 on the dialpad.
DIGIT_8	The digit 8 on the dialpad.
DIGIT_9	The digit 9 on the dialpad.
DIGIT_0	The digit 0 on the dialpad.
DDIGIT_STAR	The digit * on the dialpad.
DIGIT_POUND	The digit # on the dialpad.
1 of 2	

Table 27: Button Id Enumeration (continued)

Name	Description
CONFERENCE	The conference button
Drop	The drop button
TRANSFER	The transfer button
HOLD	hold button
BUTTON_1	The first button in the Principal Module starts here. If you want to get to other feature buttons start with this number and add the appropriate offset. For example, Principal Module Button 3 would be (BUTTON_1 + 2). The Principal Module is the module where the primary buttons for a station are configured. Depending on the set type, other buttons may also exist in the Feature Module, Display Module, and Coverage Module.
FEATURE_MODULE_START	The first button in the Feature Module starts here. If you want to get to other feature buttons start with this number and add the appropriate offset. For example, Feature Module Button 3 would be (FEATURE_MODULE_START + 2)
DISPLAY_MODULE_START	The first button in the Display Module starts here. If you want to get to other feature buttons start with this number and add the appropriate offset. For example, Display Module Button 3 would be (DISPLAY_MODULE_START + 2)
COVERAGE_MODULE_START	The first button in the Coverage Module starts here. If you want to get to other feature buttons start with this number and add the appropriate offset. For example, Coverage Module Button 3 would be (COVERAGE_MODULE_START + 2)
2 of 2	

Table 28: Registration States Enumeration

Name	Description
Idle	
Registering	
Registered	

Table 28: Registration States Enumeration

Name	Description
Unregistering	
Unknown	

Table 29: Dependency Modes Enumeration

Name	Description
Main	
Independent	
Dependent	

Table 30: Ringer Pattern Constants

Ring Pattern	Value
Ringer Off	0
Manual Signal	1
Attendant Incoming Call	4
Attendant Held Call	5
Attendant Call Waiting	6
Attendant Emergency	7
Intercom	9
Standard Ring	11
DID/Attendant Ring	12
Priority Ring	13
Ring Ping	14

Table 31: Registration Constants

Reason code constants for the registerTerminalRequest	Value
Unknown	-1
Non-specific	-2
Internal failure	-3
Network timeout	1000

Media Enumerations

Table 32: Media Modes Enumeration

Name	Description
None	NONE is when do not want access to the media. Typically, you want to share the control of a hard phone. You can monitor all messages going to/from the phone and inject messages (e.g., go off hook, push a button).
Telecommuter	Selecting TELECOMMUTER mode allows you to redirect the media to any device that has a phone number (e.g., cell phone). You can monitor all messages going to/from the phone and inject messages (e.g., go off hook, push a button). However, you have no access to media itself.
Server_Mode	Selecting SERVER_MODE mode allows you to register the phone directly on the server. The media will go to the server. You can monitor all messages going to/from the phone and inject messages (e.g., go off hook, push a button). However, you have no access to media itself. This mode will allow you to inject wav files into a call (as long as the wav files are stored on the server) and also to record calls.
Client_Mode	Selecting CLIENT_MODE mode allows you to register the phone directly on the computer where the application is running. You can specify where the media is supposed to go. You can monitor all messages going to/from the phone and inject messages (e.g., go off hook, push a button). However, you have no access to media itself. Since the application has access to the media it can do whatever it wants to with it.
UNDEFINED	UNDEFINED mode should never be used.

Service Provider Enumeration

Table 33: Convert E164 to Dial String Enumeration

Name	Description
CONVERTED	

Table 33: Convert E164 to Dial String Enumeration

Name	Description
NO_MATCH	
INVALID_INPUT	

Table 34: Convert Dial String to E164 Enumeration

Name	Description
CONVERTED	
NO_MATCH	
INVALID_INPUT	

Third Party Call Control Enumerations

Table 35: Single Step Conference Call Enumeration

Name	Description
None	No participation type for a Single Step Conference
Silent	Silent participation type for a single step conference
Active	Active participation type for a single step conference

Table 36: Call Forwarding Enumeration

Name	Description
ForwardImmediate	Immediate Forwarding
None Specified	No Forwarding Type Specified

Glossary

A

AE	Used as a “shorthand” term in this documentation for Application Enablement.
AES	Stands for Advanced Encryption Scheme.
API	Application Programming Interface. A “shorthand” term in this documentation for the Java interface provided by the Application Enablement Services. See also connector client API library
application machine	The hardware platform that the connector client API library and the client application are running on

B

BHCC	busy hour call capacity
-------------	-------------------------

C

client application	An application created using the Device, Media and Call Control API
CMAPI softphone	Application Enablement Services Device, Media and Call Control API software objects that represent softphone-enabled, Communication Manager telephones or extensions
Application Enablement Services Device, Media and Call Control API	The product name. This includes the server-side runtime software (see connector server software) and the connector client API library . This term is never used to reference only the client API library.
connector	This describes the function of Application Enablement Services Device, Media and Call Control API. In this context, “connector” means software and communications protocol(s) that allow two disparate systems to communicate. Often used to provide open access to a proprietary system. In the case of Application Enablement Services Device, Media and Call Control API, the connector enables applications running on a computing platform to incorporate telephony functionality through interaction with Communication Manager.
connector client API library	The Application Enablement Services Device, Media and Call Control Java API, also referred to as the AE
connector server machine	The hardware platform that the connector server software is running on. In these documents, the term “connector server” by itself never refers to the connector server machine. See connector server software .

connector server software

connector server software	The Application Enablement Services server-side runtime software, often referred to as the “connector server” in these documents
CSTA	Computer-Supported Telecommunications Applications
D	
DMA	Direct memory access
E	
ECMA	European Computer Manufacturers Association. A European association for standardizing information and communication systems in order to reflect the international activities of the organization.
H	
hold time	The total length of time in minutes and seconds that a facility is used during a call
J	
JDK	Java Developers Kit
J2SE	Java™ 2 Platform, Standard Edition
JMX	JMX (Java Management Extensions) is a set of specifications for application and network management in the J2EE development and application environment. JMX defines a method for Java developers to integrate their applications with existing network management software by dynamically assigning Java objects with management attributes and operations
JSW	Java Service Wrapper
JVM	Java Virtual Machine. Interprets compiled Java binary code for a computer’s processor so that it can perform a Java program’s instructions
O	
OAM	Operations, Administration and Maintenance
R	
RPM	Red Hat Package Manager
S	
SAT	System Access Terminal (for Communication Manager)
SDK	Software Development Kit. An SDK typically includes API library, software platform, documentation, and tools.

T

TCP

Transmission Control Protocol. A connection-oriented transport-layer protocol, IETF STD 7. RFC 793, that governs the exchange of sequential data. Whereas the Internet Protocol (IP) deals only with packets, TCP enables two hosts to establish a connection and exchange streams of data. TCP guarantees delivery of data, and also guarantees that packets are delivered in the same order in which the packets are sent.

TSAPI

Telephony Services Application Programming Interface. A service that provides 3rd party call control.

TTI

Terminal Translation Initialization. This is a feature in Communication Manager that allows administrators, when initially administering new DCP stations, to not initially bind the extension number to a port. When the technician is installing the stations, they then use the TTI feature access code to bind the extension number to the station.

V

VoIP

Voice over IP. A set of facilities that use the Internet Protocol (IP) to manage the delivery of voice information. In general, VoIP means to send voice information in digital form in discrete packets instead of in the traditional circuit-committed protocols of the public switched telephone network (PSTN). Users of VoIP and Internet telephony avoid the tolls that are charged for ordinary telephone service.

X

XML

Extensible Markup Language

XSD

XML Schema Definition. Specifies how to formally describe the elements in an Extensible Markup Language (XML) document. This description can be used to verify that each item of content in a document adheres to the description of the element in which the content is to be placed. This protocol uses these instead of DTD's.

Index

A

API library	
using reference documentation	34
application	
debugging	91
developing	37
improving performance	92
Avaya Support Centre	
website	10
Avaya University	8

B

bearer channels	
unencrypted	90
BHCC	125
buttons	
function	112
get information	18
ID constants	119
knowing	73
on a device.	73
press	18

C

call	
incoming	73
call appearances	
detecting incoming call	73
green lamp turns off.	73
calls.	32
ended by far end	73
end-of-call cleanup	78
cleanup	78
client API library	
using reference documentation	34
client application	125
also see application	
client media mode	68
CM Link	125
CMAPI softphone	125
codecs	
choosing	71
files to be played	75
recordings	75
coder/decoder, see codecs	

Communication Manager	
features.	107
loss of communication with.	64
network configuration	93
connector	125
CSTA	
becoming familiar with	8
concepts	30

D

debugging	91
deployment	
security	89
detecting incoming call	73
development environment.	29
device	
cleanup.	78 , 79
logical elements	31
physical element	31
registering	63
unregistering	79
display	
for incoming call.	73
get contents.	18
updated event.	19
Display Updated event	16
DMA.	126
downloading	
Communication Manager API SDK	29
dropped call	73
DTMF digits	
start collecting.	21
stop collecting.	21
Tone Detected event	22

E

ECMA	10 , 126
encoding algorithms	75
encryption	90
events	32
Display Updated.	16
exiting application	78
extension	
specifying for a device	17

Index

F

far-end RTP media parameters [76](#)
freeing up resources [78](#)

G

G.711 [67](#)
 specifying during registration [71](#)
G.729 [67](#)
 converter [74](#)
 non-standard RIFF values [74](#)
 specifying during registration [71](#)

H

hold time [126](#)
hookswitch
 get status [18](#)
 set status [19](#)
 status changed event [19](#)

I

ideas [107](#)
incoming call
 detecting [73](#)

J

J2SE [126](#)
JDK [126](#)
JVM [126](#)

L

lamps
 color [112](#)
 mode [99](#), [100](#), [101](#), [111](#)
 status change [19](#)
Linux kernel
 updates [93](#)
Linux operating system
 tuning [92](#)
local media mode
 when far-end RTP parameters change [76](#)
logical elements [31](#)

M

media
 controlling [66](#)
 encoding [71](#)

 modes [68](#)
 when far-end RTP parameters change [76](#)
Media Start event
 using [76](#)
media stream
 playing messages to [74](#)
 recording messages from [74](#)
message waiting indicator
 get status [18](#)

N

network
 failure [64](#)
 performance [92](#)
network regions [72](#)

P

password
 specifying for station [64](#)
performance [92](#)
physical elements [17](#), [31](#)
playing messages [74](#)
Programmer's Reference [10](#)

R

race conditions [92](#)
real-time protocol, see RTP
recording [74](#)
recovery [76](#)
registerDevice method [63](#)
registering devices [63](#)
Resource Interchange File Format, see RIFF
RIFF [74](#)
ringer
 detecting incoming call [73](#)
 get status [18](#)
 pattern [122](#)
 status changed event [19](#)
RPM [126](#)
RTP [66](#)
 media stream [66](#)
 parameter changes [68](#)
 parameters [67](#), [76](#)

S

SAT [126](#)
SDK [126](#)
security
 considerations [89](#)
server media mode [68](#)

services	
Register Device	
using	63
signaling channel	
unencrypted	90
station administration	
button assignments	73
stations	
buttons.	73

T

test environment	30
TTI	127

V

verification application	91
VOIP	
readiness guide	93
VoIP	127

W

Wave files	
file structure	74
websites	
Avaya Developer Connection	8
Avaya Support Centre	8
Avaya University	8
ECMA	10

X

XML messages	
security	90

