

Multimedia Terminal Framework Programmer Guide



WE ARE TAKING YOUR
COMMUNICATIONS ONLINE
TO A RICHER COMMUNICATIONS
LEVEL. OUR PRODUCTS ENABLE A RICHER COMMUNICATIONS
EXPERIENCE TO TAKE PLACE ANYWHERE.



RADVISION
Delivering the Visual Experience

NOTICE

© 2001-2007 RADVISION Ltd. All intellectual property rights in this publication are owned by RADVISION Ltd. and are protected by United States copyright laws, other applicable copyright laws and international treaty provisions. RADVISION Ltd. retains all rights not expressly granted.

The publication is RADVISION confidential. No part of this publication may be reproduced in any form whatsoever or used to make any derivative work without prior written approval by RADVISION Ltd.

No representation of warranties for fitness for any purpose other than what is specifically mentioned in this guide is made either by RADVISION Ltd. or its agents.

RADVISION Ltd. reserves the right to revise this publication and make changes without obligation to notify any person of such revisions or changes. RADVISION Ltd. may make improvements or changes in the product(s) and/or the program(s) described in this documentation at any time.

If there is any software on removable media described in this publication, it is furnished under a license agreement included with the product as a separate document. If you are unable to locate a copy, please contact RADVISION Ltd. and a copy will be provided to you.

Unless otherwise indicated, RADVISION registered trademarks are registered in the United States and other territories. All registered trademarks recognized.

For further information contact RADVISION or your local distributor or reseller.

Multimedia Terminal Framework version 2.5.1.54, November, 2007

Publication 11

<http://www.radvision.com>

CONTENTS

PART 1: INTRODUCTION

1 Overview

What's in this Chapter	3
Introduction	3
Products	4
Advantages	5

2 Architecture

What's in this Chapter	7
Components	7
Capability Guidelines	9
Model Design	11
Terminations	11
Events, Signals, and Packages	15
Diagrams	17
Structure	20
Call Control	20
MDM Adaptor	21
SIP Adaptor	22
MDM	22
SIP Stack	23
SDP Stack	23
Common Core	23
Sample Call Flows	24
State Machines	26

PART 2: USING THE MULTIMEDIA TERMINAL FRAMEWORK

3 Building Your Application

What's in this Chapter	45
Introduction	45
Step 1: Initialization	46
Initialization Steps	46
Configuration	52
Registering Terminations	58
Starting the System	62
Step 2: Playing Signals	64
Signal Callbacks	64
List of Signals	66
List of Signals for Analog Termination	68
Step 3: Reporting Events	69
Reporting Events API	69
List of Events	70
List of Events for Analog Termination	73
Collecting Digits	75
Matching Digits to an Address	82
Step 4: Integrating Media	84
Media Callbacks	85
Call Flows	109

4 Extensibility

What's in this Chapter	111
Overview	111
MDM Control	112
MDM Extension Callbacks	113
MDM Extension APIs	117

Code Flow	118
User Interface Files	119
Examples	119
SIP Control	133
SIP Extension Callbacks	135

5 *Sample Application*

What's in this Chapter	157
Introduction	157
Configuration	159
GUI Application	165
Running the Sample Application	168
Structure and Objects	185
Sample Code	190
EPP	195
EPP Objects	195
EPP Sockets	196
Call Flow Example	197
EPP Events Table	198
EPP Signals Table	200
Integrating MTF without EPP	204
IPv6	207
Message Waiting Indication	208

6 *Logging*

What's in this Chapter	213
Multimedia Terminal Framework	213
Initializing the Log	213
Closing the Log	218
Stack Log	219
SIP Stack Log	219

7	<i>Shutting Down</i>	
	What's in this Chapter	227
	Shutting Down	227

PART 3: FEATURES

8	<i>Features</i>	
	What's in this Chapter	233
	Common Features	233
	Caller ID	233
	Call Waiting	237
	Hold	238
	Call Transfer	240
	Three-Way Conference	254
	Headset/Handsfree	257
	Mute	259
	Warm Restart	260
	IPv6	261
	Protocol-Specific Features	263
	SIP Features	263

PART 4: MEDIA ADD-ON

9	<i>Media Add-on</i>	
	What's in this Chapter	315
	Overview	315
	Working with the Media Add-on	316
	Architecture	319
	Media Framework	321
	Supported Codecs and Features	323

10	<i>Optimizations</i>	
	What's in this Chapter	325
	Introduction	325
	Reducing Footprint	326
	Minimizing Memory	326
	Optimizing Performance	327
APPENDIX A	<i>Cache Allocator</i>	329
	Introduction	329
	Compile-time Configuration Flags	331
APPENDIX B	<i>What's New in Version 1.4.2</i>	333
	New Features	333
	Interface Changes	333
APPENDIX C	<i>What's New in Version 2.0</i>	335
	New Features	335
	New API and Types	336
	Interface Changes	337
	Modifications	338
APPENDIX D	<i>What's New in Version 2.1</i>	341
	New Features	341
APPENDIX E	<i>What's New in Version 2.5</i>	343
	Interface Changes	343
	New Features	346

APPENDIX F	<i>What's New in Version 2.5.1.47</i>	349
	New Features	349
	Interface Changes	350
	Memory Optimization	350
APPENDIX G	<i>What's New in Version 2.5.1.54</i>	351
	Memory Reduction	351
	Interface Changes	352
	Other Changes	353
	<i>Index</i>	355

ABOUT THIS MANUAL

The RADVISON Multimedia Terminal Framework provides building blocks for the fast development of VoIP media devices and includes RADVISON Protocol Stacks, a Call Control layer, a unified protocol-agnostic Media Device Manager (MDM) API and Protocol Abstraction Layer, and relevant user documentation.

This Programmer Guide describes the components of the RADVISON Multimedia Terminal Framework and explains how to integrate it with a user application. The guide is divided into the following parts:

- [Part 1: Introduction](#)
- [Part 2: Using the Multimedia Terminal Framework](#)
- [Part 3: Features](#)
- [Part 4: Media Add-on](#)
- [Appendix A: Cache Allocator](#)
- [Appendix B: What's New in Version 1.4.2](#)
- [Appendix C: What's New in Version 2.0](#)
- [Appendix D: What's New in Version 2.1](#)
- [Appendix E: What's New in Version 2.5](#)
- [Appendix F: What's New in Version 2.5.1.47](#)
- [Appendix G: What's New in Version 2.5.1.54](#)

RELATED DOCUMENTATION

The following documentation is provided with the RADVISON Multimedia Terminal Framework:

- [Multimedia Terminal Framework Programmer Guide](#)
- [Multimedia Terminal Framework Reference Guide](#)

- SDP API
- Porting Guide for RADVISION Stacks
- Release Notes

PART 1: INTRODUCTION

1

OVERVIEW

1.1 WHAT'S IN THIS CHAPTER

This chapter describes the RADVISION Multimedia Terminal Framework and includes the following:

- Introduction
- Products
- Advantages

1.2 INTRODUCTION

RADVISION's Multimedia Terminal Framework allows application developers to focus on functionality rather than on standard specifications when developing Customer Premise Equipment (CPE) devices. Developers working on functionality on the layer above the Multimedia Terminal Framework do not need to be experts on VoIP standards; the Multimedia Framework frees them to focus on application development, hardware integration and user interaction.

The Multimedia Terminal Framework can be used to build the following types of products:

- Mobile Handsets
 - Dual mode phones
 - VoIP over WiFi
 - IMS Clients
- IP STBs (Set-Top-Boxes)
- Videophones
- IP Phones
- Integrated Access Devices (IADs) or Residential Gateways

A VoIP (voice and video over IP) phone is a telephone that transports voice and video over data networks—packet or circuit switch—rather than over legacy voice networks. The VoIP phone is connected to an IP network and uses IP signaling protocols such as SIP to initiate and control calls, in addition to sending and receiving media over the network as digitized packets. The latter is generally done using the RTP/RTCP protocol. The VoIP phone, sometimes combined with other elements of the network, may offer a number of telephony features beyond the ability to establish simple calls. Standard requirements for the Terminal Framework are described in TIA-EIA-IS-811.

The RADVISION Multimedia Terminal Framework is designed to reduce time to market for terminal developers. By using the Multimedia Terminal Framework, developers can create products for SIP networks with few or no changes to their application. Applications written with the Multimedia Terminal Framework are insulated from the underlying protocols and Call Control mechanics.

1.3 PRODUCTS

The Multimedia Terminal Framework may be used to build a wide range of products requiring client-side functionality:

- **Mobile Handsets**

Mobile handsets today are in transition from using circuit-switched technology to packet-switched technology. This transition occurs in PDAs and dual mode handsets employing WiFi networks, but also on the mobile network itself. Packet-switched telephony on mobile handsets today uses SIP.

- **IP Phones**

The IP Phone offers features that are commonly found in business phones. These features may include multiple lines, LCD displays, speakerphones, and supplemental services such as Hold, Transfer, and Conference.

- **Videophones**

The Videophone offers the same features as the IP Phone, with the addition of video capabilities.

- **IP STBs**

IP Set-Top-Boxes today are in the center of the digital home. As such, support of video conferencing in the STBs themselves offer the means of connecting between households. Here, quality of service, privacy and conferencing are key issues.

- **IADs**

The Residential Gateway is a small device to which a number of traditional analog phones may be connected. When a Residential Gateway is used, only this device—and not the analog phones—will be connected to the IP network.

Note An application based on the Multimedia Terminal Framework can support any of the above products.

A complete Terminal Framework solution can be achieved by integrating the RADVISION Multimedia Terminal Framework with DSP Services (voice compression and decompression, echo cancellation, VAD, etc.), Video Services (camera and screen) and Telephony Services (Hook state, Key state, Handset, etc.) provided by the developer or by a third party. Media transport can be implemented by using the RADVISION Advanced RTP/RTCP Toolkit or a proprietary solution. Media processing can be implemented by using the RADVISION Media Framework or a proprietary solution.

1.4 ADVANTAGES

RADVISION's Multimedia Terminal Framework is a unique solution that enables developers to produce CPE devices faster and more reliably than ever before. The Multimedia Terminal Framework reduces time to market and allows developers to focus on the application. With support for multi-protocol enhanced telephony services, the extensible solution can be integrated with any hardware and platform. RADVISION's Multimedia Terminal Framework:

- **Allows customers to focus on the application**

The Multimedia Terminal Framework hides the actual signaling protocols used, allowing developers to focus on the application. This eliminates long learning curves for actual signaling protocols and media components.

- **Supports enhanced telephony services**

The Multimedia Terminal Framework comes with the complex telephony services already implemented, so that developers can seamlessly integrate them into products without dealing with the underlying signaling mechanisms and internal state machines. This includes Transfer, Forward, N-way Conferencing, Hold, Mute and more.

Advantages

- **Can be integrated with any hardware and platform**

The Multimedia Terminal Framework is a generic solution that is supplied in source code format. It is based on RADVISION's Common Core (an operating system abstraction) and supports multiple operating systems. In addition, the Multimedia Terminal Framework's APIs are specifically designed to handle all types of hardware integration issues, such as Ring, Busy, On-hook, Off-hook, etc. No mapping is required from a specific signaling protocol to the telephony hardware.

- **Supports multiple protocols**

The Multimedia Terminal Framework's APIs are protocol agnostic. This gives developers the confidence that any future protocol will work out of the box with the same code developed on top of it, with only minor changes required. In a world of ever-changing protocols, this ensures future-proof devices.

- **Is extensible**

The Multimedia Terminal Framework is not a closed solution. Developers can change the developed terminal's default behavior to tweak incoming or outgoing messages to fit their own proprietary needs. They may also add new and exciting features that differentiate their products.

2

ARCHITECTURE

2.1 WHAT'S IN THIS CHAPTER

This chapter explains the model on which the Multimedia Terminal Framework is based, the basic components of the Multimedia Terminal Framework, and the software objects and thread model used in the Multimedia Terminal Framework:

- Components
- Capability Guidelines
- Model Design
- Structure
- Threading Model

2.2 COMPONENTS

The Multimedia Terminal Framework architecture combines RADVISON's industry leading family of VoIP components into a powerful developer framework that both increases application features/functionality and reduces time to market.

Components

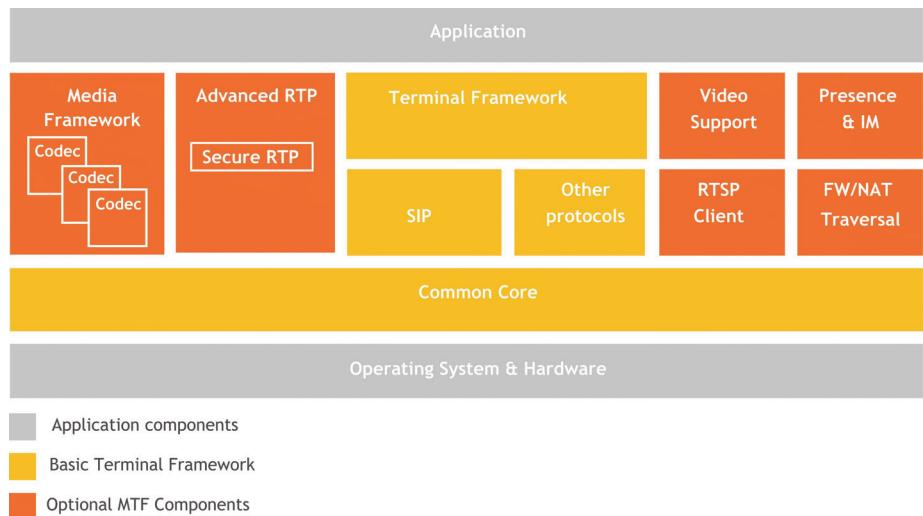


Figure 2-1 *Multimedia Terminal Framework Components*

Figure 2-1 illustrates the software layer concept of the multi-protocol Multimedia Terminal Framework for chipset-based applications. The Terminal Framework is the basic component of the Multimedia Terminal Framework. It is designed to provide the basis of any client implementation, allowing applications to interact with telephony concepts without specific knowledge of the underlying signaling protocol in use.

The Terminal Framework uses a Media Control model for abstracting the signaling information for presentation to the application. It also implements the call state machine of the Multimedia Terminal Framework. This includes call establishment (make/receive call) and tear down as well as a number of supplementary services, such as:

- Caller ID
- Multiple-line appearances
- Conference (3-way)
- Transfer (blind and semi-attended)
- Call Forwarding/Diversion (unconditional, on-busy, no answer)
- Call Waiting
- Hold

- Mute
- Handset/Headset/Speaker Terminations

The SIP and Other Protocols are used by the Terminal Framework to access the low-level protocol signaling required for VoIP. It can be accessed by the Multimedia Terminal Framework application to implement additional proprietary features for flexibility and extensibility. Other than that, the Terminal Framework hides the logic required to maintain the protocol specific state machines. Other components are optional, giving additional value to developers. They enable firewall/NAT traversal, media processing and other features.

2.3 CAPABILITY GUIDELINES

The platform on which the Multimedia Terminal Framework is implemented should provide or emulate the set of signals, events, and media capabilities described below. The requirements for the user platform comply with TIA/EIAIS-81.

Signaling

The user application should implement telephony signaling capabilities such as:

- Ringing tone
- Ringback tone
- DTMF tone
- Dial tone
- Busy tone
- Congestion tone
- Warning tone
- Call Waiting tone
- Caller Waiting tone

Event Detection

The user application should have the ability to report events such as:

- Off Hook
- On Hook
- Flash Hook
- DTMF Key Down/Up
- Special Key Down/Up

Media Capabilities

The user application must support at least:

- Audio codec
- Video codec
- Jitter removal
- Packet buffering
- Echo cancellation
- Voice activation detection
- Mixing (for 3-way conferencing)
- Tone detection

Display

The user application may support a display and/or soft keys (keys with display). The display interface should allow detailed control, such as XY-coordinates, line-wrap handling, and screen dimension auditing.

2.4 MODEL DESIGN

The Multimedia Terminal Framework model sees the Terminal Framework as a set of logical entities:

- Terminations
- Connections
- Calls
- Events and Signals

This model is based on the Megaco/H.248 protocol, which offers a low-level view of the device that allows each element of it to be controlled.

TERMINATIONS

A termination is a source of events and media, and a destination of signals and media. There are two types of terminations:

- **Physical terminations**

Physical terminations are permanent terminations that map to one or more physical elements in the phone. There are four types of physical terminations:

- User Interface (UI) termination

This termination groups all User Interface elements such as keys, LCD display, and lights. The user application can register only one UI termination.

- Audio Transducer (AT) terminations

Each of these represents an audio device in the phone. For example, the at/ht termination maps to the Handset device. The user application can register one or more AT terminations, one for each audio transducer supported by the phone.

- Video Transducer (VT) Terminations

Each represents a video device: vt/cam maps to the camera, and vt/screen maps to the video screen.

- Analog terminations

Represents an analog line. The user application can register one or more analog terminations, one for each analog line that the Residential Gateway provides.

- **Ephemeral terminations**

Ephemeral terminations are temporary RTP terminations that represent the media flow and exist only for the duration of a call. The ephemeral termination groups one or more media streams. It is deleted when those streams are no longer needed.

Terminations used by the phone must be registered with the Multimedia Terminal Framework by the user application when they become active, and must be unregistered when they become inactive.

[Figure 2-2](#) shows a sample model of the Terminal Framework. The Terminal Framework application includes:

- One UI termination, grouping all User Interface elements
- One or more Audio Transducer terminations representing the audio devices
- May include Video terminations representing video screens and cameras
- During a call, will include RTP terminations (one for each call)

The RTP termination is connected to one of the Audio Transducer terminations at a time, and to the RTP port.

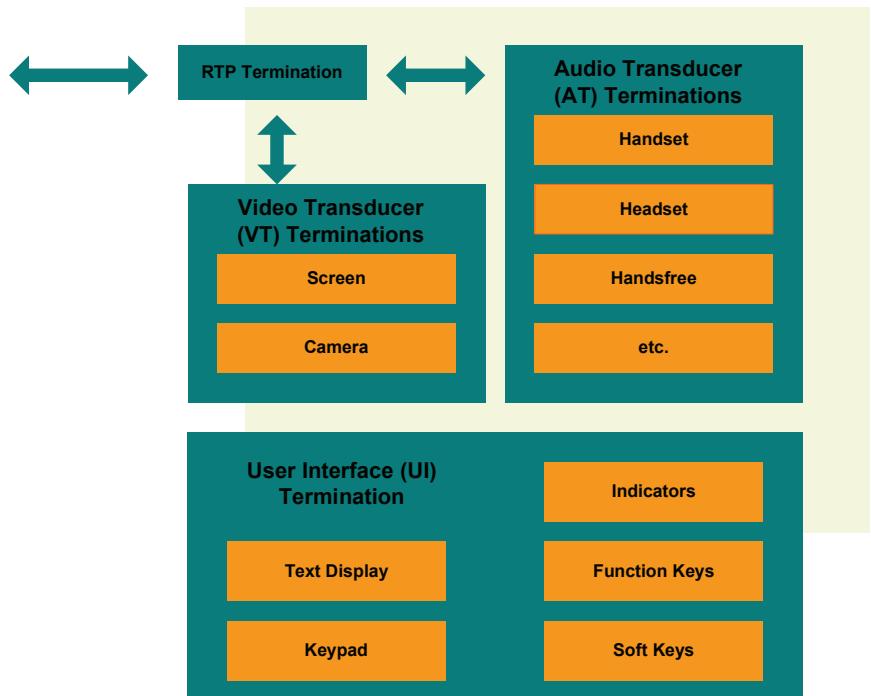


Figure 2-2 Model of a Videophone Application

Figure 2-3 shows a sample model of a Residential Gateway application. Each of the analog phones registers to the Multimedia Terminal Framework separately and is represented by an Analog termination.

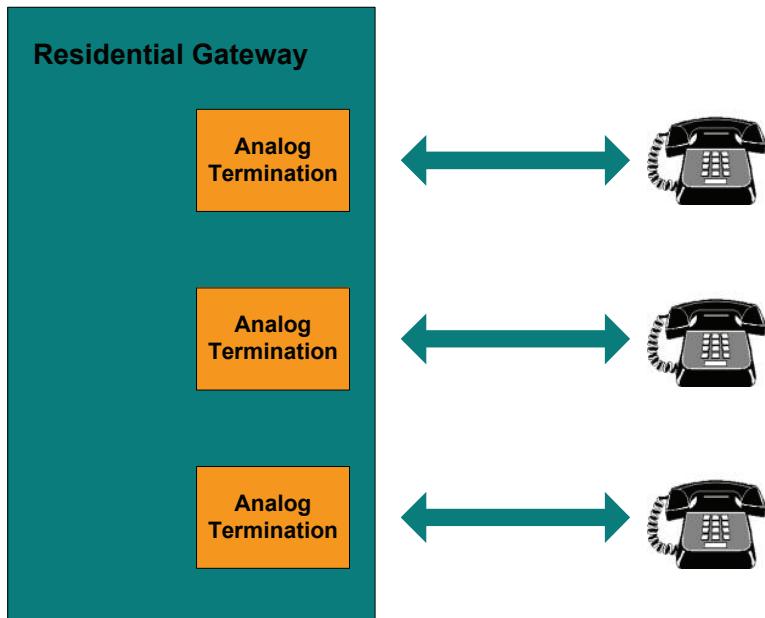


Figure 2-3 Model of a Residential Gateway Application

CONNECTIONS

Connection objects represent one party of the call. There are two types of connections:

- **MDM Connection**—Represents the local party. Each MDM Connection maps to a physical line in the phone. The MDM Connection object is located in the MDM Adaptor.
- **Protocol Stack Connection**—Represents the remote party of the call. Each Connection maps to a call-leg object created by the SIP Stack for each call. The SIP Connection object is located in the SIP Adaptor.

CALLS

A call object holds all connections (one or more) that participate in it. These may be MDM connections and network connections, according to the state of the call. The call object has a common interface with all connections, meaning that it does not differentiate between a local party and a remote (network) party. The call object is located in the Call Control layer.

The diagram below illustrates the unified interface between the call object and the connections. When a call applies Call Make on a connection, the same API is used, but each connection implements it in its own way:

- The MDM connection applies the Ringing signal on the termination
- The SIP connection sends an INVITE message

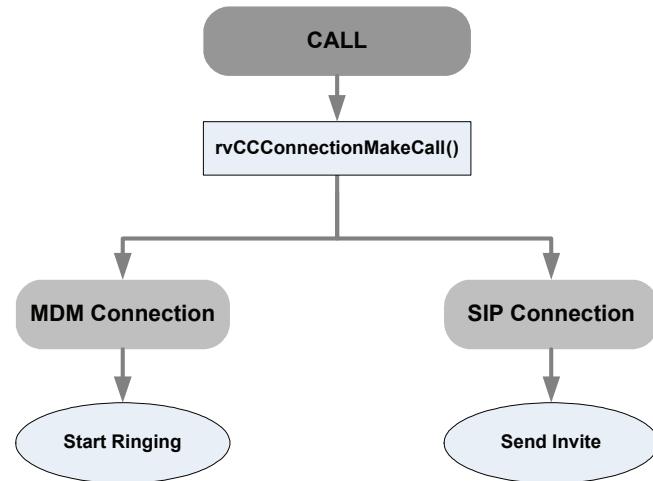


Figure 2-4 Make Call implementation

EVENTS, SIGNALS, AND PACKAGES

An event is an action that occurs on a phone and that is usually initiated by the user of that phone. Examples of such events are lifting the Handset off-hook or pressing a key in a keypad.

Signals are applied to the phone application by the Multimedia Terminal Framework according to the state of the call. For example, a Ring signal is applied when an incoming call is received. A Dialtone signal is applied when the user goes off-hook. The Multimedia Terminal Framework uses a well-defined set of signals and events to interface with the application.

Figure 2-5 shows the flow of signals and events between the Multimedia Terminal Framework and the user application.

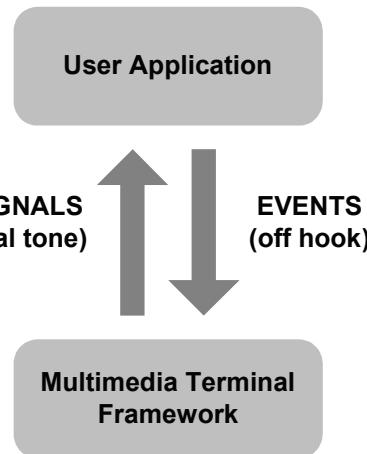


Figure 2-5 Flow of Signals and Events

Signals and events are grouped into packages. Each package defines a set of signals and events, each with its own unique ID. Sometimes signals and events may have parameters. Each parameter contains a property ID plus a specific value. Packages also define properties. Some of these properties are related to the state of the termination, while others are associated with a specific media stream in the termination (for example, Echo Cancel On/Off).

PACKAGES AND TERMINATION TYPES

Each termination type uses different packages:

- The User Interface (UI) termination supports the packages "ds", "kp", "kf" and "ind".
- The Audio Transducer (AT) and Video Transducer (VT) terminations support the packages "dg" and "cg".
- The Analog termination supports the packages "dg", "dd", "cg", and "al".

Note For a definition of these packages, see the Megaco documents H.248 Annex G (UI packages used in the UI termination) and *RFC 3015*.

DIAGRAMS

The Multimedia Terminal Framework model is illustrated by the following examples:

[Figure 2-6](#) illustrates an outgoing call. When the user presses digits to setup a call, the user application reports on digit events through the MDM interface. The event is propagated to the relevant termination, which sends it to be processed by the relevant connection. The connection sends a callback to the call, which passes the event to the other party of the connection: the SIP connection. The SIP connection uses the SIP Manager to start an outgoing call by sending out a new message (INVITE).

[Figure 2-7](#) illustrates an incoming call. When a new call arrives, the SIP Stack receives an INVITE message that raises a New Call event to the SIP Adaptor. The SIP Adaptor creates a new connection and passes on an Offering event to its state machine. At this stage, the Call Control maps the call to the correct termination. A Call object is created, which passes a Make Call event to the corresponding MDM connection. A Ringing signal is applied to the termination to indicate an incoming call.

Model Design

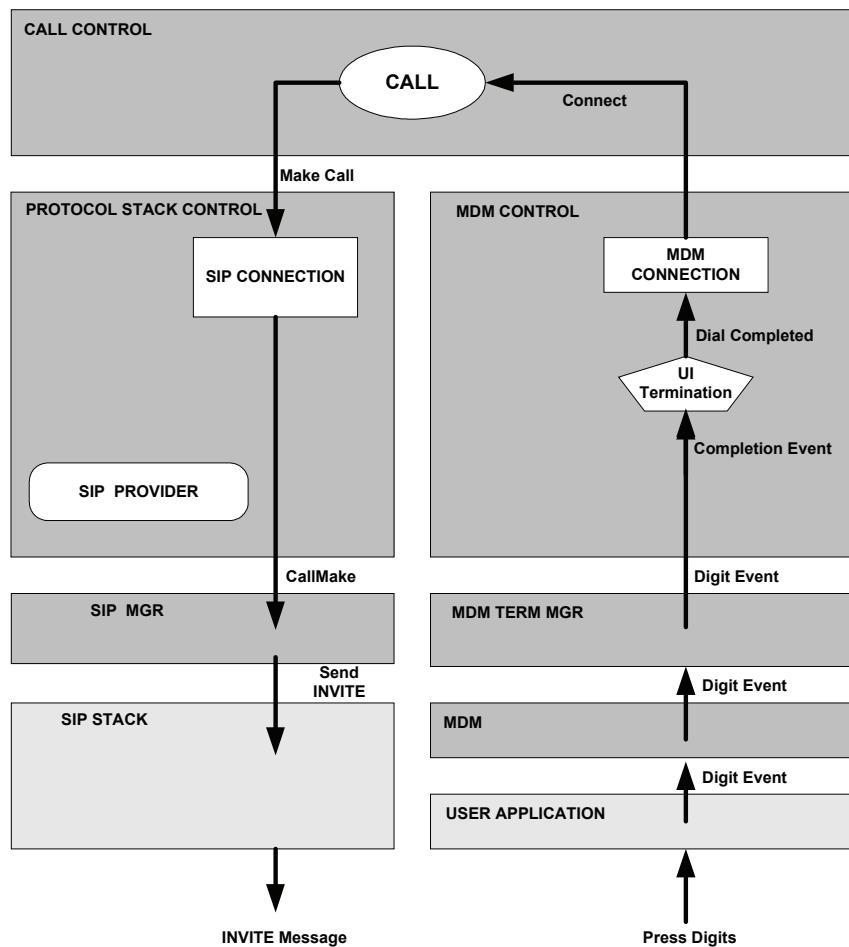


Figure 2-6 User Application Event

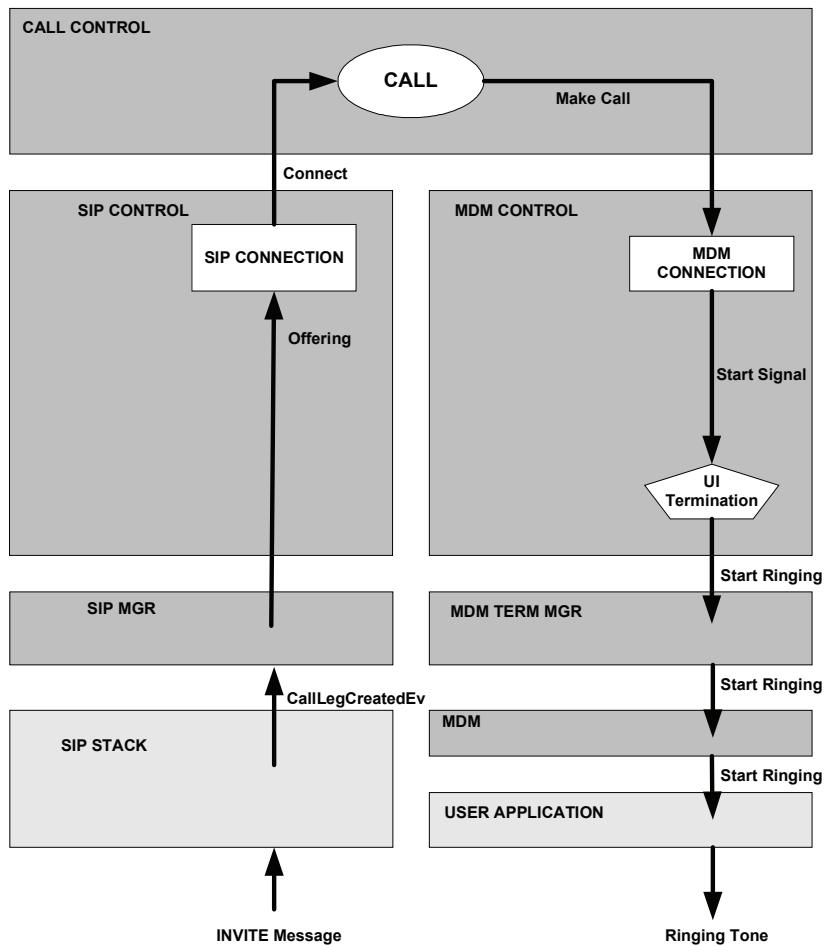


Figure 2-7 Network Event

2.5 STRUCTURE

The Multimedia Terminal Framework consists of the following components:

- Call Control
- MDM Adaptor
- SIP Adaptor
- MDM

It is based on the following RADVISION Toolkits:

- SIP Stack
- SDP Stack
- Common Core

In addition, developers may implement Media Transport by using RADVISION's Advanced RTP/RTCP Toolkit (optional).

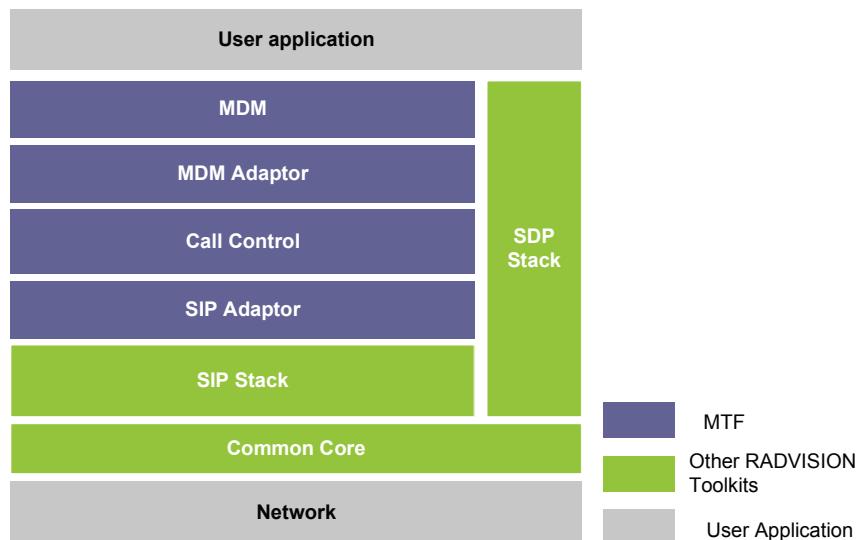


Figure 2-8 *Multimedia Terminal Framework Structure*

CALL CONTROL

The Call Control module implements all logic needed for managing the state of a call. This module is also responsible for implementing an extensive set of Supplementary Services, such as multiple line support, Call Waiting, Caller ID, Hold, Transfer, and Conference.

The Call Control module handles all events, signals, and media connections at the call level. It sees the adaptors underneath it (the MDM Adaptor and the Protocol Adaptor) symmetrically and does not differentiate between them. The interface between the Call Control and the adaptors is unified and protocol independent, so that the Call Control can process and pass events from one adaptor to the other symmetrically (see [Figure 2-8](#)). The Call Control interacts with the underlying adaptors through a unified set of APIs, and the adaptors report to the Call Control by protocol-independent callbacks (events).

The main functionalities of the Call Control module are as follows:

- Managing a state machine of the call for handling incoming calls, outgoing calls, and Supplementary Services.
- Mediating between the MDM Adaptor and Network Adaptor.

The Call Control Manager also provides a level of customization. For example, a user callback is called to translate the accumulated digits to a network address, so that the user can provide his own mapping scheme from telephone numbers to network addresses. For more information about customizing the Call Control, see the chapter [Extensibility](#).

MDM ADAPTOR

The MDM Adaptor is responsible for handling all aspects of the terminations and the local party of the call. It manages the media of each termination, applies the signals, and processes the events received from the user application.

The MDM adaptor consists of two modules:

- **MDM Control**
 - Manages the state machine of the MDM connection
 - Controls media streams; notifies the user application to:
 - ◆ Create/modify/destroy media streams
 - ◆ Connect/disconnect media streams
 - Sends signals to the user application to apply signals to the phone.
- **MDM Termination Manager**
 - Manages the events queue for user application events
 - Handles the default digitmap mechanism

SIP ADAPTOR

The protocol-specific module processes protocol events and passes them to the Call Control module when necessary. It also offers a protocol-independent API to the Call Control (for example, "makeCall").

The protocol-specific module handles protocol-specific issues related to the network side of the call. For example, the SIP Call Control processes SDP bodies of incoming SIP messages and builds SDP bodies of outgoing SIP messages, for media negotiation. As [Figure 2-8](#) illustrates, this module is implemented on top of the relevant Protocol Stack.

The network adaptors consist of two modules:

- **Control**

- Handles the state machine of the SIP connection.
 - Handles incoming and outgoing network messages.

- **Manager**

The SIP Manager is responsible for building outgoing SIP messages and listening for incoming SIP messages.

This module includes the following:

- Stack configuration and initialization
 - Sending and receiving protocol-specific messages (for example, adding SDP body, etc.)
 - Implementing Stack callbacks

MDM

The Media Device Manager (MDM) API is the interface between the user application and the Multimedia Terminal Framework. The MDM API controls the Media Device through a low-level interface. This interface is protocol independent, which means that changing the signaling protocol (for example, from SIP to other protocols) will cause minor changes in the application.

The MDM interface is protocol agnostic and uses an abstract model to handle signals, events, and media. The MDM model is based on the Megaco model, which offers a low-level view of devices that make up a Terminal Framework. This allows for easy integration with DSP Services and Telephony Services device drivers.

The MDM enables fast integration of signal control stacks with the DSP-level transcoding software and telephony hardware drivers. The MDM interface is designed to match DSP Services and Telephony Services interfaces. Thus, interfacing the DSP Services and Telephony Services drivers to MDM requires a minimum of logic. Abstracts of the DSP level transcoding software and telephony hardware drivers from the specific call-signaling protocol make both the DSP solution and the call signaling of the application agnostic.

MDM is used freely here as a synonym of Multimedia Terminal Framework or the Multimedia Terminal Framework Call Control, as the application sees the Terminal Framework through the MDM.

SIP STACK

The Protocol Stack is the Multimedia Terminal Framework module that implements the signaling protocol (SIP or other protocols) the Terminal Framework will use to establish and control calls. As [Figure 2-8](#) illustrates, there is no need for direct interaction with the Protocol Stack. However, users who want to exert tight control over Stack behavior can do so via extension APIs. For more information, see the [Extensibility](#) chapter.

The multi-protocol Multimedia Terminal Framework is based on the RADVISON SIP Protocol Toolkit. The SIP Toolkit provides SIP services that enable media over IP communication. The SIP Toolkit is fully compliant with *RFC 3261* (SIP) and can be used as a building block for the development of a wide range of applications, from cellular phones to high-end servers.

SDP STACK

The RADVISON SDP Stack is a stand-alone SDP library that provides SDP message processing functionality. SDP is a tool for specifying media capabilities, not a communication protocol. The SDP Stack performs the following:

- Parsing and encoding of SDP messages or message parts, such as media descriptions and RTP maps
- Browsing and editing of SDP messages and message parts.
- SDP Message creation, either from new or existing messages or message parts.

The SDP Stack parser and encoder supports all header fields specified in SDP. The SDP library provides an API that enables you to access all message parts, including the sub-fields of the different headers. In some cases, non-standard parameters may be read and written to messages. The SDP Stack component is fully compliant with RFC 2327 and various extensions. For more information, see the *RADVISON SDP Stack Programmer and Reference Guide*.

COMMON CORE

The OS Abstraction layer (Common Core) is the only component of the SIP Toolkit that is platform-specific. It exports OS services to the other components, such as network services and memory allocation, so that all components of the SIP Toolkit can be kept platform-independent. For more information about porting the Common Core of the SIP Stack, see the *Porting Guide for RADVISON Protocol Stacks*.

SAMPLE CALL FLOWS

The call flow illustrated in [Figure 2-9](#) shows how a call is set up through the Multimedia Terminal Framework using SIP. The application generates events to which signals and media commands are applied.

Note The interface with the application (left-hand side) is the same in sample call flows while it is different on the network side (right-hand side), depending on the signaling protocol used.

When a protocol message (for example, a SIP INVITE message) arrives through the network, the termination to which this message is addressed is found and the message is propagated to the corresponding state machine, which represents the call. According to the state, some signals and media commands will be applied to the termination through the MDM.

For example, the INVITE message can cause a Ring Tone to be applied if the termination is idle, or a Call Waiting tone when a different call is active if the termination supports multiple lines. A CreateMedia callback will be called through the MDM to request the local termination to open a media channel, giving it the parameters of the remote party.

Similarly, when an event is generated in the terminal, it is processed through the state machine, which will then generate commands through the MDM or send a protocol message to the network. For example, the Off Hook event can cause a Dial Tone to be applied to the termination and move to the state where digits are collected without requiring a network message to be sent. On the other hand, the On Hook event on an active call will cause a SIP BYE message to be sent to the other party.

SIP CALL FLOW

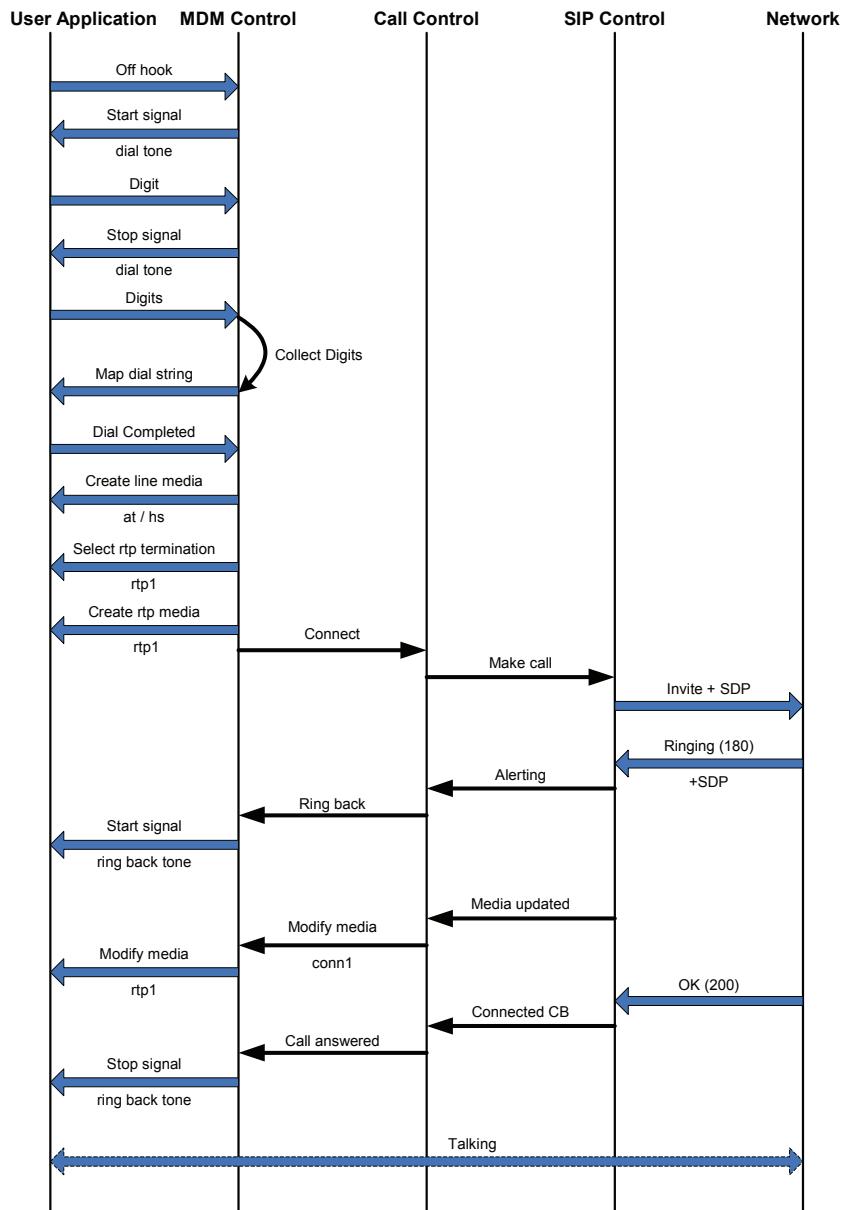


Figure 2-9 SIP Call Flow

STATE MACHINES

MDM CONNECTION (RvCCConnMDM) STATE MACHINE

This section describes an MDM Connection and a Call state machine.

A new connection is created each time a new call is originated or received on a termination. The connection implements the logic of the Call Control, processes events from the user and the network, and applies signals and commands. This Connection State Machine describes possible connection states and transitions.

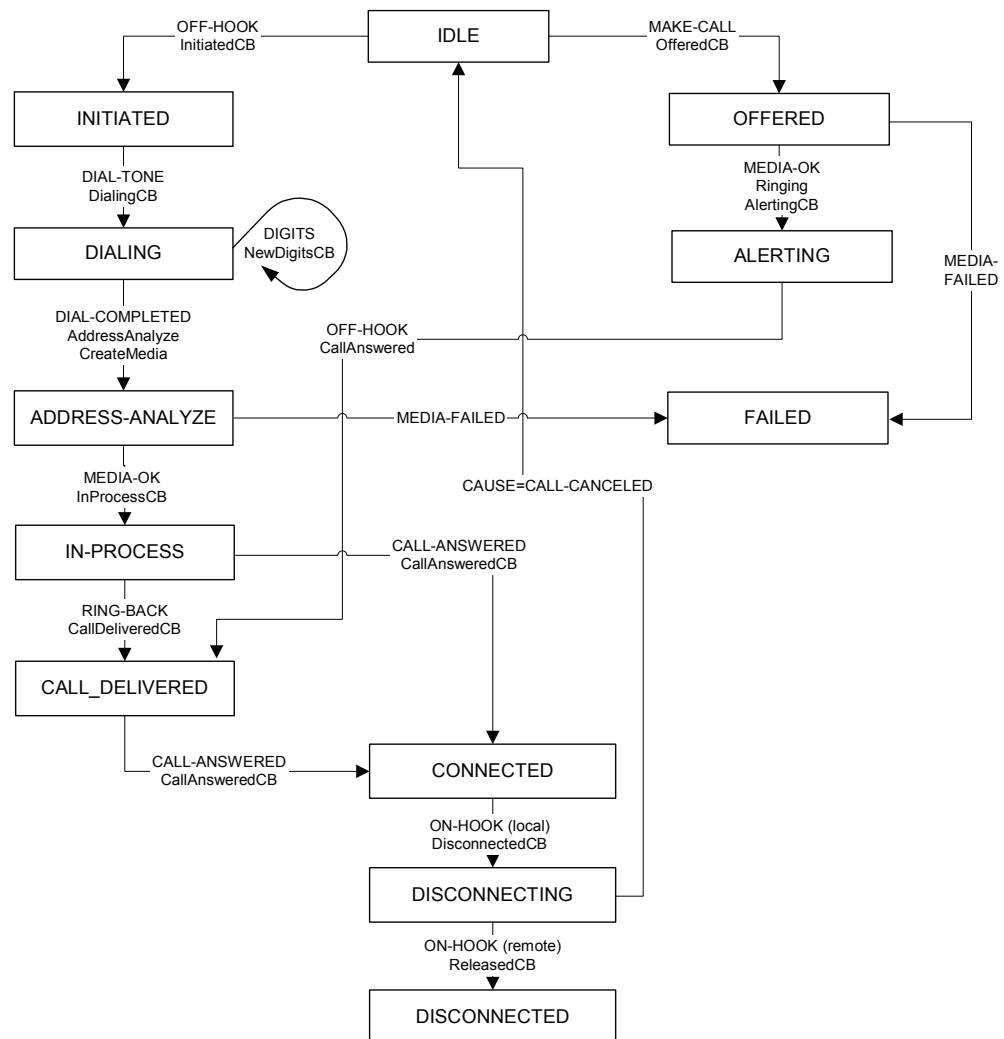


Figure 2-10 Connection State Machine

This section describes the states and events used in the state machines.

CONNECTION STATES

A connection object describes the relationship between a Terminal (an entity representing the physical end of a call, usually a phone) and a Call. A new connection is created for a Terminal for each new incoming or outgoing call. The sections below describe the possible states that the connection can transition.

RV_CCCONNSTATE_IDLE

This state is the initial state for all new connections. Connections in the IDLE state are not actively part of a call. Connections typically do not stay in the IDLE state for long, but instead transition to other states.

RV_CCCONNSTATE_INITIATED

This state indicates that the originating end of a call has begun the process of placing a call, but has not yet begun dialing the destination address. Typically, a terminal (phone) has gone off-hook.

RV_CCCONNSTATE_DIALING

This state indicates that the originating end of a call has begun dialing a destination telephone address, but has not yet completed dialing. At this stage the user callback RvMdmTermMatchDialStringCB() is called for every digit received.

RV_CCCONNSTATE_ADDRESS_ANALYZE

This state is entered when the complete initial information package or dialing string from the originating party are available. At this stage the user callback RvMdmTermMapDialStringToAddressCB() is called. This state is exited when the routing address becomes available.

RV_CCCONNSTATE_INPROCESS

This state implies that the connection object is contacting the destination side. The contact is established as a result of the underlying protocol messages.

RV_CCCONNSTATE_CALL_DELIVERED

This state indicates that an outgoing call is being offered to the destination side (which is in an ALERTING state). For incoming calls, the connection transitions to this state after the call is answered but before transitioning to the CONNECTED state.

RV_CCCONNSTATE_OFFERED

This state indicates that an incoming call is being offered to the connection.

RV_CCCONNSTATE_ALERTING

This state implies that the Terminal is being notified of an incoming call.

RV_CCCONNSTATE_DISCONNECTED

This state implies that the connection is no longer part of the call. A connection in this state is interpreted as one that previously belonged to a call.

RV_CCCONNSTATE_CONNECTED

This state implies that a connection is actively part of a call. In common terms, two people talking to one another are represented by two connections in the CONNECTED state.

RV_CCCONNSTATE_FAILED

This state indicates that a connection to this end of the call has failed.

RV_CCCONNSTATE_REJECTED

This state indicates that an incoming call has been rejected before transitioning to the OFFERED state. For example, this may have happened because no lines were available.

RV_CCCONNSTATE_ALERTING_REJECTED

This state indicates that an incoming call has been rejected by the user while it was already in the ALERTING state. For example, because the user pressed a REJECT key.

TRANSFER STATES

These additional states describe states that the connection transitions when receiving or initiating a transfer. These states are parallel to the normal states of an incoming/outgoing call. The transfer operation involves three sides: new line A, the transferring phone; new line B, the transferree, and new line C, the transfer destination. For further details about the Transfer process, see the section [Call Transfer](#) in the [Features](#) chapter.

[RV_CCCONNSTATE_TRANSFER_INIT](#)

This state is relevant for transferring endpoint (A). The state indicates that the Transfer process has been started.

[RV_CCCONNSTATE_TRANSFER_INPROCESS](#)

This state is relevant for transferring endpoint (A). The state is relevant to [RV_CCCONNSTATE_INPROCESS](#), and indicates that the Transfer destination has been contacted.

[RV_CCCONNSTATE_TRANSFER_OFFERED](#)

This state is relevant for transfer destination endpoint (C). This state indicates that an incoming transferred call is being offered to the connection. This connection represents the transfer destination (User C in the example above) and the call replaces the existing call with the transferring party. The call moves to the [TRANSFER_ALERTING](#) state.

[RV_CCCONNSTATE_TRANSFER_ALERTING](#)

This state is relevant for transfer destination endpoint (C). This state indicates that an incoming transferred call is alerting in the connection. This connection represents the transfer destination, and the call replaces the existing call with the transferring party. The call is automatically accepted and will move to the [TRANSFER_DELIVERED](#) state.

[RV_CCCONNSTATE_TRANSFER_DELIVERED](#)

This state is relevant for transferree endpoint (B). For incoming transfer calls (User C in the example above), the connection transitions to this state after the call is answered and before transitioning to the [CONNECTED](#) state.

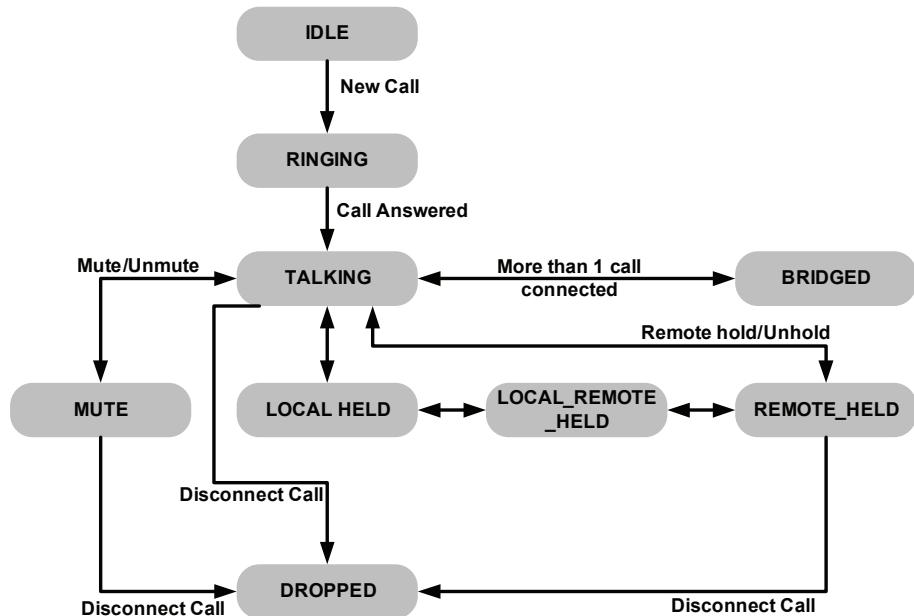


Figure 2-11 Call State Machine

TERMCNN STATES

RV_CCTERMCONSTATE_IDLE

There are no calls on the terminal; all lines are idle.

RV_CCTERMCONSTATE_RINGING

The active line is ringing with the incoming call.

RV_CCTERMCONSTATE_TALKING

The active line is connected to the remote party.

RV_CCTERMCONSTATE_HELD

The active line has put the call on Hold.

RV_CCTERMCONSTATE_REMOTE_HELD

The active line was put on Hold by the remote party.

RV_CCTERMCONSTATE_BRIDGED

At least one call is connected, in addition to the active line.

RV_CCTERMCONSTATE_DROPPED

Either the local party or the remote party is disconnected.

RV_CCTERMCONSTATE_MUTE

The active line is Mute.

RV_CCTERMCONSTATE_REMOTE_HELD_LOCAL_HELD

Both parties have put the call on Hold.

TERMINAL STATES**RV_CCTERMINAL_IDLE_STATE**

This state indicates an idle state.

RV_CCTERMINAL_CFW_ACTIVATING_STATE

This state indicates that a Call Forward activation process has started and has not yet been completed. This process starts when the user presses the Call Forward button, and is complete when:

- the user presses the Call Forward button again, to cancel the activation
- the user completes dialing the destination number
- the user answers an incoming call

When the process ends, the callback RvIppCfwActivateCompletedCB() is called. When the terminal is in this state, all other keys are blocked except for the Line and Hook keys, to enable the user to answer incoming calls.

TERMINAL EVENTS

Terminal events are events related to the terminal, which will be processed by the Connection State Machine. Terminal events can originate:

- From the phone user—for example, lifting the receiver or pressing the Line key will cause an `RV_CCTERMEVENT_OFFHOOK` event.
- From the signaling protocol—when receiving a message through the network, an event will be propagated to the appropriate Connection State Machine. For example, if an incoming call has reached the terminal, a `MAKECALL` event will be applied.
- From internal events in the state machine, by moving from one state to another, or after an operation. For example, a `MEDIAOK` event is propagated if the required media stream was created successfully (through a user callback).

`RV_CCTERMEVENT_NONE`

No action will be taken.

`RV_CCTERMEVENT_UNKNOWN`

Unknown event, will be ignored.

`RV_CCTERMEVENT_GW_ACTIVE`

This event indicates that the gateway is active. It is used by the Call Control to initialize the display. Caused by an `rvcc/ga` event.

`RV_CCTERMEVENT_OFFHOOK`

The user has gone off-hook (has lifted the phone receiver) or has pressed a Line key. These events are logically equivalent, as a call can be answered or initiated by pressing the Line key. Caused by a `kf/kd` event with keyid of `kh` for a UI termination, or an `al/of` event for an Analog termination.

`RV_CCTERMEVENT_DIALTONE`

Applied by the Call Control to move from `INITIATED` to `DIALING` event.

`RV_CCTERMEVENT_DIGITS`

The user has pressed a DTMF key. Caused by a `kp/kd` event for UI termination, or a `dd/d(n)` event for Analog termination.

RV_CCTERMEVENT_DIGIT_END

The user has released a DTMF key. Caused by a kp/ku event for UI termination.
Not used in Analog terminations.

RV_CCTERMEVENT_DIALCOMPLETED

Dialing is complete. This can either be an internal event (if the user returns from the RvMdmTermMatchDialStringCB callback with a value RV_MDMDIGITMAP_UNAMBIGUOUSMATCH or RV_MDMDIGITMAP_NOMATCH) or sent directly by the user (if the user does the match asynchronously). For the UI termination, it is caused by the kp/ce event. For the Analog termination, it is caused by the dd/ce event.

RV_CCTERMEVENT_MAKECALL

Internal Call Control event. It initiates an incoming call in the connection.

RV_CCTERMEVENT_RINGBACK

Internal Call Control event. The outgoing call has reached the other end.

RV_CCTERMEVENT_RINGING

Internal Call Control event. An incoming call is transitioning to the ALERTING state.

RV_CCTERMEVENT_CALLANSWERED

Internal Call Control event. An outgoing call has been answered in the destination end and is moving to a CONNECTED state.

RV_CCTERMEVENT_ONHOOK

The user went on-hook, thereby disconnecting the call. For the UI termination this can be caused by a kf/ku event with keyid of kh, a al/on event, or a line event (kf/ku with keyid 100n) if the line was already in a call. For the Analog termination it is caused by the al/on event.

RV_CCTERMEVENT_HOLD

Internal hold event propagated to the Connection State Machine. This can originate, for example, from a user pressing the Hold key.

RV_CCTERMEVENT_MUTE

The user pressed the Mute key. Currently not implemented.

RV_CCTERMEVENT_HOLDKEY

The user pressed the Hold key. Caused by a kf/ku event with keyid of kl.

RV_CCTERMEVENT_UNHOLD

Internal Unhold event propagated to the Connection State Machine. This can originate, for example, from a user pressing the Line key on a line in a HELD state.

RV_CCTERMEVENT_CONFERENCE

The user pressed the Conference key. Caused by a kf/ku event with keyid of kc. The first time will activate an additional line to call the added party. The second time will connect all parties in the conference.

RV_CCTERMEVENT_TRANSFER

The user pressed the Transfer key. Caused by a kf/ku event with keyid of kt. The first time will activate an additional line to call the transfer destination. The second time will complete the transfer and drop the user from the transfer.

RV_CCTERMEVENT_TRANSFER_INIT

This event indicates to transferring endpoint (A) to start the Transfer process, i.e., to send the transferree endpoint (B) a signaling message indicating that it should establish a call with the Transfer destination endpoint.

RV_CCTERMEVENT_BLIND_TRANSFER

The user pressed the Blind Transfer key. Caused by a kf/ku event with keyid of kbt.

RV_CCTERMEVENT_LINE

The user pressed the Line key. Caused by a kf/ku event with keyid of l00n.

RV_CCTERMEVENT_LINEOTHER

This is an internal event. The user pressed the Line key, and there is a connected call (either on hold or not) in a different line.

RV_CCTERMEVENT_HEADSET

The user pressed the Headset key. Caused by a kf/ku event with keyid of ht.

RV_CCTERMEVENT_HANDSFREE

The user pressed the Handsfree key. Caused by a kf/ku event with keyid of hf.

RV_CCTERMEVENT_AUDIOHANDSET

This is an internal event. The event indicates the Call Control to make the Handset the active audio termination, meaning to move the media from the current active termination to the new one, etc.

RV_CCTERMEVENT_AUDIOHANDSFREE

This is an internal event. The event indicates to the state machine to make the Speaker the active audio termination, i.e., to move the media from the current active termination to the new one, etc.

RV_CCTERMEVENT_FAILGENERAL

This is an internal event. The event indicates a general failure, while the reason code specifies the reason for the failure.

RV_CCTERMEVENT_MEDIAOK

Internal Call Control event. Creating or modifying a media stream on a termination has succeeded.

RV_CCTERMEVENT_MEDIAFAIL

Internal Call Control event. Creating or modifying a media stream on a termination has failed.

RV_CCTERMEVENT_DISCONNECTING

Internal Call Control event. Causes the connection to disconnect from the call.

RV_CCTERMEVENT_DISCONNECTED

Internal Call Control event. The connection has been disconnected. Release resources.

RV_CCTERMEVENT_INCOMINGCALL

This is an internal event. The event is usually caused by an incoming signaling message indicating the establishment of a new call.

RV_CCTERMEVENT_REJECTCALL

Internal Call Control event. Rejects an incoming call before the OFFERED state. For example, because there are no available lines.

RV_CCTERMEVENT_TRANSFER_OFFERED

Internal Call Control event. An incoming transferred call is being offered to the connection. This connection represents the transfer destination. The call will replace the existing call with the transferring party.

RV_CCTERMEVENT_REMOTE_DISCONNECTED

Internal Call Control event. The other party has disconnected the call.

RV_CCTERMEVENT_REJECT_KEY

The user has rejected an incoming call on a given line. Caused by rvcc/reject event.

RV_CCTERMEVENT_MEDIANOTACCEPTED

This is an internal event. The event indicates that the media offered by the remote party is not supported by the local party. Processing the event will result in a local warning tone and an outgoing signaling message, indicating that the call is rejected with reason 415—Media Not Supported for SIP.

RV_CCTERMEVENT_MODIFYMEDIA

This event is relevant for SIP Phone only. The event indicates the beginning of a dynamic media change process and is caused by the user application calling rvMdmTermModifyMedia(). Processing the event will result in a Re-Invite message being sent that includes the new media.

RV_CCTERMEVENT_MODIFYMEDIA_DONE

This is an internal event. The event is processed when the state machine has finished processing a dynamic media change and will result in notifying the user application about whether or not the process has been completed successfully.

RV_CCTERMEVENT_USER

All values higher than this value can be used by the user and will be ignored by the Call Control.

EVENT REASON

The event reason field provides additional information to the Call Control about the events.

RV_CCCAUSE_NORMAL

This reason is used when no specific reason is required.

RV_CCCAUSE_INCOMING_CALL

The call is an incoming call.

RV_CCCAUSE_OUTGOING_CALL

The call is an outgoing call.

RV_CCCAUSE_CALL_WAITING

The incoming call is a call waiting (there is an active call in the terminal already).

RV_CCCAUSE_BUSY

The call is not completed because the party is busy (no lines available).

RV_CCCAUSE_TRANSFER

The reason for the call is transfer.

RV_CCCAUSE_LOCAL_HOLD

The call was put on hold by the local user.

RV_CCCAUSE_REMOTE_HOLD

The call was put on Hold by the remote party. The event is caused by an incoming signaling message indicating that the call should be put on Hold.

RV_CCCAUSE_NEW_CALL

The call is a new call.

RV_CCCAUSE_MEDIA_NOT_SUPPORTED

The phone does not support the required media parameters.

RV_CCCAUSE_REORDER_TONE

Indicates a general failure, will cause a warning tone to be heard.

RV_CCCAUSE_NOT_FOUND

Reason for rejecting an incoming call - destination was not found.

RV_CCCAUSE_CALL_CANCELLED

Call was canceled by the local party.

RV_CCCAUSE_EVENT_BEGIN

This reason indicates that Key Down was pressed for a digit.

RV_CCCAUSE_EVENT_END

This reason indicates that Key Up was pressed for a digit.

RV_CCCAUSE_OPERATION_SUCCEEDED

The process ended successfully.

RV_CCCAUSE_OPERATION_FAILED

The process ended with failure.

RV_CCCAUSE_AUTH_FAIL

(SIP only) Authentication failed.

RV_CCCAUSE_UNKNOWN

Reason is unknown.

RV_CCCAUSE_UNHOLD

Call was released from Hold.

RV_CCCAUSE_RESOURCES_NOT_AVAILABLE

Operation failed due to lack of resources.

RV_CCTERMEVENT_ONHOOK_OTHER

Internal Call Control event.

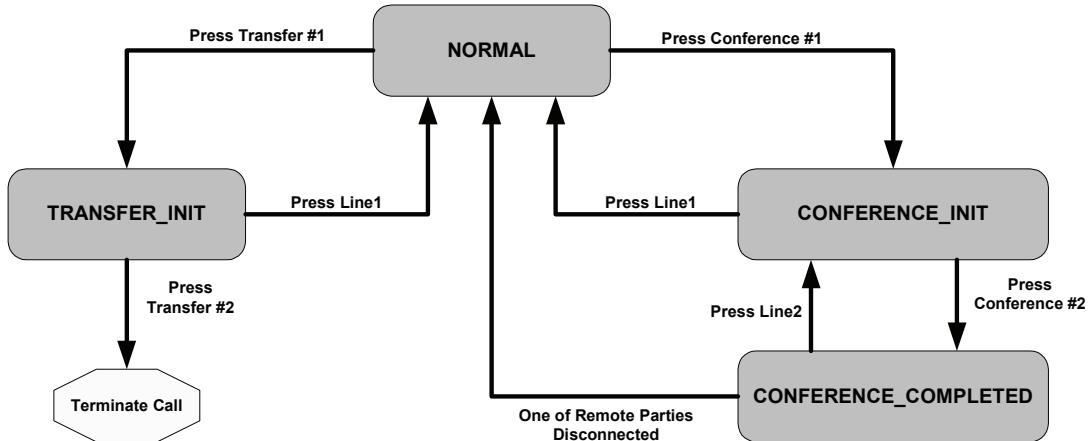
RV_CCTERMEVENT_REDIAL

This event is sent to activate Redial functionality.

RV_CCTERMEVENT_CFW

This event is sent to activate Call Forward functionality.

CALL STATE MACHINE

**Figure 2-12** Call State Machine

CALL STATES

The call has four states:

[RV_CCCALLSTATE_NORMAL](#)

This state indicates a basic call with two parties. When the call is in this state, it includes one MDM connection and one network connection.

[RV_CCCALLSTATE_CONFERENCE_INIT](#)

This state indicates starting a conference process. The call moves to this state after the user first presses the Conference key. In this state the call contains three connections: Two MDM connections (one belongs to the original call, the other to the new call) and one network connection belonging to the original call.

[RV_CCCALLSTATE_CONFERENCE_COMPLETED](#)

This state indicates the completion of the conference process, meaning that all three parties have been connected. In this state the call includes four connections: Two MDM connections and two network connections.

RV_CCCALLSTATE_TRANSFER_INIT

This state indicates the start of a Transfer process. The call moves to this state after the user first presses Transfer. In this state the call still contains two connections: One MDM connection and one network connection (belonging to the original call), and a new call is created. The relation between the calls will be through the "transferLine" field in the MDM connections.

2.6 THREADING MODEL

This section describes the thread model used by the Multimedia Terminal Framework, and the interaction between the threads and the user thread. The Multimedia Terminal Framework is a multi-threaded application. It interacts with the user thread by calling APIs and callbacks. When the Multimedia Terminal Framework is initialized, a Terminal Framework thread is created. The Terminal Framework thread interacts with the user thread as follows:

- User thread to Terminal Framework thread—Two types of interaction are possible:
 - Asynchronous interaction (non blocking)—When the user thread sends events to the Multimedia Terminal Framework by calling `rvMdmTermProcessEvent()`, the events are put in the Multimedia Terminal Framework events queue so that the user thread will return immediately instead of waiting for processing to end. The Multimedia Terminal Framework will process the events later and will notify the user of the results when necessary.
 - Synchronous interaction (blocking)—When the user thread calls Terminal Framework APIs (except for `rvMdmTermProcessEvent()`, which sends events), it will return when processing has ended. API processing will run in the context of the user application.

- Terminal Framework thread to user thread—The Multimedia Terminal Framework calls user application callbacks as part of event processing. These callbacks are an implementation of the user application and may include calls to the user application APIs.

Note

- All calls to Terminal Framework APIs should be done from the same user thread.
- A user thread that calls Multimedia Terminal Framework APIs should have a minimum Stack size of 40K.

Figure 2-13 illustrates the relations between the threads:

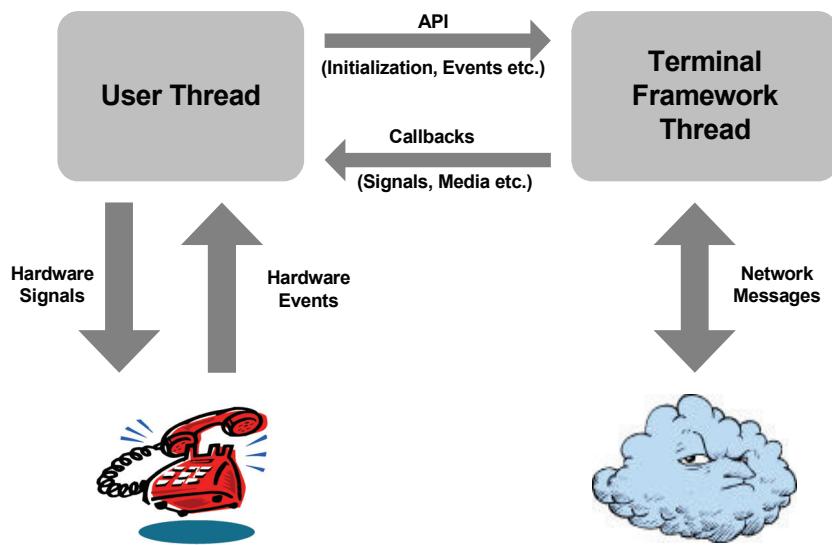


Figure 2-13 Thread Relations

PART 2: USING THE MULTIMEDIA TERMINAL FRAMEWORK

3

BUILDING YOUR APPLICATION

3.1 WHAT'S IN THIS CHAPTER

This chapter explains how to initialize the Multimedia Terminal Framework and build an application:

- Introduction
- Step 1: Initialization
- Step 2: Playing Signals
- Step 3: Reporting Events
- Step 4: Integrating Media

3.2 INTRODUCTION

The Multimedia Terminal Framework implements a standalone, feature-rich phone that interacts with the user through APIs and callbacks. Once the Multimedia Terminal Framework has been initialized, the application reports all user events through the Multimedia Terminal Framework interface and executes callback functions when called.

You can build an application based on the Multimedia Terminal Framework by performing the following steps:

- [Step 1: Initialization](#)—this is done using a set of APIs for the initialization and configuration of the Multimedia Terminal Framework and the SIP Stack.
- [Step 2: Playing Signals](#)—the Multimedia Terminal Framework provides a set of callbacks to be implemented by the user application, which are called by the Multimedia Terminal Framework whenever the user application is required to play a signal or set an indicator on the terminal.

Step 1: Initialization

- [Step 3: Reporting Events](#)—the Multimedia Terminal Framework provides API to be called when the user application wants to send an event to the Multimedia Terminal Framework.
- [Step 4: Integrating Media](#)—the Multimedia Terminal Framework provides a set of callbacks to be implemented by the user application, which are called by the Multimedia Terminal Framework whenever the user application is required to perform a media operation.

3.3 STEP 1: INITIALIZATION

INITIALIZATION STEPS

This section describes how to initialize and configure the Multimedia Terminal Framework.

Initializing the Multimedia Terminal Framework is done in a series of steps described below. The description of each step is followed by an example from the Multimedia Terminal Framework sample application. For further details, see the [Sample Application](#) chapter.

1. INITIALIZE THE SYSTEM

Initializing the Terminal Framework system is done by calling [rvIppSipSystemInit\(\)](#). The Terminal Framework system must be initialized before any Terminal Framework or Stack API is called. The function should be called once and will perform multiple one-time initializations.

EXAMPLE

[/*Initialize Terminal Framework System*/](#)

```
rvIppSipSystemInit();
```

2. INITIALIZE THE LOG

The Multimedia Terminal Framework provides a log that is initialized by calling the following function:

```
IppLogInit(IppLog SourceFilterElm* ippFilters)
```

For more information, see the [Logging](#) chapter.

3. INITIALIZE CONFIGURATION STRUCTURE

This is done by calling [rvIppSIPInitConfig\(\)](#). This function sets default values in the Terminal Framework configuration parameters. After this function is called, you can change the configuration parameters according to your requirements.

4. INITIALIZE THE PROTOCOL STACK

5. CONSTRUCT THE FRAMEWORK

6. REGISTER MEDIA CALLBACKS

7. CREATE CLASSES

EXAMPLE

/*Initialize Terminal Framework Configuration Structure*/

```
RvIppSipPhoneCfg cfg;
rvIppSipInitConfig(&cfg, sizeof(cfg));
```

This step is performed by calling **rvIppSipStackInitialize()**. As a first step in working with the Multimedia Terminal Framework, you must initialize the SIP Protocol Stack. The Multimedia Terminal Framework initializes the Stack with the values configured by the user. This function returns the Stack handle as an output parameter.

This is done by calling **rvIppSipPhoneConstruct()**. These functions receive as an argument a pointer or handle to a previously initialized Protocol Stack and the Multimedia Terminal Framework configuration structure.

Register callbacks for media control and set user data to be received through these callbacks. For more information about the implementation of these callbacks, see the section [Step 4: Integrating Media](#).

- rvMdmTermMgrSetUserData();
- rvMdmTermMgrRegisterSelectTermCB();
- rvMdmTermMgrRegisterDeleteEphTermCB();
- rvMdmTermMgrRegisterConnectCB();
- rvMdmTermMgrRegisterDisconnectCB();
- rvMdmPackageRegisterEventParametersCB();

Termination classes represent types of terminations. Once created, a termination class can be configured by setting various callbacks and specifying the packages supported by the class. The following steps should be taken to construct and initialize Terminal Framework classes:

- **Construct termination classes**

This is done by calling **rvMdmTermMgrCreateTermClass()**. This function returns a pointer to the newly-created class, which you should store and use later when you register terminations (see the section [Registering Terminations](#)). You should create four types of classes, one for each termination type:

- **User Interface (UI) class**—The class with which you will register the UI termination (see the section [Registering Terminations](#)).

- **AudioTransducer (AT) class**—The class with which you will register all AT terminations that are supported by your hardware (e.g., Handset) (see the section [Registering Terminations](#)).
 - **Analog phone class**—The class with which you will register Analog terminations (see the section [Registering Terminations](#)).
 - **RTP class**—The class with which you will register RTP Terminations (see the section [Step 4: Integrating Media](#)).
- **Register class callbacks**
- You can register class callbacks by registering the functions used to apply signals and control the media. These callbacks will be called by the MDM when it needs to perform an operation (apply a signal or manipulate a media stream) on a termination. The following callbacks should be registered with UI, AT, and Analog classes:
- `rvMdmTermClassRegisterStartSignalCB()`; (For more information on callback implementation, see the section [Signal Callbacks](#).)
 - `rvMdmTermClassRegisterStopSignalCB()`; (For more information on callback implementation, see the section [Signal Callbacks](#).)
 - `rvMdmTermClassRegisterPlaySignalCB()`; (For more information on callback implementation, see the section [Signal Callbacks](#).)
 - `rvMdmTermClassRegisterCreateMediaCB()`; (For more information on callback implementation, see the section [Step 4: Integrating Media](#).)
 - `rvMdmTermClassRegisterModifyMediaCB()`; (For more information on callback implementation, see the section [Step 4: Integrating Media](#).)
 - `rvMdmTermClassRegisterDestroyMediaCB()`; (For more information on callback implementation, see the section [Step 4: Integrating Media](#).)

- ❑ rvMdmTermClassRegisterMapDialStringToAddressCB();
(For more information on callback implementation, see the section [Matching Digits to an Address](#).)

The following callbacks should be registered with the RTP class:

- ❑ rvMdmTermClassRegisterCreateMediaCB(); (For more information on callback implementation, see the section [Step 4: Integrating Media](#).)
- ❑ rvMdmTermClassRegisterModifyMediaCB(); (For more information on callback implementation, see the section [Step 4: Integrating Media](#).)
- ❑ rvMdmTermClassRegisterDestroyMediaCB(); (For more information on callback implementation, see the section [Step 4: Integrating Media](#).)

■ Set Packages

To register and set packages for each class, call the following APIs:

- ❑ rvMdmTermMgrRegisterIPPhoneUiPackages()
- ❑ rvMdmTermClassSetIPPhoneUIPackages()
- ❑ rvMdmTermClassSetAudioTransducerPackages()
- ❑ rvMdmTermClassSetAnalogLinePackages()

■ Set user data for each class

`rvMdmTermClassSetUserData()`

This user data is received when the callbacks registered above are called.

8. LOAD MEDIA CAPABILITIES

The Multimedia Terminal Framework must be configured with the media capabilities of your hardware. Media capabilities describe all codecs along with their associated media attributes supported by your hardware. These capabilities are presented to you each time the Terminal Framework creates media for a new call, and you can choose the parameters to be used for that call.

The following steps are required to load media capabilities to the Multimedia Terminal Framework:

1. Build your media capabilities into an SDP structure using SDP Stack APIs (see example below). Since the RTP session is not yet open at this stage, the local IP address and port are not yet determined and a "\$" or "65535" sign will be used instead. The

Step 1: Initialization

function should be called for *both* audio transducer/analog and RTP termination classes.

2. Load the SDP object to the Multimedia Terminal Framework by calling:

```
void rvMdmTermClassAddMediaCapabilities(
    RvMdmTermClass *           c,
    const RvSdpMsgList *       localDescr,
    const RvSdpMsgList *       remoteDescr,
    const RvMdmParameterList* localProperties);
```

The parameters for this function are:

c—A pointer to the RTP or Audio or Analog class.

localDescr—As described above, to fill in this parameter the user application should build a descriptor that matches its local capabilities as derived from the phone configuration. For more information, see the SDP User Guide and the example below.

remoteDescr—Not used in the current version. Should be NULL.

localProperties—Not used in the current version. Should be NULL.

To be able to modify a capability rather than add a new one, the application must first delete the existing capabilities, using the function `rvMdmTermClassClearMediaCapabilities()`.

3. Inform the Multimedia Terminal Framework that the media capabilities have been updated by calling:

```
void rvMdmTermMgrMediaCapabilitiesUpdated(
    RvMdmTermMgr*      mgr,
    RvMdmTermClass *   rtpClass);
```

with the following parameters:

mgr—A pointer to the Termination Manager

rtpClass—A pointer to the RTP class.

EXAMPLE

/* Construct Classes */

/*Create Classes*/

```
analogClass = rvMdmTermMgrCreateTermClass(termMgr);  
atClass = rvMdmTermMgrCreateTermClass(termMgr);  
uiClass = rvMdmTermMgrCreateTermClass(termMgr);  
rtpClass = rvMdmTermMgrCreateTermClass(termMgr);
```

/*Register Class Callbacks*/

```
rvMdmTermClassRegisterStartSignalCB(uiClass, userUIStartSignalCB);  
rvMdmTermClassRegisterStartSignalCB(atClass, userATStartSignalCB);  
rvMdmTermClassRegisterStartSignalCB(analogClass,  
    userAnalogStartSignalCB);  
..... etc.
```

```
rvMdmTermClassRegisterCreateMediaCB(rtpClass, userRtpCreateMedia);  
rvMdmTermClassRegisterModifyMediaCB(rtpClass, userRtpModifyMedia);  
rvMdmTermClassRegisterDestroyMediaCB(rtpClass,  
    userRtpDestroyMedia);
```

/*Set Packages*/

```
rvMdmTermMgrRegisterIPPhoneUiPackages(termMgr);  
rvMdmTermClassSetIPPhoneUIPackages(uiClass);  
rvMdmTermClassSetAudioTransducerPackages(atClass);  
rvMdmTermClassSetAnalogLinePackages(analogClass);
```

/*Set User Data*/

```
rvMdmTermClassSetUserData(analogClass, userClassData);  
rvMdmTermClassSetUserData(atClass, userClassData);  
rvMdmTermClassSetUserData(uiClass, userClassData);
```

CONFIGURATION

This section explains the Multimedia Terminal Framework configuration parameters. To configure the SIP Phone, the following parameters should be set in the structure RvIppSipPhoneCfg (located in the rvSipControlApi.h file) before calling rvIppSipStackInitialize().

Note The only mandatory parameter to be set is localAddress. All other parameters are optional and will be set to the default if they are not configured by the user application.

stackTcpPort

The TCP port on which the Stack listens.

Default: 5060.

stackUdpPort

The UDP port on which the Stack listens.

Default: 5060.

userDomain

This domain name will be sent in the From header of outgoing INVITE messages in either of the following cases: If the Registrar address parameter is configured as an IP address, or if the Registrar address is not configured at all.

localAddress

Local IP address. This parameter is mandatory and may not be left empty.

registrarAddress

The Registrar IP address or domain name. If this parameter is not set, Registration messages will not be sent.

Default: NULL.

registrarPort

The number of the Port on which the Registrar listens.

Default: 5060.

outboundProxyAddress

The IP address or domain name of the outbound Proxy. If this parameter is set, all outgoing messages (including Registration messages) will be sent to this Proxy according to the Stack behavior.

Default: NULL.

Note To configure a host name instead of an IP address, set this parameter to NULL and configure the outboundProxyHostName parameter in the SIP Stack (see the chapter [Extensibility](#), section).

outboundProxyPort

The number of the Port on which the outbound Proxy listens.

Default: 5060.

transportType

The Transport type of the outgoing messages. Valid values are:

- RVSIP_TRANSPORT_UNDEFINED—The transport type of outgoing messages will be determined according to the DNS resolution of the destination.
- RVSIP_TRANSPORT_TCP—The transport type of outgoing messages will be TCP.
- RVSIP_TRANSPORT_UDP—The transport type of outgoing messages will be UDP.

Default: RVSIP_TRANSPORT_UDP.

maxCallLegs

The maximum number of call-legs the Stack can handle simultaneously. This number should match the maximum number of simultaneous calls required by your application. When setting the value of this parameter, note that the SIP Stack does not release the call-leg object immediately but only after (configured) timeouts. Therefore, this parameter should be set to more than the maximum number of calls. (For more information, see the RADVISION SIP Stack documentation.) The value of this parameter is used by the Multimedia Terminal Framework to set the default value of the SIP Stack parameter maxTransactions, so that maxTransactions = maxCallLegs * 5 (as recommended

by the SIP Stack). The Multimedia Terminal Framework sets the values of other SIP Stack parameters to their default values (see the chapter [Optimizations](#)). For information on how to change these values, see the chapter [Extensibility](#).

maxRegClients

The maximum number of RegClient objects the Stack can handle simultaneously. This number should match the maximum number of physical terminations required by your application. (The number of physical terminations in IAD applications is the number of analog lines. In other types of applications it is 1.) When setting the value of this parameter, note that the SIP Stack does not release the reg-client object immediately but only after (configured) timeouts. Therefore, this parameter should be set to more than the maximum number of terminations. (For more information, see the RADVISION SIP Stack documentation.) The Multimedia Terminal Framework sets the values of other SIP Stack parameters to their default values (see the chapter [Optimizations](#)). For information on how to change these values, see the chapter [Extensibility](#).

tcpEnabled

This parameter indicates support for TCP. When set to RV_TRUE, the Terminal Framework will support a TCP connection. When set to RV_FALSE, the Terminal Framework will not support a TCP connection (incoming TCP messages will be ignored).

Default: RV_TRUE.

priority

The priority of the Terminal Framework task. It is recommended to use one of the following values, which define the appropriate value according to the operating system:

- RV_THREAD_PRIORITY_MAX
- RV_THREAD_PRIORITY_DEFAULT
- RV_THREAD_PRIORITY_MIN

These values are defined in the file *rvthread.h*.

Default: RV_THREAD_PRIORITY_DEFAULT

username

Used for Authentication. This parameter can also be configured for each termination separately (see section [Registering Terminations](#) later in this chapter). If this parameter is configured, it will be used for all terminations whose username has not been configured. If this parameter is left empty and is not configured in a termination, the Authentication header will not be sent with the Registration request.

Default: NULL.

password

Used for Authentication. This parameter can also be configured for each termination separately (see section [Registering Terminations](#) later in this chapter). If this parameter is configured, it will be used for all terminations whose username has not been configured. If this parameter is left empty and is not configured in a termination, the Authentication header will not be sent with the Registration request.

Default: NULL.

autoRegister

If set to RV_TRUE, the Terminal Framework will send the initial registration request to the Registrar for every termination that registers to the Multimedia Terminal Framework. If set to RV_FALSE, the initial registration request will not be sent. It can be sent manually at any time by calling `rvMdmTermMgrRegisterAllTermsToNetwork_()` or `rvMdmTermMgrRegisterTermToNetwork_()`. Regardless of the value of this parameter, the Terminal Framework will send re-Registration requests.

Default: RV_TRUE.

registrationExpire

The timeout (in seconds) for sending Re-registration requests to the Registrar.

Default: 60,000 seconds.

unregistrationExpire

This parameter defines the timeout (in seconds) to wait for a reply from the Registrar, after an Unregistration request was sent. The Unregistration request is sent when the user application unregisters a termination from the Multimedia

Step 1: Initialization

Terminal Framework by calling *rvMdmTermMgrUnregisterTermination()*. Once this function is called, an un-registration process begins. This process is asynchronous.

If the termination is registered with a Registrar, the Multimedia Terminal Framework will send an Unregistration request to the Registrar and wait for a reply. The *UnregistrationExpire* parameter defines the timeout to wait for a reply. The process will continue only when the timeout expires or when a reply is received from the Registrar.

When either of these scenarios occurs, the Multimedia Terminal Framework will call the user callback *RvMdmTermUnregisterTermCompletedCB()* to notify the user application that the unregistration process has ended.

Default: 20 seconds.

referTimeout

The timeout (in milliseconds) for waiting for NOTIFY after sending REFER, before disconnecting the call-leg.

Default: 2,000 milliseconds.

debugLevel

This parameter is not used.

logOptions

This parameter defines log options for the Sip Stack. See the [Logging](#) chapter.

Default: NULL.

dialToneDuration

Duration of Dial Tone signal (in milliseconds) when going off-hook. When the user goes off-hook and timeout expires, Dial Tone will be stopped and the connection will disconnect. A value of 0 indicates an infinite Dial Tone.

Default: 30,000 milliseconds.

watchdogTimeout

This parameter indicates the time interval (in seconds) used for the periodical printing of Multimedia Terminal Framework and SIP Stack resources to the log. If this parameter is set to zero, the timer is disabled.

Multimedia Terminal Framework resources will be printed to the log only if the following is defined:

- The Multimedia Terminal Framework is **not** compiled with the flag RV_MTF_PERFORMANCE_ON
 - Log configuration includes INFO level in IPP_UTIL module
- SIP Stack resources will be printed to the log only if the following is defined:
- Log configuration includes INFO level in IPP_SIPCTRL module

Default: 0.

callWaitingReply

When the incoming call is a Call Waiting call, this parameter indicates which SIP message will be sent as a reply to the Invite. Possible values are:

- RV_REPLY_RINGING=180
- RV_REPLY_QUEUED=182

Defined in *rvSipControlApi.h*.

Default: RV_REPLY_QUEUED

outOfBandDTMF

When this parameter is set to RV_TRUE, out-of-band DTMF is enabled. For more information, see the [Features](#) chapter, section [Out-of-Band DTMF](#).

transportTlsCfg

For information see the [Features](#) chapter, section [TLS](#).

cfwCallCfg

For information see the [Features](#) chapter, section [Call Forward](#).

RvSdpStackCfg

This parameter is an SDP structure that enables the configuration of the SDP Stack by the user application. The Multimedia Terminal Framework will use this parameter for initialization of the SDP Stack as part of Multimedia Terminal Framework initialization. This parameter enables the configuration of log filters by the user application.

connectOn180

When this parameter is set to True, media will be connected both in 180 Ringing, 183 Session In Progress and 200 OK. When set to False, media will be connected in 183 Session In Progress and 200 OK only.

REGISTERING TERMINATIONS

Terminations used by the phone must be registered with the Multimedia Terminal Framework. Physical terminations are registered after the Multimedia Terminal Framework has been initialized and started. Ephemeral terminations are registered during a call, per Multimedia Terminal Framework request. A termination is identified by its name (termination ID) and each termination is registered only once. The section below describes how terminations are configured and registered to the Multimedia Terminal Framework.

For more information about terminations, see the chapter [Architecture](#).

TERMINATION CONFIGURATION

For each physical termination, a set of parameters can be configured. This section explains how the user application can configure these parameters and how they are used by the Multimedia Terminal Framework.

Note Configuring terminations is optional.

Perform the following steps to configure a termination:



Step 1: Construct structure RvMdmTermDefaultProperties by calling the following function:

```
void rvMdmTermDefaultPropertiesConstruct(  
    RvMdmTermDefaultProperties* x);
```



Step 2: Set values in the structure RvMdmTermDefaultProperties.

The following optional parameters may be configured:

Digitmap

A user-defined digitmap can be built for each termination. The Multimedia Terminal Framework uses the digitmap when the user dials digits to establish a call. If the digitmap is not configured, the Multimedia Terminal Framework will

use its default. For more information, see the section [Step 3: Reporting Events](#). See also the example IppSipSample in the Terminal Framework sample application.

Perform the following steps to configure a digitmap for the termination:

1. Construct structure RvMdmDigitMap by calling the following function:

```
RvMdmDigitMap *rvMdmDigitMapConstruct (RvMdmDigitMap
*x) ;
```

2. Build the structure RvMdmDigitMap, using RvMdmDigitMap APIs.
3. Set digitmap in the structure RvMdmTermDefaultProperties by calling the following function:

```
void rvMdmTermDefaultPropertiesSetDigitMap(
    RvMdmTermDefaultProperties* x,
    RvMdmDigitMap* digitMap,
    const char* name) ;
```

4. Destruct the structure RvMdmDigitMap by calling the following function:

```
void rvMdmDigitMapDestruct (RvMdmDigitMap *x) ;
```

[Username and password](#)

The username and password parameters are used to authenticate an endpoint to a server. The Multimedia Terminal Framework will use them when sending a Client Authentication as a response to Unauthorized or Proxy Authentication Requested (401/407). Username and password may also be configured per application in the general configuration (see the section [Configuration](#)). When these parameters are configured, they override the username and password set in the general configuration.

Perform the following steps to configure username and password for the termination:

1. Set a username in the structure RvMdmTermDefaultProperties by calling the following function:

```
void rvMdmTermDefaultPropertiesSetUsername (
    RvMdmTermDefaultProperties* x,
    char* username) ;
```

Step 1: Initialization

2. Set a password in the structure RvMdmTermDefaultProperties by calling the following function:

```
void rvMdmTermDefaultPropertiesSetPassword(  
    RvMdmTermDefaultProperties* x,  
    char* password);
```

Phone numbers

Note Phone numbers are currently not supported for the SIP Phone.

Name Presentation

Name Presentation consists of a name and of permissions to display that name in outgoing and incoming calls.

Perform the following steps to configure Name Presentation for the termination:

1. Construct the structure RvMdmTermPresentationInfo by calling the following function:

```
RvMdmTermPresentationInfo*  
rvMdmTermPresentationInfoConstruct(  
    RvMdmTermPresentationInfo* x);
```

2. Build the structure RvMdmTermPresentationInfo, using RvMdmTermPresentationInfo APIs defined in the file *rvmddmobject.h*.

3. Set phone numbers in the structure RvMdmTermDefaultProperties by calling the following function:

```
RvBool rvMdmTermPropertiesSetPresentationInfo(  
    RvMdmTermDefaultProperties* termProperties,  
    RvMdmTermPresentationInfo* presentationInfo);
```

4. Destruct the structure RvMdmTermDefaultProperties:

```
void rvMdmTermDefaultPropertiesDestruct(  
    RvMdmTermDefaultProperties* x);
```

TERMINATION REGISTRATION

The user application registers UI, AudioTransducer and video and analog terminations to the Multimedia Terminal Framework after initialization. For more information about termination types, see the chapter [Architecture](#).

Perform the following steps to register a physical termination to the Multimedia Terminal Framework:

1. Construct the structure RvMdmTermDefaultProperties by calling the following function:

```
void rvMdmTermDefaultPropertiesConstruct(
    RvMdmTermDefaultProperties* x);
```

2. Configure the termination (optional). See above for details.
3. Register the termination by calling one of the following functions:

```
RvMdmTerm* rvMdmTermMgrRegisterPhysicalTermination(
    RvMdmTermMgr*           mgr,
    RvMdmTermClass *        c,
    const char*             id,
    struct RvMdmTermDefaultProperties_* termProperties,
    RvMdmServiceChange      *sc)
—or—
RvMdmTerm*
rvMdmTermMgrRegisterPhysicalTerminationAsync(
    RvMdmTermMgr*           mgr,
    RvMdmTermClass *        c,
    const char*             id,
    struct RvMdmTermDefaultProperties_* termProperties,
    void*                  userData)
```

with the following arguments:

- mgr**—Pointer to the Termination Manager.
- c**—Pointer to a previously-created termination class to which the termination belongs.
- id**—A unique name for the termination.
- TermProperties**—Pointer to the configuration structure. This structure should be constructed and may be set with values (see above for details).
- userData**—Void pointer to user data. This pointer will be sent to the application when the callback RvMdmTermRegisterPhysTermCompletedCB() is called.

Return Value

A pointer to RvMdmTerm, which is later used to report events on the termination and receive signals and media commands.

Step 1: Initialization

4. Set user data. The application can associate application-specific data to a termination for later use by calling the following function:

```
void rvMdmTermSetUserData(  
    RvMdmTerm* term, void * data);
```

with the following arguments:

- ❑ **term**—Pointer received by the Multimedia Terminal Framework when the termination was registered (see Step 3).
- ❑ **data**—Pointer to user data. This user data can later be retrieved from the termination pointer by calling `rvMdmTermGetUserData()`.

Example

```
RvMdmTermDefaultProperties termProperties;  
  
/* Construct structure RvMdmTermDefaultProperties */  
rvMdmTermDefaultPropertiesConstruct (&termProperties);  
  
/* Set values in RvMdmTermDefaultProperties...*/  
  
/* Register termination*/  
RvMdmTerm* termination = rvMdmTermMgrRegisterPhysicalTermination  
(&term_mgr, class, "myTerm", &termProperties, NULL);  
  
/* Set user data*/  
rvMdmTermSetUserData(ui_termination, my_ui_term);  
  
/* Destruct structure RvMdmTermDefaultProperties */  
rvMdmTermDefaultPropertiesDestruct (&termProperties);
```

STARTING THE SYSTEM

Having done the above you can now start the system by calling:

```
rvMdmTermMgrStart (RvMdmTermMgr *mgr,  
    RvMdmServiceChange* sc, int delay)
```

with the following parameters:

- **mgr**—Pointer to the Termination Manager.
- **sc**—Must always be NULL.
- **delay**—Must always be 0.

After this step the Multimedia Terminal Framework will be ready to process events and network messages.

3.4 STEP 2: PLAYING SIGNALS

Signals are used by the Multimedia Terminal Framework to apply commands (such as the playing of tones) to the user application, which in turn applies the commands to the Terminal Framework hardware. This section describes how these commands are implemented in your application.

SIGNAL CALLBACKS

The MDM applies a signal to the application through a user callback, such as Ring or Dialtone. The Multimedia Terminal Framework applies signals by calling the following user callbacks:

- **rvMdmTermStartSignalCB()**

This callback indicates to the user to start playing a signal. It is used for signals that are stopped when the callback `rvMdmTermStopSignalCB()` is called.

- **rvMdmTermPlaySignalCB()**

This callback indicates to the user to play a signal. It is used for signals that need not be stopped (`rvMdmTermStopSignalCB` will not be called for these signals).

- **rvMdmTermStopSignalCB()**

This callback indicates to the user to stop playing a signal. This occurs when one of the following happens:

- The signal time-out has elapsed
- A new signal is received
- An event is detected on the termination

This callback is not called for signals that start playing when `rvMdmTermPlaySignal()` is called. Such signals should be played for a short period and should stop by themselves.

CALLBACK PARAMETERS

The above callbacks receive the following parameters:

1. **Term**

A pointer to the termination (`RvMdmTerm*`). Using this pointer, you can get a pointer to your own data associated with the termination by calling `rvMdmTermGetUserData()`.

2. **Signal**

A pointer to the signal (`RvMdmSignal*`). You can use Signal APIs (file `rvmmdm.h`) to extract information, for example:

- `rvMdmSignalGetPkg()`—Get the package of the signal
- `rvMdmSignalGetId()`—Get the ID of the signal.

- rvMdmSignalGetArguments()—Get the parameters of the signal
- rvMdmParameterListGet2() and
rvMdmParameterValueGetValue()—Obtain individual parameters

3. reportCompletion

This parameter is not used.

4. mdmError

This parameter is not used.

Example

```
void userStartSignal(RvMdmTerm *term, RvMdmSignal *signal, OUT
RvMdmError *mdmError)
{

    UserTerm                      *userTerm=NULL;
    const RvMdmParameterList*      params;
    const char                     *pkg, *id;
    int                            userSignal;

    /*Get user data associated with this termination*/
    userTerm = (UserTerm *)rvMdmTermGetUserData(term);
    pkg = rvMdmSignalGetPkg(signal);
    id = rvMdmSignalGetId(signal);
    args = rvMdmSignalGetArguments(signal);

    /* Map signal parameters to the user signal ...*/
    if (!strcmp(pkg, "cg"))
    {
        if (!strcmp(id, "dt"))
            userSignal = DIAL_TONE;
        if (!strcmp(id, "rt"))
            userSignal = RINGING_TONE;
        if (!strcmp(id, "bt"))
            userSignal = BUSY_TONE;
        ....
    }
    /*Play signal on hardware*/
    userStartSignal(userSignal);
}
```

Step 2: Playing Signals

LIST OF SIGNALS

[Table 3-1](#) presents a list of signals and their parameters supported by the Multimedia Terminal Framework.

Table 3-1 *List of Signals*

No	Signal	Package	Id	Parameter	Value
1	Dial Tone	cg	dt	--	--
2	Ringing Tone	ind	is	Indid	ir
				State	on – set indicator on off – set indicator off
				distRing (optional)	Alternative ringing as NULL terminated string
3	Ringback Tone	cg	rt	distRing (optional)	Distinctive Ringing as NULL terminated string
4	Call Waiting (callee)	cg	cw	--	--
5	Call Waiting (caller)	cg	cr	--	--
6	Busy Tone	cg	bt	--	--
7	Warning Tone	cg	wt	--	--
8	Digit 1 Tone	dg	d1	--	--
9	Digit 2 Tone	dg	d2	--	--
10	Digit 3 Tone	dg	d3	--	--
11	Digit 4 Tone	dg	d4	--	--
12	Digit 5 Tone	dg	d5	--	--
13	Digit 6 Tone	dg	d6	--	--
14	Digit 7 Tone	dg	d7	--	--
15	Digit 8 Tone	dg	d8	--	--
16	Digit 9 Tone	dg	d9	--	--

Table 3-1 List of Signals

17	Digit 0 Tone	dg	d0	--	--
18	Digit * Tone	dg	ds	--	--
19	Digit # Tone	dg	do	--	--
20	Hold indicator	ind	is	Indid	il
				State	on – set indicator on off – set indicator off
21	Mute Indicator	ind	is	Indid	mu
				State	on – set indicator on off – set indicator off
22	Audio Indicator	ind	is	Indid	hf – Handsfree (Speaker) ht – Headset hs - Handset
				State	on – set indicator on off – set indicator off
23	Line Indicator	ind	is	Indid	l001– set Line 1 indicator l002– set Line 2 indicator
				State	on – set indicator on off – set indicator off blink – set indicator blink
24	Set text display	dis	di	r	Number of raw in text screen
				c	Number of column in text screen
				str	Text to display as NULL terminated string
25	Clear text display	dis	cld	--	--
26	Caller ID	rvec	callerid	name	Caller name as NULL terminated string

Step 2: Playing Signals

Table 3-1 List of Signals

				number (not in use)	Caller number as NULL terminated string
				Address	Caller address as NULL terminated string
				id	Caller ID to display as NULL terminated string
27	Line active	rvcc	la	LineId	L001 – Line 1 L002 – Line 2
28	Line inactive	rvcc	li	LineId	L001 – Line 1 L002 – Line 2

LIST OF SIGNALS FOR ANALOG TERMINATION

[Table 3-2](#) presents the list of signals and their parameters supported by the Multimedia Terminal Framework for Analog lines.

Note For signals not included in this list, see the list of signals in [Table 3-1](#).

Table 3-2 List of Signals for Analog Lines

No	Signal	Package	Id	Parameter	Value
1	Dial Tone	cg	dt	--	--
2	Ringing Tone	cg	ri	--	--
3	Busy Tone	cg	bt	--	--
4	Warning Tone	cg	wt	--	--
5	Ringback Tone	cg	rt	--	--

3.5 STEP 3: REPORTING EVENTS

REPORTING EVENTS API

Events are used by the user application to notify the Multimedia Terminal Framework about actions that have occurred on the Terminal Framework. This section describes how a user application reports hardware events and lists the events that the Multimedia Terminal Framework supports. It also explains the digit-collection mechanism for outgoing calls.

The application reports events to the Multimedia Terminal Framework by calling an MDM function (for example, when the user has pressed a DTMF key). The user application must report every event that occurs on a termination by calling the following function:

```
void rvMdmTermProcessEvent(
    RvMdmTerm*           term,
    const char*          pkg,
    const char*          id,
    RvMdmMediaStream*    media,
    RvMdmParameterList*  args)
```

Usually the events are generated on the physical termination and map to some event of the packages or events supported by this termination class.

The user application must set event parameters when sending the event, such as the package ID, the event ID, and in some cases the event parameters. The MDM then relays the event to the Multimedia Terminal Framework Call Control for processing according to the state of the call.

The function `rvMdmTermProcessEvent()` receives the following parameters:

- **Term**—the pointer to the termination on which the event happened. For example, the user pressed a digit in the UI termination.
- **pkg**—the package name; one of the packages described above.
- **id**—the event ID; an event belonging to the package described above.
- **media**—this parameter is not used.
- **args**—the event parameters, or NULL if none exist. For example, the event kd (key down) of the package kp has the parameter keyid.

Example

This example shows how to send Off Hook event:

```
RvMdmParameterList paramsList;
```

Step 3: Reporting Events

```
RvMdmPackageItem item;

/* Constructors */
rvMdmParameterListConstruct (&paramsList);
rvMdmPackageItemConstruct (&item, "", "keyid");

/* Add parameter to list */
rvMdmParameterListOr (list, &item, "kh");
/* Send Event to MTF */
rvMdmTermProcessEvent ( term, "kf", "ku", NULL, &paramsList );

/* Destructors */
rvMdmPackageItemDestruct (&pkgItem);
rvMdmParameterListDestruct (&paramsList);
```

LIST OF EVENTS

Table 3-3 presents a list of events and their parameters supported by the Multimedia Terminal Framework.

Table 3-3 *List of Events*

No	Event	Package	Id	Parameter	Value
1	On Hook	kf	ku	keyid	kh
2	Off Hook	kf	kd	keyid	kh
3	Hold	kf	ku	keyid	kl
4	Conference	kf	ku	keyid	kc
5	Attended Transfer	kf	ku	keyid	kt
6	Blind Transfer	kf	ku	keyid	kbt
7	Line 1	kf	ku	keyid	l001
8	Line 2	kf	ku	keyid	l002
9	Mute	kf	ku	mu	--
10	Redial	kf	ku	keyid	redial
11	Digit 1 down	kp	kd	keyid	k1
12	Digit 2 down	kp	kd	keyid	k2

Table 3-3 List of Events

13	Digit 3 down	kp	kd	keyid	k3
14	Digit 4 down	kp	kd	keyid	k4
15	Digit 5 down	kp	kd	keyid	k5
16	Digit 6 down	kp	kd	keyid	k6
17	Digit 7 down	kp	kd	keyid	k7
18	Digit 8 down	kp	kd	keyid	k8
19	Digit 9 down	kp	kd	keyid	k9
20	Digit 0 down	kp	kd	keyid	k0
21	Digit * down	kp	kd	keyid	ks
22	Digit # down	kp	kd	keyid	ko
23	Digit 1 up	kp	ku	keyid	k1
				duration	Duration in milliseconds
24	Digit 2 up	kp	ku	keyid	k2
				duration	Duration in milliseconds
25	Digit 3 up	kp	ku	keyid	k3
				duration	Duration in milliseconds
26	Digit 4 up	kp	ku	keyid	k4
				duration	Duration in milliseconds
27	Digit 5 up	kp	ku	keyid	k5
				duration	Duration in milliseconds
28	Digit 6 up	kp	ku	keyid	k6
				duration	Duration in milliseconds

Step 3: Reporting Events

Table 3-3 List of Events

29	Digit 7 up	kp	ku	keyid	k7
				duration	Duration in milliseconds
30	Digit 8 up	kp	ku	keyid	k8
				duration	Duration in milliseconds
31	Digit 9 up	kp	ku	keyid	k9
				duration	Duration in milliseconds
32	Digit 0 up	kp	ku	keyid	k0
				duration	Duration in milliseconds
33	Digit * up	kp	ku	keyid	ks
				duration	Duration in milliseconds
34	Digit # up	kp	ku	keyid	ko
				duration	Duration in milliseconds
35	Completion Event	kp	ce	Ds	The collected digits as NULL terminated string
				Meth	"UM" – Unambiguous match "PM" – Partial match, completion by timer expiry or unmatched event. "FM" – Full match, completion by timer expiry or unmatched event.
36	Call Forward unconditional	kf	ku	activate	on – to activate CFW off – to deactivate CFW
				keyid	cfcwu
37	Call Forward Busy	kf	ku	activate	on – to activate CFW off – to deactivate CFW
				keyid	cfcfbw

Table 3-3 List of Events

38	Call Forward No Reply	kf	ku	activate	on – to activate CFW off – to deactivate CFW
				keyid	cfnr
39	GA Active	rvcc	ga	--	--
40	Reject Call	rvcc	reject	keyid	l001 – to reject Line 1 l002 – to reject Line 2
41	Handsfree	kf	ku	keyid	hf
42	Headset	kf	ku	keyid	ht

LIST OF EVENTS FOR ANALOG TERMINATION

[Table 3-4](#) presents the list of events and their parameters supported by the Multimedia Terminal Framework for Analog lines.

Note For events that are not included in this list, see [Table 3-3](#).

Table 3-4 List of Events for Analog Lines

No	Event	Package	Id	Parameter	Value
1	On Hook	al	on	--	--
2	Off Hook	al	of	--	--
3	Digit 0 down	dd	d0	--	--
4	Digit 1 down	dd	d1	--	--
5	Digit 2 down	dd	d2	--	--
6	Digit 3 down	dd	d3	--	--
7	Digit 4 down	dd	d4	--	--
8	Digit 5 down	dd	d5	--	--
9	Digit 6 down	dd	d6	--	--

Step 3: Reporting Events

Table 3-4 *List of Events for Analog Lines*

10	Digit 7 down	dd	d7	--	--
11	Digit 8 down	dd	d8	--	--
12	Digit 9 down	dd	d9	--	--
13	Digit * down	dd	ds	--	--
14	Digit # down	dd	do	--	--
15	Digit 0 up	dd-end	d0	--	--
16	Digit 1 up	dd-end	d1	--	--
17	Digit 2 up	dd-end	d2	--	--
18	Digit 3 up	dd-end	d3	--	--
19	Digit 4 up	dd-end	d4	--	--
20	Digit 5 up	dd-end	d5	--	--
21	Digit 6 up	dd-end	d6	--	--
22	Digit 7 up	dd-end	d7	--	--
23	Digit 8 up	dd-end	d8	--	--
24	Digit 9 up	dd-end	d9	--	--
25	Digit * up	dd-end	ds	--	--
26	Digit # up	dd-end	do	--	--

COLLECTING DIGITS

The number of digits to be dialed when making an outgoing call is highly variable, as it depends on the configuration, dial plans, etc. The Multimedia Terminal Framework provides a flexible mechanism through which it manages the collection of digits, matching the dial string to a pattern, starting and stopping the collection of digits, storing the digits, setting timers between pressed digits, etc. For every digit the user presses, a Digit event should be sent indicating which key was pressed.

While processing the digit event, the Multimedia Terminal Framework should determine whether the digits dialed so far match a valid pattern, i.e., whether they match a possible phone number. You can either use your own mechanism to match the digits, or use the default mechanism of the Multimedia Terminal Framework. Both methods are described in the sections that follow.

USING YOUR OWN MECHANISM

If you choose to use your own mechanism to determine whether the dialed digits match a valid pattern, you should implement the following callback:

```
RvMdmDigitMapMatchType RvMdmTermMatchDialStringCB(
    RvMdmTerm*      term,
    const char      *dialString,
    unsigned int     *timerDuration);
```

This function is called whenever the user application sends a new digit to the Multimedia Terminal Framework. Depending on the return value of this callback, the Multimedia Terminal Framework decides whether or not to continue collecting digits.

The callback reports the following parameters:

- **term**—a pointer to the termination
- **dialString**—collected digits
- **timerDuration**—an output parameter indicating the period of time to wait for the next digit

Return Value

The user application returns a matching status, which indicates whether the digits dialed by the user match a valid pattern or not.

The possible values are:

Status	Description	Action
RV_MDMDDIGITMAP_NOMATCH	The dial string does not match any legal pattern.	<ol style="list-style-type: none">1. The Terminal Framework stops collecting digits.2. The Terminal Framework starts playing a warning tone.
RV_MDMDDIGITMAP_PARTIALMATCH	The dial string may match a legal pattern.	The Terminal Framework will continue collecting digits.
RV_MDMDDIGITMAP_UNAMBIGUOUSMATCH	The dial string matches a legal pattern.	<ol style="list-style-type: none">1. The Terminal Framework stops collecting digits.2. The Terminal Framework tries to map this dial string to a destination address (see the section Matching Digits to an Address).

Callback Implementation

The implementation of this callback will use your application mechanism to determine the matching status, i.e., whether the dial string matches a digitmap pattern or not. This mechanism may be a database, configuration, algorithm, etc.

If your mechanism is:

- **Synchronous (blocking)**

Return a matching status as described above.

- **Asynchronous (non-blocking)**

Perform the following:

- Always return RV_MDMDDIGITMAP_PARTIALMATCH.
- Once the final status has been determined (match, no match, or timeout), send a completion even to the Multimedia Terminal Framework. The completion event indicates to the Multimedia Terminal Framework to stop collecting digits. This is done as follows:

- ◆ Build the event by calling the following function:

```
void rvMdmKpDigitMapBuildEvComplete(  
    RvMdmParameterList*      parameters,  
    const char*                digitString,  
    RvMdmDigitMapMatchType  matchType,  
    void*                      userData);
```

with the following parameters:

Parameters—this parameter should be constructed without setting values.

DigitString—the collected digits.

MatchType—the matching status.

UserData—pointer to allocator, will be used to allocate internal members of the event.

- ◆ Send the Completion event by calling the following function:

```
void rvMdmTermProcessEvent(  
    RvMdmTerm*                  term,  
    const char*                  pkg,  
    const char*                  id,  
    RvMdmMediaStream*           media,  
    RvMdmParameterList*          args);
```

with the following parameters:

term—a pointer to the termination.

pkg—use a different package according to termination type. For a UI termination: package "kp". For an Analog termination: package "dd".

id—set the value ce.

media—send value NULL.

args—this parameter should be constructed without setting values.

Example

```
RvMdmParameterList params;  
  
/*Construct internal object*/  
rvMdmParameterListConstruct(&params);  
/*Build the Completion event*/  
rvMdmKpDigitMapBuildEvComplete(&params, "1234",  
    RV_MDM_DIGITMAP_UNAMBIGUOUSMATCH, &rvDefaultAlloc);  
/*Send the event to Multimedia Terminal Framework*/  
rvMdmTermProcessEvent( mdmTerm, "kp", "ce",  
    NULL, &params);  
/*Destruct internal object*/  
rvMdmParameterListDestruct(&params);
```

The diagrams below demonstrate the events flow when implementing RvMdmTermMatchDialStringCB() in both synchronous and asynchronous ways:

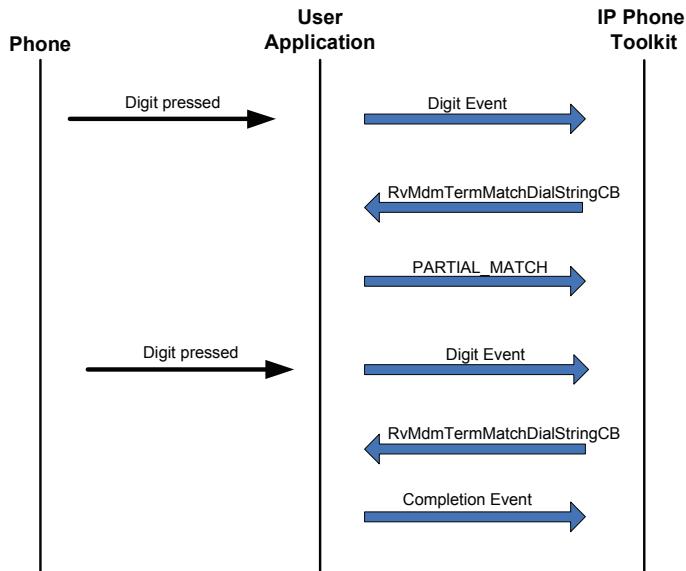


Figure 3-1 *Digitmap Events Flow (Asynchronous)*

Step 3: Reporting Events

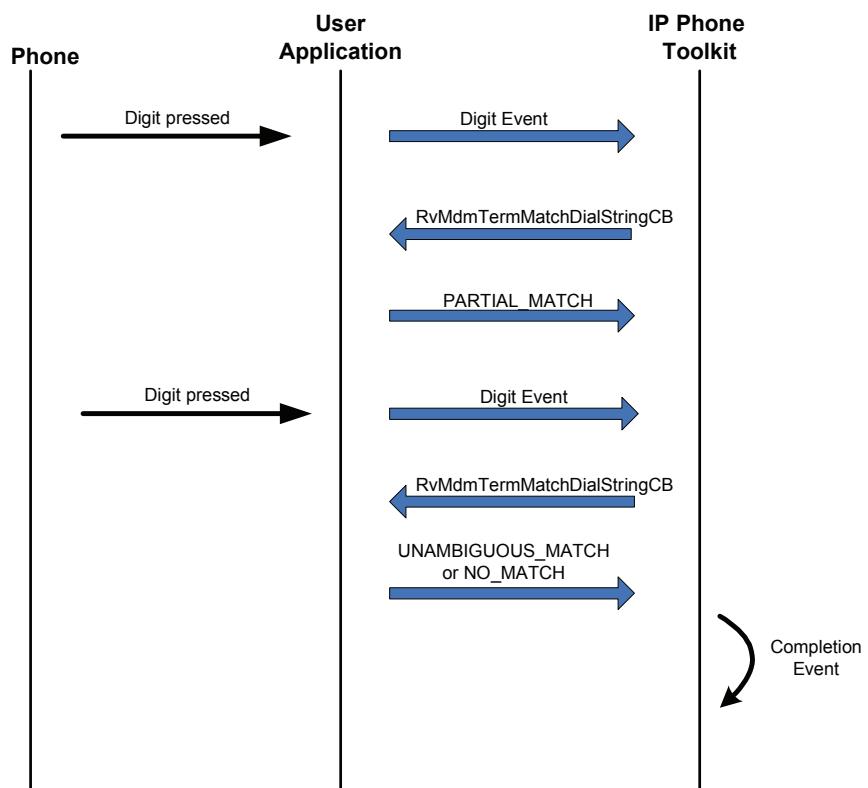


Figure 3-2 *Digitmap Events Flow (Synchronous)*

Once you have implemented this callback, you need to register it to the Multimedia Terminal Framework by calling the following API:

```
void rvMdmTermClassRegisterMatchDialStringCB(RvMdmTermClass*  
    c, RvMdmTermMatchDialStringCB matchDialStringF)
```

The function should be called when initializing the termination class to which the termination belongs.

USING THE MULTIMEDIA TERMINAL FRAMEWORK DEFAULT MECHANISM

If the Terminal Framework handles the matching of the dial string, its default implementation will determine the matching status by matching the dial string to a pre-defined digitmap. A digitmap is a pattern which should match the dial string. The pattern defines the string length, the digits it is allowed to contain, prefixes, etc. The Multimedia Terminal Framework has a default digitmap pattern; the user application can choose whether it will use this default digitmap or its own.

If the default mechanism is used, the user application should ***not*** register the RvMdmTermMatchDialStringCB() callback.

USING A DEFAULT DIGITMAP PATTERN

The default digitmap pattern defines a valid digitmap as a four-digit dial string consisting of any digit from 0-9. If the user application uses the Terminal Framework's default digitmap, it should ***not*** configure a digitmap when registering a termination before calling
rvMdmTermMgrRegisterPhysicalTermination() or
rvMdmTermMgrRegisterPhysicalTermination Async()—the Multimedia Terminal Framework will use its default predefined digitmap.

USING YOUR OWN DIGITMAP PATTERN

If you do not want to use the Terminal Framework's default digitmap, you can define your own digitmap pattern for each termination. If digitmap is configured, the Multimedia Terminal Framework will use your configured digitmap instead of the default one to determine the matching status of a dial string. For more information about configuring your own digitmap, see the section [Registering Terminations](#).

Step 3: Reporting Events

The diagram below demonstrates the events flow when the Multimedia Terminal Framework default mechanism for matching a dial string is used.

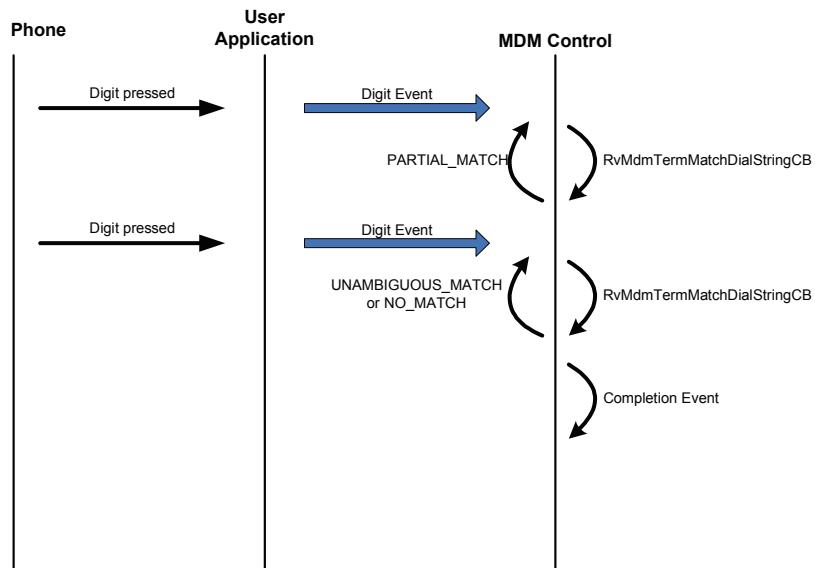


Figure 3-3 User Digitmap Events Flow

MATCHING DIGITS TO AN ADDRESS

Once all digits have been collected from the user (using any of the mechanisms described in the previous section), the digits needs to be mapped to a destination address. The user application should do the mapping by implementing the following callback:

```
RvBool RvMdmTermMapDialStringToAddressCB(  
    RvMdmTerm*      term,  
    const char*     dialString,  
    char*          address);
```

If mapping is not found (i.e., the return value is false), the Multimedia Terminal Framework sets the destination address to the registrar address, if one has been configured.

Parameters

The callback has the following parameters:

term—A pointer to the termination.

dialString—A string with all collected digits as dialed by the user.

address—Output parameter. Includes the Destination address.

Return Value

True if the destination address was found. False if it was not.

Callback Implementation

The implementation retrieves the Destination address and sets it in the output parameter "address". The Destination address can be retrieved from an internal database, configuration, mapping algorithm, LDAP, etc. The Destination address should have the formats described below.

SIP address formats:

```
<scheme>:<username>@<ip address / domain name>:<port>
```

Scheme, user name and port are optional:

- If the port number is not specified, the default port is 5060.
- If the scheme is not specified, the default scheme is "sip":

Examples:

```
User-057@172.20.69.57
User-057@172.20.69.57:5080
sip:User-057@172.20.69.57
sips:User-057@172.20.69.57
```

Callback Registration

This callback is registered during initialization (see the section [Initialization Steps](#)) by calling the following function:

```
void rvMdmTermClassRegisterMapDialStringToAddressCB (
    RvMdmTermClass*                      c,
    RvMdmTermMapDialStringToAddressCB     mapDialStringToAddressF);
```

3.6 STEP 4: INTEGRATING MEDIA

A complete Terminal Framework solution requires the integration of the Multimedia Terminal Framework with the following:

- DSP Services for Audio (voice compression/decompression, jitter buffer, echo cancellation, mixing, etc.)
- DSP Services for Video (video compression/decompression, lip synch, fast update, etc.)
- Telephony Services (camera, screen, handset, etc.) provided by the developer or by a third party
- Implementation of media transport using either the RADVISION RTP/RTCP Toolkit or any other solution

Information required for the integration of the Multimedia Terminal Framework with DSP Services and Telephony Services is provided in the [Overview](#) chapter.

To allow media originating in your phone to reach the remote phone, the media must flow from the physical device where it originates (for example: the Handset) to the RTP session. From the RTP session, it will reach the remote phone through an RTP packet stream. The packet stream parameters (IP address, port, payload, etc.) should be controlled to match the requests of the remote side.

The Multimedia Terminal Framework controls the physical devices and RTP streams through a list of callback functions that are listed in this chapter. In these callback implementations, the user application uses DSP and Telephony Services to handle RTP/RTCP, initialize physical devices, etc., according to the received parameters.

Note This section is not relevant when the Multimedia Terminal Framework Media Add-on is integrated. For more information, see the chapter [Media Add-on](#).

MEDIA CALLBACKS

During any call scenario, the user application receives callback events from the Multimedia Terminal Framework to manipulate the media streams according to the state of the call (for example: Create a media stream, connect media streams, etc.). The user application is supplied with media parameters and is required to inform the Multimedia Terminal Framework of the choices it has made and of the results of the operations. This is done by implementing a set of callbacks, as described below.

Note For more information about SDP APIs required for media callback implementation, see the RADVISION SDP Stack documentation.

1. **RvMdmTermMgrSelectTerminationCB()**

Before a new media stream is opened, the Multimedia Terminal Framework instructs the user application to create a new RTP termination by calling the following callback:

```
typedef struct RvMdmTerm_* (*RvMdmTermMgrSelectTerminationCB) (
    RvMdmTermMgr*      mgr,
    RvMdmTerm*         tempTerm)
```

Parameters

mgr—A pointer to the Termination Manager.

tempTerm—A pointer to a temporary termination created by the Multimedia Terminal Framework. The purpose of this termination is only to indicate the termination type RV_MDMTERMTYPE_EPHEMERAL.

Return Value

The user application should return a pointer to the RTP termination if the termination was created successfully, or NULL if it was not.

Callback Implementation

The implementation of this callback is as follows:

1. Register a new RTP termination by calling:

```
RvMdmTerm* rvMdmTermMgrRegisterEphemeralTermination(
    RvMdmTermMgr*           mgr,
    RvMdmTermClass*         c,
    const char*             id,
    RvMdmTermDefaultProperties_* termProperties)
```

This function receives the following parameters:

mgr—A pointer to the Termination Manager.

c—A pointer to the RTP class that was created during initialization.

id—A unique name for the termination. The name should have the format "rtp/x", where x can be anything.

termProperties—This parameter is currently not used and should be set to NULL.

Return Value

The function returns a pointer to RvMdmTerm, which the user application will use as the return value of the callback.

2. Set user data with the new termination by calling:

```
void rvMdmTermSetUserData(
    RvMdmTerm*           term,
    void *               data)
```

This function receives the following parameters:

term—A pointer to the termination, as was received when rvMdmTermMgrRegisterEphemeralTermination() was called.

data—A pointer to the user data. This data will be received in subsequent calls to all other media callbacks related to this call.

When the callback is called

This callback will be the first media callback to be called in every call scenario. It will be called once for every call.

Callback Registration

The user application should register the callback during initialization (see [Step 1: Initialization](#)) by calling the function:

```
void rvMdmTermMgrRegisterSelectTermCB (
    RvMdmTermMgr*           mgr,
    RvMdmTermMgrSelectTerminationCB selectF);
```

with the following parameters:

mgr—A pointer to the Termination Manager.

selectF—A pointer to the callback implementation. The implementation must be based on the steps described above.

2. RvMdmTermCreateMediaCB()

The Multimedia Terminal Framework calls the following callback to create one or more new media streams:

```
typedef RvBool (*RvMdmTermCreateMediaCB) (
    RvMdmTerm*                      term,
    RvMdmMediaStream*                media,
    RvMdmMediaStreamDescr*           streamDescr,
    RvMdmError*                     mdmError)
```

The callback is registered during initialization (see [Step 1: Initialization](#)) by calling `rvMdmTermClassRegisterCreateMediaCB()` twice, for two different implementations:

1. One implementation will be registered with the following parameters:
c—A pointer to the RTP class that was created during initialization.
createMediaF—Callback implementation for the RTP termination (see below).
2. The second implementation will be registered with the following parameters:
c—A pointer to the Audio Transducer (AT) class or Analog class that was created during initialization.
createMediaF—Callback implementation for the physical device (see below).

Callback implementation for RTP termination

This section describes the implementation of this callback when it is registered with the RTP class.

Parameters

The callback has the following parameters:

term—A pointer to the RTP termination, as received by the user application when `RvMdmTermMgrSelectTerminationCB()` was called.

media—This parameter is not used.

streamDescr—This is an input/output parameter:

- **Input**—This structure includes all media capabilities of the local party and may include capabilities of the remote party. The user application will use this data to choose the preferred codecs and parameters with which to open the media. For more information, see the section Callback Implementation below.
- **Output**—After choosing the codecs and media parameters with which the media is opened, the user application should clear the structure and add only the chosen parameters. For more information, see the section Callback Implementation below.
- **mdmError**—This parameter is not used.

Return Value

The user application should return True if a media stream was created successfully, and False if it was not.

Callback Implementation

Step 1: Choose common capabilities

According to the Offer/Answer standard, in incoming calls the common capabilities should be a subset of all media capabilities offered in incoming signaling messages.

When the callback is called, the "streamDescr" parameter includes two sets of capabilities:

- **Local capabilities**—All media capabilities that were loaded by the user during initialization. Retrieve these capabilities by calling the functions:

```
RvSdpMsgList *sdpListLocal =  
    rvMdmMediaStreamDescrGetLocalDescr(streamDescr);  
RvSdpMsg* msgLocal = rvSdpMsgListGetElement(  
    sdpListLocal, 0);
```

- **Remote capabilities**—Media capabilities of the remote party, as received in the incoming signaling message. Retrieve these capabilities by calling the functions:

```
RvSdpMsgList *sdpListRemote =
    rvMdmMediaStreamDescrGetRemoteDescr(streamDescr);
RvSdpMsg* msgRemote =
    rvSdpMsgListGetElement(sdpListRemote, 0);
```

Remote capabilities are empty ("sdpListRemote" will be NULL) if the callback is called in the establishment of an outgoing call.

The user application should go through the local and remote lists and choose one or more codecs and parameters that are supported by both parties and that the user application is capable of sending and receiving. The user application must be able to send and receive all chosen codecs simultaneously for the duration of the call.

- If no common capabilities are found, return "False" and ignore all of the steps described below.
- If common capabilities are found, proceed with the steps described below.

Step 2: Open an RTP session

Create a media stream with the common capabilities chosen in the previous step. When creating the media stream, the following parameters must be considered:

- **Media type**—For each media type (audio and video), review the remote list and do the following:
 - If an audio type exists, open an audio stream.
 - If a video type exists, open a video stream.

Get the media type by calling the function *rvSdpMediaDescrGetMediaType()*.

For example:

```
for (i=0; i < n; ++i)
{
    descr = rvSdpMsgGetMediaDescr(msgLocal, i);
    if( rvSdpMediaDescrGetMediaType( descr ) ==
        RV_SDPMEDIATYPE_AUDIO)
    {
        /* Open Audio stream */
    }
}
```

```
if( rvSdpMediaDescrGetMediaType( descr ) ==  
RV_SDPMEDIATYPE_VIDEO)  
{  
    /* Open Video stream */  
}  
}
```

- **Codec**—The media stream should be able to send/receive packets of the chosen codec. Depending on its capabilities, the user application can either create one session for sending and another for receiving, or one stream for both. Get the codec by calling:

rvSdpMediaDescrGetPayloadNumber()

Each codec may have additional parameters (name and rate) that can be retrieved by calling *rvSdpMediaDescrGetRtpMap()*.

- **Remote address and port**—Set the destination address of the media stream to match the remote address and port.
- **Other parameters**—Additional parameters of the media stream may be included in the SDP message, such as silence suppression, ptime, etc. These parameters can be retrieved by calling *rvSdpMediaDescrGetAttribute()*.

Step 3: Build a response

The user application should build a response with the common media capabilities that were chosen. This is done by modifying the local SDP message of the "streamDescr" parameter to include only the common capabilities:

- Clear the local SDP message and add the chosen common capabilities to it. This can be done in one of the following ways:
 - Clear all media descriptors in the SDP message and add only the chosen ones.
 - Clear only those media descriptors that were not chosen by the application.
- Go through all connections of the local SDP message and set the local IP address and port on which the user application is listening for RTP packets, by calling *rvSdpConnectionSetAddress()* and *rvSdpMediaDescrSetPort()*.

See the sample code in Step 1 (above) for an example of how to retrieve a local SDP message. The local SDP message will be included in the outgoing reply of the remote party.

When the callback is called

This callback is called once in every call scenario:

- **Incoming call**—Called after the local user has answered the call.
- **Outgoing call**—Called after the local user has completed dialing and before the signalling message is sent out.

Callback Registration

The user application should register the callback during initialization (see explanations at the beginning of the callback description).

Callback Implementation for Physical Device

This section describes the implementation of this callback, which should be registered with an AT or an Analog class.

Parameters

The callback has the following parameters:

term—A pointer to the physical termination (Audio or Video), as was received when it was registered by the user application during initialization (by calling *rvMdmTermMgrRegisterPhysicalTermination()*).

media—This parameter is not used.

streamDescr—This is an input parameter only. This structure will include all media capabilities of the local party. The user application will use this data to choose the preferred parameters with which to initialize the device.

mdmError—This parameter is currently not used.

Return Value

The user application should return True if the device was initialized successfully, and False if not.

Callback Implementation

The implementation of this callback is as follows:

1. **Device Initialization**—The user application may perform the necessary initializations for the Audio/Video devices. It is recommended to consider media parameters as received in the

local and remote descriptor and set the device accordingly (for example, silence suppression).

2. **Set stream mode**—Check stream mode by calling the function:

rvMdmMediaStreamDescrGetMode(), and set the mode of the device accordingly. For example, the audio device should not send voice if the mode is **RV_MDMSTREAMMODE_RECVONLY**.

When the callback is called

This callback will be called once during runtime for each Audio and Video termination (in the first call only) to setup the Audio/Video device.

Callback Registration

The user application should register this callback during initialization (see explanations at the beginning of the callback description).

3. RvMdmTermModifyMediaCB()

After creating a media stream, the Multimedia Terminal Framework may need to modify the media parameters of the stream by calling a callback of the following type:

```
typedef RvBool (*RvMdmTermModifyMediaCB) (
    RvMdmTerm*           term,
    RvMdmMediaStream*     media,
    RvMdmMediaStreamDescr* streamDescr,
    RvMdmError*          mediaError)
```

The callback is registered during initialization (for more information, see [Step 1: Initialization](#)) by calling *rvMdmTermClassRegisterCreateMediaCB()* twice for two different implementations:

1. One implementation will be registered with the following parameters:
 - c**—A pointer to the RTP class that was created during initialization.
 - modifyMediaF**—Callback implementation for the RTP termination (see explanations below).

2. The second implementation will be registered with the following parameters:
c—A pointer to the AudioTransducer(AT) class or Analog class that was created during initialization.
modifyMediaF—Callback implementation for physical device (see explanations below).

[Callback implementation for RTP termination](#)

This section describes the implementation of this callback when it is registered with the RTP class.

Parameters

This callback has the following parameters:

term—A pointer to the RTP termination as received by the user application when RvMdmTermMgrSelectTerminationCB() was called.

media—This parameter is not used.

streamDescr—This is an input/output parameter:

- **Input**—This structure will include the parameters of the local media and the capabilities of the remote party. The user application will use this data to choose the preferred codecs and parameters with which the media stream will be modified.
- **Output**—After choosing the common codecs and media parameters, the user application should clear the structure and add only the chosen parameters.

mediaError — This parameter is not used.

Return Value

User application should return True if the media stream was modified successfully, and False if it was not.

[Callback Implementation](#)

Step 1: Choose common capabilities

According to the Offer/Answer standard, in incoming calls the common capabilities should be a subset of all media capabilities offered in incoming signaling messages.

The "streamDescr" parameter includes two sets of media capabilities:

- **Local capabilities**—Parameters of the local media stream that was opened by the local party during the callback RvMdmTermCreateMediaCB(). Retrieve these capabilities by calling the functions:

```
RvSdpMsgList *sdpListLocal =  
    rvMdmMediaStreamDescrGetLocalDescr(streamDescr);  
RvSdpMsg* msgLocal = rvSdpMsgListGetElement(  
    sdpListLocal, 0);
```

- **Remote capabilities**—Media capabilities of the remote party, as received in the incoming signaling message. Retrieve these capabilities by calling the functions:

```
RvSdpMsgList *sdpListRemote =  
    rvMdmMediaStreamDescrGetLocalDescr(streamDescr);  
RvSdpMsg* msgRemote =  
    rvSdpMsgListGetElement(sdpListRemote, 0);
```

The user application should go through the local and remote capabilities and choose one or more codecs and parameters that are supported by both parties and that the user application is capable of sending and receiving. According to RFC 3264 (Offer/Answer), the user application must be able to send and receive all chosen codecs simultaneously for the duration of the call.

If no common capabilities are found:

Return False and ignore the steps described below. When the callback returns False, a signaling message reply will be sent out with code 488: Not Acceptable Here, and the following will happen:

- If the callback was called during call establishment, the call will not be connected.
- If the callback was called during a connected call as a result of an outgoing or incoming Re-Invite, the call will stay connected with the same media parameters that were set before the Re-Invite was sent/received.
- If the callback was called during a connected call as a result of a local or remote Hold or Unhold, the call will stay connected like before (Hold or Unhold will not be performed).

If common capabilities were found:

Complete the steps described below, and then return True.

Step 2: Modify the RTP session

Once common capabilities have been chosen, the existing media stream/s need to be modified according to the chosen common capabilities. When modifying the media stream/s, the following input parameters must be considered:

- **Media type**—For each media type (audio and video), go over the remote list and do the following:

- If the media type exists and a stream was already open, modify the stream according to the new parameters, or open a new stream if none existed.
 - If the media type does not exist, and a stream is open, close the stream.

Get the media type by calling *rvSdpMediaDescrGetMediaType()*.

- **Codec**—The media stream should be modified to enable sending/receiving packets of the newly chosen codec. Depending on the capabilities of the user application, the user application can either modify the existing RTP session to enable sending and/or receiving the new codec, or close the existing RTP session and open a new one with the new codec. Get the codec by calling:

rvSdpMediaDescrGetPayloadNumber().

Each codec may have additional parameters (name and rate) that can be retrieved by calling:

RvSdpMediaDescrGetRtpMap().

- **Remote address and port**—Set the destination address of the media stream to match the remote address and port.
- **Other parameters**—Additional media stream parameters may be included in the SDP message (silence suppression, ptime, etc.). These parameters can be retrieved by calling *rvSdpMediaDescrGetAttribute()*.

Step 3: Set Stream mode

The stream mode indicates the direction of internal and external connections, as illustrated in [Figure 3-4](#):

- **Internal Connection**—The internal connection indicates the stream between the physical device (for example, a Handset) and the RTP session.
- **External Connection**—The external connection indicates the RTP session between the local and the remote party.

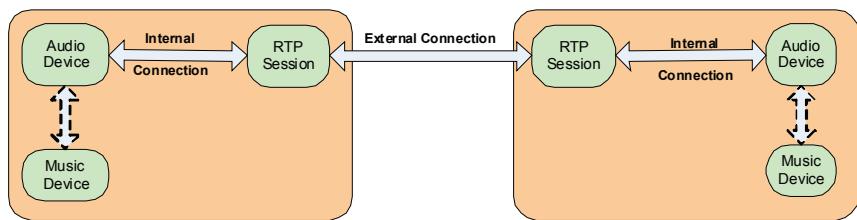


Figure 3-4 Stream Mode

Two types of stream mode exist:

- ❑ "mode" parameter—Indicates the direction of internal connection (See explanations below).
Retrieve this parameter by calling:

```
RvMdmStreamMode mode =
    rvMdmMediaStreamDescrGetMode(streamDescr);
```
- ❑ SDP "mode" attribute—Indicates the direction of the external connection. To retrieve this parameter, go through all media descriptors in the SDP message and look for the mode attribute by calling
`rvSdpMediaDescrGetConnectionMode()`.

This function will return one of the following values:

- ❖ **RV_SDPCONNECTMODE_NOTSET**—If no mode attribute was found in this media descriptor. Since according to the Offer/Answer standard (RFC 3264, section 5.1) the default is "sendrecv", you may treat this value as `RV_SDPCONNECTMODE_SENDRECV`.
- ❖ **RV_SDPCONNECTMODE_SENDONLY**—If the attribute "a=sendonly" was found in this media descriptor.
- ❖ **RV_SDPCONNECTMODE_RECVONLY**—If the attribute "a=recvonly" was found in this media descriptor.
- ❖ **RV_SDPCONNECTMODE_SENDRECV**—If the attribute "a=sendrecv" was found in this media descriptor.

- ◆ **RV_SDPCONNECTMODE_INACTIVE**—If the attribute "a=inactive" was found in this media descriptor.

Use the following code to get the SDP mode attribute from the local SDP message:

```
Int cnt = rvSdpMsgGetNumOfMediaDescr (msgLocal);
for( i= cnt-1; i >=0; i--)
{
    RvSdpMediaDescr *descr = rvSdpMsgGetMediaDescr(msgLocal, i);
    RvSdpConnectionMode mode = rvSdpMediaDescrGetConnectionMode(
        descr);

    ...
}
```

Use the following code to get the SDP mode attribute from the remote SDP message:

```
Int cnt = rvSdpMsgGetNumOfMediaDescr (msgRemote);
for( i= cnt-1; i >=0; i--)
{
    RvSdpMediaDescr *descr = rvSdpMsgGetMediaDescr(msgRemote, i);
    RvSdpConnectionMode mode = rvSdpMediaDescrGetConnectionMode(
        descr);

    ...
}
```

The user application should set the direction of the internal and external connections based on the values of the stream mode, according to the scenario. [Table 3-5](#) and the explanations below it summarize the changes that the user application should make.

Step 4: Integrating Media

Table 3-5 Changes to be made by the user application

# Scenario	Input (Stream Mode)	What should be External Connection	What should be Internal Connection	Output (Local SDP)
1 Local Hold	1. Mode = RV_MDMSTREAMMODE_SENDONLY 2. "a=sendonly" in Local SDP	SendOnly	RTP session and Audio device should be disconnected. For Music on Hold— Connect RTP to Music device.	*a=sendonly
2 Local Unhold	1. Mode = RV_MDMSTREAMMODE_SENDRECV 2. "a=sendrecv" in Local SDP	SendRecv	RTP session and Audio device should be connected. For Music on Hold— Disconnect RTP from Music device.	*a=sendrecv
3 Remote Hold	1. Mode = RV_MDMSTREAMMODE_RECVONLY 2. "a=sendonly" in Remote SDP	RecvOnly	RTP session and Audio device should be set to one-way stream.	a=recvonly
6 Remote Unhold	1. Mode = RV_MDMSTREAMMODE_SENDRECV 2. "a=sendrecv" in Remote SDP	SendRecv	RTP session and Audio device should be connected.	a=sendrecv
4 Local Hold during Remote Hold or Remote Hold during Local Hold	1. Mode = RV_MDMSTREAMMODE_INACTIVE 2. "a=sendonly" in Local SDP	Inactive	RTP session and Audio device should be disconnected.	a=inactive

Table 3-5 Changes to be made by the user application

5	Mute	1. Mode = RV_MDMSTREAMMODE_RECVONLY 2. "a= sendrecv" in local SDP	RecvOnly	RTP session and Audio device should be set to one-way stream .	**N/A
6	Unmute	1. Mode = RV_MDMSTREAMMODE_SENDRECV 2. "a= sendrecv" in local SDP	SendRecv	RTP session and Audio device should be connected.	**N/A
7	Dynamic media change	1. Mode = RV_MDMSTREAMMODE_SENDRECV 2. Local SDP will include current media parameters. 3. Remote SDP will include incoming SDP message.	According to the new parameters	According to the new parameters.	According to user application decision.

* Already updated by the Multimedia Terminal Framework. The user application does not need to update SDP.

** Irrelevant, as no signaling message is sent out in case of Mute and Unmute.

Scenarios:

- **Local Hold**—Call was put on Hold by the local user.
- **Local Unhold**—Call was Unheld by the local user.
- **Remote Hold**—Call was put on Hold by the remote party.
- **Remote Unhold**—Call was Unheld by the remote party.
- **Local Hold during Remote Hold or Remote Hold during Local Hold**—Call was put on Hold by the local user after it was put on Hold by the remote party, or Call was put on Hold by the remote party after it was put on Hold by the local user.
- **Mute**—Call was Muted by the local user.
- **Unmute**—Call was Unmuted by the local user.
- **Dynamic media change**—Local user or remote party requested to change media parameters during a connected call.

Internal Connection Direction

Table 3-5 summarizes the direction of this connection in each scenario:

- **When RTP session and Audio device should be connected—**The user application should make the required modifications so that media will flow from the RTP session to the physical device and vice versa.
- **When RTP session and Audio device should be disconnected—**The user application should make the required modifications so that *no* media will flow from the RTP session to the physical device and vice versa.
- **When RTP session and Audio device should be set to one-way stream—**The user application should make the required modifications so that media will flow from the RTP socket to the physical device only.
- **When RTP session should be connected to Music device—**The user application should make the required modifications so that media packets of music will be sent through the RTP session.
- **When RTP session should be disconnected from Music device—**The user application should make the required modifications so that *no* media packets of music will be sent through the RTP session.

External Connection Direction

Table 3-5 summarizes the direction of this connection in each scenario:

- **When this connection should be set to SendRecv—**The user application should read and write from/to the RTP socket.
- **When this connection should be set to SendOnly—**The user application should only write to the RTP socket.
- **When this connection should be set to RecvOnly—**The user application should only read from the RTP socket.
- **When this connection should be set to Inactive—**The user application should not read or write from/to the RTP socket.

Output

The user application needs to build a response (SDP message) that indicates the media that was opened by the user. This response will include the attribute: **a=sendonly** or **a=recvonly** or **a=sendrecv** or **a=inactive** according to the table. This attribute should be set in the local SDP of the parameter streamDescr as part of building the response (see Step 4).

Step 4: Build a response

The user application should build a response with the common media capabilities that were chosen. This is done by modifying the local SDP message of the "streamDescr" parameter so that it will include only the common capabilities:

- Clear the local SDP message and add the chosen common capabilities to it. This can be done in one of the following ways:
 - Clear all media descriptors in the SDP message and add only the chosen ones.
 - Clear only those media descriptors that were not chosen by the application.
- Go through all connections of the local SDP message and set the local IP address and port on which the user application is listening for RTP packets by calling *rvSdpConnectionSetAddress()* and *rvSdpMediaDescrSetPort()*.
- Set the stream mode according to [Table 3-5](#). See Step 3 and [Table 3-5](#) regarding Output.

See the sample code in Step 1 above for retrieving the local SDP message. These capabilities will be included in the outgoing reply of the remote party.

When the callback is called

This callback is called in the following scenarios:

1. When Re-Invite is received (dynamic media change)—The remote party requested to change media parameters during a connected call.
2. When Re-Invite is sent and a response is received (dynamic media change)—The local user requested to change media parameters during a connected call.
3. When the local user or the remote party puts the call on Hold.

4. When the local user or the remote party puts the call on Unhold.
5. When the local user puts the call on Mute—This changes the media internally; no signaling message is sent out.
6. When the local user puts the call on Unmute—This changes the media internally; no signaling message is sent out.

Callback Registration

The user application should register this callback during initialization (see explanations at the beginning of the callback description).

Callback Implementation for Physical Device

This section describes this callback when it is registered with the Audio Transducer (AT) class or the Analog class. This callback is currently not in use, but an empty implementation must be registered:

Callback Registration

The user application should register this callback during initialization (see explanations at the beginning of the description of the callback).

Callback Implementation

The implementation can be an empty implementation.

4. RvMdmTermMgrConnectCB

To allow media originating in the user phone to reach the remote phone, media must flow from the physical entity where it originates (for example, the Handset) to the RTP session. From the RTP session, the media will reach the remote phone through an RTP packet stream. The Multimedia Terminal Framework will notify the user application to connect the media between the terminations by calling a callback of the following type:

```
typedef RvBool (*RvMdmTermMgrConnectCB) (  
    RvMdmTermMgr*           mgr,  
    RvMdmTerm*              source,  
    RvMdmMediaStream*       m1,  
    RvMdmTerm_*             target,  
    RvMdmMediaStream*       m2,  
    RvMdmStreamDirection   direction,  
    RvMdmError*             mdmError)
```

Parameters

The parameters of this function are:

mgr—A pointer to the Termination Manager.

source—The first termination may be either:

- **Physical termination**—A pointer to the physical termination as received when it was registered by the user application during initialization (by calling *rvMdmTermMgrRegisterPhysicalTermination()*).
- **RTP termination**—A pointer to the RTP termination as received by the user application when *RvMdmTermMgrSelectTerminationCB()* was called.

m1—A pointer to the first termination media stream. This parameter can be used to retrieve user data.

target—The second termination may be either:

- **Physical termination**—A pointer to the physical termination, as received when it was registered by the user application during initialization (by calling *rvMdmTermMgrRegisterPhysicalTermination()*).
- **RTP termination**—A pointer to the RTP termination as received by the user application when *RvMdmTermMgrSelectTerminationCB()* was called.

m2—A pointer to the second termination media stream.

direction—Indicates the stream flow between the RTP session and the physical device. This parameter will always be set to *RV_MDMSTREAMDIRECTION_BOTHWAYS*, meaning that the stream flow should be bidirectional.

mdmError—This parameter is not used.

Return Value

The user application should return True if terminations were connected successfully, and False if they were not.

Callback Implementation

The user application should connect two terminations according to their type:

1. When the source is an RTP termination, and the target is an audio or video device (or vice versa)—Connect the RTP

session to the physical device (for example, a Handset as an audio device, a camera as a video device, etc.).

2. When both the source and the target are RTP terminations (in conference scenarios)—Connect two RTP sessions, i.e., mix the RTP inputs of both RTP streams and send the result to both (for conferencing purposes). The termination type can be retrieved by calling:

```
RvMdmTermType rvMdmTermGetType (RvMdmTerm* term);
```

When the callback is called

This callback is called in the following scenarios:

- Every outgoing and incoming call—Connect between an Audio termination and an RTP termination.
- Conference Calls—When setting up a Conference Call, this callback is called three times:
 - When setting up the first call—Connect between Audio and Video terminations and an RTP termination.
 - When setting up the second party—Connect between Audio and Video terminations and an RTP termination.
 - When the conference is completed—Connect the two RTP terminations of both calls. The user application should mix the RTP inputs of the two RTP streams and send the result to both so that media will flow between all three parties of the call.

Callback Registration

The user application should register the callback during initialization by calling the function:

```
void rvMdmTermMgrRegisterConnectCB (
    RvMdmTermMgr*           mgr,
    RvMdmTermMgrConnectCB   connectF)
```

Parameters

mgr—A pointer to the Termination Manager.

connectF—A pointer to the callback implementation.

5. CALLBACK RvMdmTermMgrDisconnectCB()

The Multimedia Terminal Framework will call a callback of the following type to disconnect media between two terminations:

```
typedef RvBool (*RvMdmTermMgrDisconnectCB) (
    RvMdmTermMgr*           mgr,
    RvMdmTerm*              source,
    RvMdmMediaStream*       m1,
    RvMdmTerm_*             target,
    RvMdmMediaStream*       m2,
    RvMdmError*             mdmError)
```

Parameters

The callback has the following parameters:

mgr—A pointer to the Termination Manager.

source—The first termination may be either:

- **Physical termination**—A pointer to the physical termination as received when it was registered by the user application during initialization (by calling *rvMdmTermMgrRegisterPhysicalTermination()*).
- **RTP termination**—A pointer to the RTP termination as received by the user application when *RvMdmTermMgrSelectTerminationCB()* was called.

m1—A pointer to the first termination media stream. This parameter can be used to retrieve user data.

target—The second termination may be either:

- **Physical termination**—A pointer to the physical termination as received when it was registered by the user application during initialization (by calling *rvMdmTermMgrRegisterPhysicalTermination()*).
- **RTP termination**—A pointer to the RTP termination as received by the user application when *RvMdmTermMgrSelectTerminationCB()* was called.

m2—A pointer to the second termination media stream.

mdmError—This parameter is not used.

Return Value

The user application should return True if terminations were disconnected successfully, and False if they were not.

Callback Implementation

When this callback is invoked, the user application should stop the data flow from the source termination to the target termination.

Callback Registration

The user application should register the callback during initialization by calling the function:

```
void rvMdmTermMgrRegisterDisconnectCB (
    RvMdmTermMgr*                      mgr,
    RvMdmTermMgrDisconnectCB*           disconnectF)
```

Parameters

mgr—A pointer to the Termination Manager.

disconnectF—A pointer to the callback implementation.

When the callback is called

This callback is called when the call is terminated.

6. RvMdmTermDestroyMediaCB()

To release all resources related to a media stream, the Multimedia Terminal Framework will call a callback of the following type:

```
typedef RvBool (*RvMdmTermDestroyMediaCB) (
    RvMdmTerm*                      term,
    RvMdmMediaStream*                media,
    RvMdmError*                     mdmError)
```

The callback is registered during initialization (for more information, see [Step 1: Initialization](#)) by calling *rvMdmTermClassRegisterDestroyMediaCB()* twice for two different implementations of this callback:

1. One implementation is registered with the following parameters:
 - c—A pointer to the RTP class that was created during initialization.
destroyMediaF—Callback implementation for the RTP termination (see explanations below).
2. The second implementation is registered with the following parameters:
 - c—A pointer to the Audio Transducer (AT) class or the Analog class that was created during initialization.
destroyMediaF—Callback implementation for the physical device (see explanations below).

Callback implementation for RTP termination

This section describes this callback when it is registered with the RTP class.

Parameters

The callback has the following parameters:

term—A pointer to the RTP termination as received by the user application when *RvMdmTermMgrSelectTerminationCB()* was called.

media—This parameter is not used.

mdmError—This parameter is not used.

Return Value

The user application should return True if the session was terminated successfully, and False if it was not.

Callback Implementation

The user application releases all resources allocated when the callbacks *RvMdmTermCreateMediaCB()* and *RvMdmTermModifyMediaCB()* were called.

When the callback is called

This callback is called every time a call is terminated, either by the local user going On Hook, or by the remote party ending the signaling session.

Callback Registration

The user application should register this callback during initialization (see explanations at the beginning of the description of the callback).

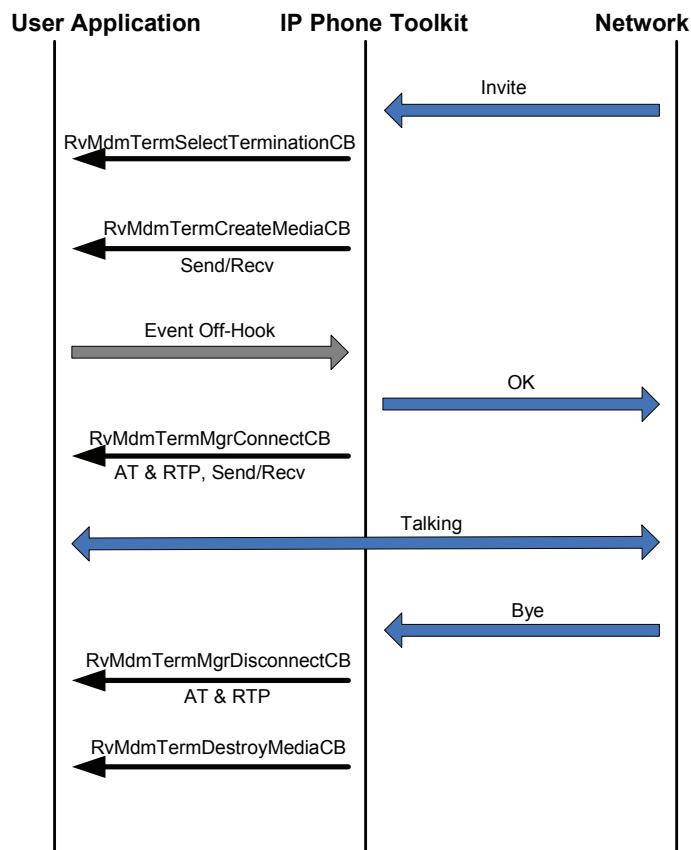
Callback implementation for physical termination

This section describes this callback when it is registered with the Audio Transducer (AT) class or the Analog class. This callback is currently not in use, but an empty implementation must be registered.

CALL FLOWS

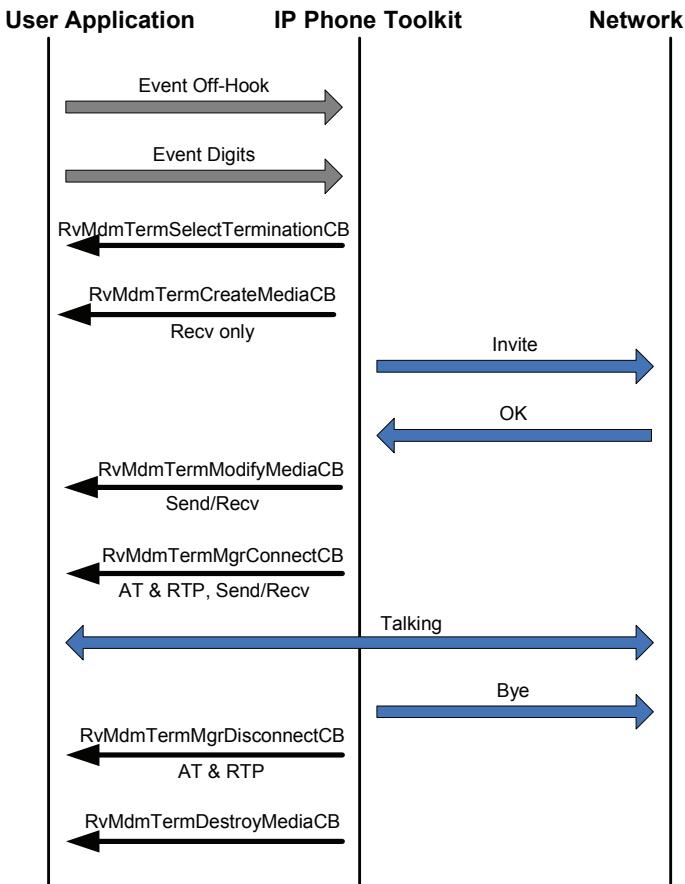
The call flows below illustrate how the Multimedia Terminal Framework handles media in basic calls.

The Multimedia Terminal Framework receives a basic incoming call.



Step 4: Integrating Media

The Multimedia Terminal Framework sends out a basic outgoing call.



4

EXTENSIBILITY

4.1 WHAT'S IN THIS CHAPTER

This chapter explains how Multimedia Terminal Framework functionality can be extended and/or modified by the user application to accommodate the specific requirements of each application:

- [Overview](#)
- [MDM Control](#)
- [SIP Control](#)

4.2 OVERVIEW

The Multimedia Terminal Framework Extensibility feature comprises a set of APIs and events that allow the user application to extend and modify Multimedia Terminal Framework behavior to match its specific needs. The user application receives Extensibility events when certain hardware and Stack events occur. When such events are received, the user application can choose to handle the event itself or have the Multimedia Terminal Framework do its default processing. In certain states or events, the Extensibility events allow users to send their own signals to the user application. The Extensibility feature was based on customer needs for modifications and it is not feature-specific. The functionality of the Extension feature is described in the sections that follow.

4.3 MDM CONTROL

MDM Control Extensibility enables the user application to extend and modify functionality of the local phone. This functionality is affected by events received from the user application (usually as a result of the user pressing a key) and is common to all protocols.

When the user application registers to MDM Control Extensibility callbacks, it receives MDM events. MDM events are sent by the user application to the Multimedia Terminal Framework (usually as a result of the user pressing a key). When these events are received, the user application can do its own processing in addition to or instead of that of the Multimedia Terminal Framework state machine.

MDM Control Extensibility APIs allow the user application to do the following:

- Implement features such as speed dialing, distinctive ringing, redial, backspace, etc.
- Intervene in the Multimedia Terminal Framework's default processing of events received from the user application
- Control the text display on the LCD
- Apply user signals on the phone in certain states or events

To support this feature:

The user application should implement a set of callbacks and register them with the Multimedia Terminal Framework before initialization. You can choose to implement some or all of the callbacks. In addition, the user application can use a set of APIs to retrieve information from Multimedia Terminal Framework objects. The Extension callbacks and APIs are defined in the *rvMdmControlApi.h* file. The callbacks are described below.

The user application must take the following steps:



Step 1: Implement Extension callbacks

Implement some or all of the Extension callbacks (see [MDM Extension Callbacks](#)), and initialize the structure `RvIppMdmExtCbk`s with pointers to these implementations. You can choose to implement some or all of the callbacks. A callback that is not implemented will be set to NULL in the structure `RvIppMdmExtCbk`s.



Step 2: Register Extension callbacks

The following function registers MDM Extension callbacks:

```
void rvIppMdmRegisterExtCbk(
    RvIppMdmExtCbk* cbks)
```

This function must be called after rvIppSipSystemInit() and before any other Multimedia Terminal Framework API.

MDM EXTENSION CALLBACKS

MAPPING USER EVENTS

```
typedef RvCCTerminalEvent (*RvIppMdmMapUserEventCB) (
    const char*          pkg,
    const char*          id,
    RvMdmParameterList* args,
    char*                key);
```

Description

The purpose of this callback is to map a user-defined package and ID to an event. The package and the ID are received when the user application sends them to the Multimedia Terminal Framework by calling rvMdmTermProcessEvent().

Invoked

When the Terminal Framework receives a package and an ID from the user application but fails to map it to an Terminal Framework event (an event that was defined by the Multimedia Terminal Framework).

User application can

Map its own package and ID for user-defined events that are unknown to the Multimedia Terminal Framework. These events can later be processed by the application. The examples "Backspace" and "Redial" demonstrate how it is used.

Return value

The return value is the event that will be sent to the Multimedia Terminal Framework state machine. Can be either Multimedia Terminal Framework event RvCCTerminalEvent (defined in rvCallControlApi.h) or a user-defined event. Note that a user event must be defined as greater than RV_CCTERMEVENT_USER.

PRE-PROCESS EVENT

```
typedef RvCCTerminalEvent (*RvIppMdmPreProcessEventCB) (
    RvIppConnectionHandle    connHndl,
    RvCCTerminalEvent        eventId,
    RvCCEventCause           reason);
```

Description

Enables the user application to process an MDM event before the Terminal Framework does its default processing.

Invoked

Before the MDM event is processed by the Terminal Framework state machine.

Return value

The returned value indicates which event will be processed and whether or not the user wants the Multimedia Terminal Framework to continue processing this event. RV_CCTERMEVENT_NONE indicates that the Multimedia Terminal Framework will not process the event.

POST-PROCESS EVENT

```
typedef void (*RvIppMdmPostProcessEventCB) (
    RvIppConnectionHandle    connHndl,
    RvCCTerminalEvent        eventId,
    RvCCEventCause           reason);
```

Description

Enables the user application to process an MDM event after the Terminal Framework does its default processing.

Invoked

After the MDM event has been processed by the Terminal Framework state machine.

Return value

None.

CHANGE DISPLAY

```
typedef void (*RvIppMdmDisplayCB) (
    RvIppConnectionHandle    connHndl,
    RvIppTerminalHandle     terminalHndl,
    RvCCTerminalEvent       event,
    RvCCEventCause          cause,
    void*                   displayData);
```

Description

Enables the user application to change the screen text display. If the user application does not implement this callback, the Terminal Framework will use a default display function, rvCCConnMdmDisplay(), which the user application can also use as sample code for its callback. The callback reports sufficient parameters to enable the user application to determine the event and the state of the call and set the display accordingly. The parameter displayData is the user data registered by Register Extensibility Callbacks.

Invoked

Each time the state of the MDM connection changes.

Return value

None.

CONNECTION CREATED

```
typedef void (*RvIppMdmConnectionCreatedCB) (
    RvIppConnectionHandle connHndl);
```

Description

Notifies the user application when a connection object is created. Can be useful in allocating user data, which is set in the connection through the function `rvIppMdmConnSetUserData()`.

Invoked

After the Terminal Framework creates a new connection object. A connection object is created when an incoming call is received or an outgoing call is created.

Return value

None.

CONNECTION DESTRUCTED

```
typedef void (*RvIppMdmConnectionDestructedCB) (
    RvIppConnectionHandle connHndl);
```

Description

Notifies the user application when a connection object is destroyed. Can be useful in freeing user data that is retrieved through `rvIppMdmConnGetUserData()`.

Invoked

Before the Terminal Framework destroys a connection object. A connection object is destroyed when a call ends.

Return value

None.

MDM EXTENSION APIs

This section presents a list of APIs that the user application can use to apply signals to the phone.

REGISTER USER SIGNALS

```
void rvMdmTermMgrRegisterUserSignal (
    RvMdmTermMgr*      mgr,
    const char*        pkg,
    const char*        signal);
```

Description

Registers a user package and event that may later be sent as signals to the user application. If the event is registered already, it will be ignored. If the user application tries to send a signal with an unregistered package and ID, the signal will not be sent.

STOP SIGNALS

```
void rvIppMdmTerminalStopSignals()
```

Description

Enables the user to stop all active signals on a termination.

CODE FLOW

When the Multimedia Terminal Framework processes an event received from the user application, the functions listed below are called in the following order:

Function	Explanation
rvMdmTermProcessEvent()	Multimedia Terminal Framework API called by the user application.
rvCCTerminalMdmMapEvent()	Multimedia Terminal Framework internal mapping events function.
RvIppMdmMapUserEventCB()	Extension callback invoked by the Multimedia Terminal Framework.
RvIppMdmPreProcessEventCB()	Extension callback invoked by the Multimedia Terminal Framework.
RvIppMdmPostProcessEventCB()	Extension callback invoked by the Multimedia Terminal Framework.
RvIppMdmDisplayCB()	Extension callback invoked by the Multimedia Terminal Framework.

USER INTERFACE FILES

When implementing Extension callbacks, the user application has access to the following APIs:

Functions	Description	File
Multimedia Terminal Framework interface		rvmdm.h, rvMdmControlApi.h
MDM Control objects	APIs of the objects available in the callback (RvIppProviderHandle, RvIppTerminalHandle, RvIppConnectionHandle, RvMdmDigitMap, RvMdmMediaDescriptor, etc.). Refer to the Multimedia Terminal Framework Reference Guide for a complete list of APIs.	rvmdmobjects.h. rvMdmControlApi.h
User data	A pointer to the user application data which can be set by calling functions of the type rvXXXGetUserData() and rvXXXSetUserData() for the objects available in the callback.	rvmdm.h, rvmdmobjects.h.
User application functions		

EXAMPLES

The following examples demonstrate how the MDM Extensibility feature is used to implement small features and extend default Multimedia Terminal Framework functionality.

BACKSPACE

This example demonstrates how the Backspace feature is implemented. A special button on the Terminal Framework should be called Backspace. Any time the user presses this button, the last-dialed digit is deleted.

1. Define a new user event

The new event macro must be greater than RV_CCTERMEVENT_USER:

```
#define USER_CCTERMEVENT_BACKSPACE    RV_CCTERMEVENT_USER + 1
```

2. Map the new event

Implement a user callback to map the relevant package and event ID to the Backspace event. The package and event ID are unknown to the Multimedia Terminal Framework.

```
RvCCTerminalEvent userCBMapEvent (const char*      pkg,
                                  const char*      id,
                                  RvMdmParameterList*   args,
                                  char*            key)
{
    if ((!strcmp(pkg, "user")) && (!strcmp(id, "backspace")))
        return USER_CCTERMEVENT_BACKSPACE;
    else
        return RV_CCTERMEVENT_NONE;
}
```

3. Process the new event

Implement a user callback to process the Backspace event. You can use the Connection and Terminal APIs to fetch the stored dial string and remove the last digit from it. For this feature the callback will return RV_CCTERMEVENT_NONE to indicate to the Multimedia Terminal Framework that it should not keep processing this event.

```
RvCCTerminalEvent userCBPreProcessEvent (
    RvIppConnectionHandle    connHndl,
    RvCCTerminalEvent        eventId,
    RvCCEventCause           reason)
{
    RvIppTerminalHandle t = rvIppMdmConnGetTerminal(connHndl);

    if (eventId == USER_CCTERMEVENT_BACKSPACE) {
        RvIppTerminalHandle t = rvIppMdmConnGetTerminal(connHndl);
        char dialString[128];

        /* Get the dial string*/
        rvIppMdmTerminalGetDialString(t, dialString, 128);
        /* Check the string is not empty*/
        .....
        /* Manipulate the string to remove the last digit*/
        .....
        /* Indicate the Multimedia Terminal Framework to not process
```

```
    this.event:*/
    return RV_CCTERMEVENT_NONE;

}

/* For all other events - return the original event. */
return eventId;
}
```

4. Register Extension callbacks

Register the two callbacks in MDM Extension APIs.

```
RvIppMdmExtClbks userMdmClbks =
{
    NULL,
    NULL,
    userCBMapEvent,
    userCBPreProcessEvent,
    NULL,
    NULL,
    NULL
};

rvIppMdmRegisterExtClbks(&userMdmClbks);
```

5. Send the new event

Send the Backspace event to the Multimedia Terminal Framework each time the user presses the corresponding key:

```
rvMdmTermProcessEvent(mdmTerm, "user", "backspace", NULL, NULL);
```

REDIAL

This example demonstrates how to implement the Redial feature. A special button on the Terminal Framework should be called Redial. Any time the user presses this button, the last-dialed phone number will be dialed again.

1. Define a new user event

The new event macro must be greater than RV_CCTERMEVENT_USER:

```
#define USER_CCTERMEVENT_REDIAL      RV_CCTERMEVENT_USER + 2
```

2. Map the new event

Implement a user callback to map the relevant package and event ID to the Redial event. The package and event ID are unknown to the Multimedia Terminal Framework.

```
RvCCTerminalEvent userCBMapEvent (const char*          pkg,
                                    const char*          id,
                                    RvMdmParameterList* args,
                                    char*                key)
{
    if ((!strcmp(pkg, "user")) && (!strcmp(id, "redial")))
        return USER_CCTERMEVENT_REDIAL;
    else
        return RV_CCTERMEVENT_NONE;
}
```

3. Process the new event

Implement a user callback to process the Redial event. This callback has two functions:

- Stores the last dial string by processing the Digits event. This event is sent every time the user presses a digit key. The dial string that contains the accumulated digits is retrieved using the Connection and Terminal APIs. This dial string should be stored, since it is reset after the dialing process is completed.
- Sends the stored dial string, by processing the Redial event. For each digit an event is built and sent separately.

For this feature the callback returns RV_CCTERMEVENT_NONE to indicate to the Multimedia Terminal Framework that it should not keep processing this event.

```
RvCCTerminalEvent userCBPreProcessEvent(
    RvIppConnectionHandle      connHndl,
    RvCCTerminalEvent         eventId,
    RvCCEventCause            reason)
{
    RvIppTerminalHandle t = rvIppMdmConnGetTerminal(connHndl);
    RvMdmTerm* mdmTerm = rvIppMdmTerminalGetMdmTerm(t);

    if (eventId == RV_CCTERMEVENT_DIGITS)
    {
        /* Get dial string*/
        rvIppMdmTerminalGetDialString(t, dialString, 128);

        /* Store dial string*/
        ....
    }

    else if (eventId == USER_CCTERMEVENT_REDIAL) {
        char* lastDialedString = getStoredDialString();
        /* For each digit...*/
        {
            /* Build an event for each digit*/
            .....
            /* Send each digit to IPP TK*/
            rvMdmTermProcessEvent(mdmTerm, "dd", "ku", NULL, params);
        }

        /*Indicate to IPP TK not to process this event:*/
        return RV_CCTERMEVENT_NONE;
    }

    /* For all other events: return the original event */
    return eventId;
}
```

4. Register Extension callbacks

Register the two callbacks in MDM Extension APIs.

```
RvIppMdmExtClbks userMdmClbks =
{
    NULL,
    NULL,
    userCBMapEvent,
    userCBPreProcessEvent,
    NULL,
    NULL,
    NULL
};

rvIppMdmRegisterExtClbks (&userMdmClbks);
```

5. Send the new event

Send the new event to Multimedia Terminal Framework every time the user presses the corresponding key:

```
rvMdmTermProcessEvent (mdmTerm, "user", "redial", NULL, NULL);
```

AUTO ANSWER

This example demonstrates how to implement the Auto Answer feature. When the feature is on, every incoming call will be automatically answered. The configuration of this feature (setting it to On and Off) is out of the Multimedia Terminal Framework scope and is assumed to be handled by the user application.

1. Process the MakeCall event

Implement a user callback to process the MakeCall event. The callback sends the Multimedia Terminal Framework an Off-Hook event when a MakeCall event, indicating an incoming call, is received. For this feature the callback returns RV_CCTERMEVENT_NONE, to indicate to the Multimedia Terminal Framework not to keep processing this event.

```
void userCBPostProcessEvent(RvIppConnectionHandle connHndl,
                           RvCCTerminalEvent eventId,
                           RvCCEventCause reason)
{
    RvMdmParameterList paramsList;
    RvMdmPackageItem pkgItem;
    RvMdmTerm* mdmTerm;
    RvIppTerminalHandle t = rvIppMdmConnGetTerminal(connHndl);

    if (eventId == RV_CCTERMEVENT_RINGING)
    {
        /*Construct local objects*/
        rvMdmParameterListConstruct(&paramsList);
        rvMdmPackageItemConstruct(&pkgItem, "", "keyid");

        /* Build an Off-Hook event */
        rvMdmParameterListOr(&paramsList, &pkgItem, "kh");

        /*Send Off-Hook event */
        mdmTerm = rvIppMdmTerminalGetMdmTerm(t);
        rvMdmTermProcessEvent(mdmTerm, "kf", "kd", NULL,
                              &paramsList);

        /*Destruct local objects*/
        rvMdmPackageItemDestruct(&pkgItem);
        rvMdmParameterListDestruct(&paramsList);
    }
}
```

}

2. Register Extension callback

Register the callback in MDM Extension APIs.

```
RvIppMdmExtClbks userMdmClbks =
{
    NULL,
    NULL,
    NULL,
    userCBPreProcessEvent,
    NULL,
    NULL,
    NULL
};

rvIppMdmRegisterExtClbks (&userMdmClbks);
```

AUTO DIALING

The example below demonstrates how the Auto Dialing feature is implemented. When user goes off-hook, a pre-configured number will be dialed. The configuration of this feature (setting it to On and Off, configuring the dial string) is out of the Multimedia Terminal Framework scope and is assumed to be handled by the user application.

1. Process the Dial Tone event

Implement user callback to process the Dial Tone event after the Multimedia Terminal Framework has processed it. After the event has been processed by the Multimedia Terminal Framework state machine, digit events can be sent as if they were pressed by the user.

```
void userCBPostProcessEvent (
    RvIppConnectionHandle connHndl,
    RvCCTerminalEvent eventId,
    RvCCEventCause reason)
{
    RvIppTerminalHandle t = rvIppMdmConnGetTerminal(connHndl);
    RvMdmTerm* mdmTerm = rvIppMdmTerminalGetMdmTerm(t);

    if (eventId == RV_CCTERMEVENT_DIALTONE)
```

```
    {
        char* dialString = getConfiguredDialString();
        /* For each digit...*/
        {
            /* Build an event for each digit*/
            .....
            /* Send each digit to IPP TK*/
            rvMdmTermProcessEvent (mdmTerm, "dd", "ku", params, NULL);
        }
    }
}
```

2. Register Extension callback

Register the callback in MDM Extension APIs.

```
RvIppMdmExtClbks userMdmClbks =
{
    NULL,
    NULL,
    NULL,
    NULL,
    userCBPostProcessEvent,
    NULL,
    NULL
};

rvIppMdmRegisterExtClbks (&userMdmClbks);
```

CHANGE DISPLAY

1. Define your display state machine

Implement the display callback to set the chosen display to the correct state.

```
void userCBDisplay(
    RvIppConnectionHandle connHndl,
    RvIppTerminalHandle termHndl,
    RvCCTerminalEvent event,
    RvCCEventCause cause,
    void* displayData)
{
```

```
RvCCConnState state = rvIppMdmConnGetState(connHndl);

switch(state) {
    case RV_CCCONNSTATE_CONNECTED:
        rvIppMdmTerminalClearDisplay(termHndl);
        rvIppMdmTerminalSetDisplay(termHndl, "Connected...", 0, 0);
        break;

    case RV_CCCONNSTATE_INITIATED:
        rvIppMdmTerminalClearDisplay(termHndl);
        break;

    case RV_CCCONNSTATE_ALERTING:
        userDisplayCallerId(connHndl, termHndl);
        break;

    default:
        break;
}
}
```

2. Register Extension callback

```
RvIppMdmExtCblk userMdmCblk =
{
    userCBDisplay,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL
};

rvIppMdmRegisterExtCblk(&userMdmCblk);
```

APPLY USER SIGNAL

1. Register the new signal

This function registers a user package and event, which you can later apply to your application.

```
rvMdmTermMgrRegisterUserSignal(termMgr, "mypkg", "mysignal");
```

2. Apply the new signal

To apply your signal, call the function rvIppMdmTerminalStartSignalUI() in one of the callbacks you have implemented for Extensibility APIs. In the following example, the signal is applied when the user disconnects a call, while a held call continues to exist.

```
void userCBConnectionDestructed(RvIppConnectionHandle connHndl)
{
    RvIppTerminalHandle t = rvIppMdmConnGetTerminal(connHndl);

    if (rvIppMdmTerminalOtherHeldConnExist(t, connHndl) == rvTrue)
        rvIppMdmTerminalStartSignalAT( term, "mypkg", "mysignal",
NULL);

}
```

"FORGOTTEN" HELD CALL

This example shows how the Terminal Framework indicates to the user that a held call exists while no other calls are active. This scenario may occur when the user has two calls on his phone: One call is active (talking), the other is on Hold. If the user disconnects the active call and does not retrieve the held call, a signal will begin to play to remind him of the "forgotten" call.

1. Implement Extension callback

```
void userCBConnectionDestructed(RvIppConnectionHandle connHndl)
{
    RvIppTerminalHandle t = rvIppMdmConnGetTerminal(connHndl);

    if (rvIppMdmTerminalOtherHeldConnExist(t, connHndl) == rvTrue)
        rvIppMdmTerminalStartRingingSignal(t);

}
```

2. Register Extension callback

```
RvIppMdmExtClbks userMdmClbks =
{
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    userCBConnectionDestructed
};

rvIppMdmRegisterExtClbks (&userMdmClbks);
```

DISABLE CALL WAITING

This example shows how the user application can disable the Call Waiting functionality. When Call Waiting is disabled, a new incoming call will be rejected automatically when another call is already connected to the terminal.

1. Process a Ringing event

Implement a user callback to process the Ringing event. The Ringing event is received in every incoming call; every time a new call arrives, the function will check whether a connected call exists in the terminal. If one exists, the function will return the RV_CCTERMEVENT_REJECTCALL event. If none exists, the function will return the same Ringing event so that processing will not be affected by this function.

```
RvCCTerminalEvent userCBPreProcessEvent (
    RvIppConnectionHandle connHndl,
    RvCCTerminalEvent eventId,
    RvCCEventCause reason)
{
    RvIppTerminalHandle t = rvIppMdmConnGetTerminal (connHndl);

    if (eventId == RV_CCTERMEVENT_RINGING) {

        RvCCTermConnState termState =
            rvIppMdmConnGetTermState (connHndl);

        if (termState == RV_CCTERMCONSTATE_BRIDGED)
```

```

{
    *reason = RV_CCCAUSE_BUSY;

    return RV_CCTERMEVENT_REJECTCALL;

}/* For all other events: Return the original event. */

return eventId;
}

```

2. Register Extension callbacks

Register the two callbacks in MDM Extension APIs.

```

RvIppMdmExtClbks userMdmClbks =
{
NULL,
NULL,
NULL,
NULL,
userCBPreProcessEvent,
NULL,
NULL,
NULL
};
rvIppMdmRegisterExtClbks (&userMdmClbks);

```

AUTOMATIC DISCONNECT

This example shows how a call that was disconnected by the remote party will also be disconnected automatically by the local user (i.e., the line will be released before the local user goes On Hook).

1. Process the Remote Disconnected event

Implement a user callback to process the Remote Disconnected event. The Remote Disconnected event is received whenever the remote party disconnects the call; this function sends an RV_CCTERMEVENT_ONHOOK event to the state machine to release the local line.

Without this function, when the remote party disconnects, the line in the local terminal is not released until the local user goes Off Hook.

With this function, the line in the local terminal is released immediately when the remote party disconnects. When the user goes On Hook afterwards, this will have no effect.

```
RvCCTerminalEvent userCBPreProcessEvent(
    RvIppConnectionHandle connHndl,
    RvCCTerminalEvent eventId,
    RvCCEventCause reason)
{
    RvIppTerminalHandle t = rvIppMdmConnGetTerminal(connHndl);

    if (eventId == RV_CCTERMEVENT_REMOTE_DISCONNECTED) {

        RvCCTermConnState termState =
            rvIppMdmConnGetTermState(connHndl);

        if (termState == RV_CCTERMCONSTATE_TALKING)
        {
            return RV_CCTERMEVENT_ONHOOK;
        }
    }
}
```

2. Register Extension callbacks

Register the two callbacks in MDM Extension APIs.

```
RvIppMdmExtClbks userMdmClbks =
{
    NULL,
    NULL,
    NULL,
    userCBPreProcessEvent,
    NULL,
    NULL,
    NULL
};
rvIppMdmRegisterExtClbks (&userMdmClbks);
```

4.4 SIP CONTROL

SIP Control Extensibility provides the user application with better control of the SIP Stack in changing or adding to the default functionality of the Multimedia Terminal Framework. It allows the user application to do the following:

- Change the Stack configuration
- Receive and process Stack events to which the Multimedia Terminal Framework does not listen
- Receive and process Stack events to which the Multimedia Terminal Framework listens:
 - Change default processing of the Multimedia Terminal Framework
 - Modify outgoing messages
 - Retrieve information from incoming messages

To support this feature:

The user application should implement a set of callbacks and register them with the Multimedia Terminal Framework before initialization. You can choose to implement some or all of the callbacks. In addition, the user application can use a set of APIs to retrieve information from Multimedia Terminal Framework objects. The Extension callbacks and APIs are defined in the *rvSipControlApi.h* file. The callbacks are described below.

The user application must take the following steps:



Step 1: Implement Extension callbacks

Implement some or all of the Extension callbacks (see [SIP Extension Callbacks](#)), and initialize the structure `RvIppSipExtClbks` with pointers to these implementations. You can choose to implement some or all of the callbacks. A callback that is not implemented will be set to NULL in the structure `RvIppSipExtClbks`.



Step 2: Register Extension callbacks

This is done by calling the following API:

```
void rvIppSipRegisterExtClbks(
    RvIppSipExtClbks* clbks)
```

This function receives a structure with pointers to callback implementations as a parameter. The function should be called after `rvIppSipSystemInit()` and before any other Multimedia Terminal Framework API.

EXAMPLE

```
/* Step 1: Initialize RvIppSipExtClbks structure*/
RvIppSipExtClbks    userSipClbks =
{
    userSipStackConfig,
    userSipRegisterStackEvents,
    userSipPreCallLegCreatedIncoming,
    userSipPostCallLegCreatedIncoming,
    userSipPreCallLegCreatedOutgoing,
    userSipPostCallLegCreatedOutgoing,
    userSipPreStateChanged,
    userSipPreMsgToSend,
    userSipPostMsgToSend,
    userSipPreMsgReceived,
    userSipPostMsgReceived,
    userSipPreRegClientStateChanged,
    &userInfo
};

rvIppSipSystemInit();
.....
/* Step 2: Register Extension callbacks */
rvIppSipRegisterExtClbks (&userSipClbks);
.....
/* Initialize Multimedia Terminal Framework */
rvIppSipStackInitialize(stackHandle, &cfg);
rvIppSipPhoneConstruct(termMgr, &cfg);
```

SIP EXTENSION CALLBACKS

The following list describes the Extension callbacks that the user application can implement:

```
typedef void (*RvIppSipStackConfigCB) (
    RvSipStackCfg* pStackCfg);
```

Description

Enables the user application to change the configuration parameters of the Stack. When this function is called, pStackCfg contains default values that can be modified. After the function returns, validation checking is done on the values of pStackCfg. Invalid values are overridden with default ones, while unsupported parameters are ignored.

Invoked

Before the SIP Stack is constructed and after the Terminal Framework has set its default Stack configuration values.

User application can

Override a partial list of SIP Stack parameters, while the remainder cannot be modified. The user application can override the following parameters (for more information on each of these parameters, see the RADVISION SIP Stack Programmer Guide):

- maxCallLegs—This parameter has already been configured by the user application as part of Multimedia Terminal Framework initialization. For more information, see the parameter "maxCallLegs" in the section [Configuration](#) of the [Building Your Application](#) chapter.
- maxTransactions
- maxRegClients—This parameter has already been configured by the user application as part of Multimedia Terminal Framework initialization. For more information, see the parameter "maxRegClients" in the section [Configuration](#) of the [Building Your Application](#) chapter.
- messagePoolNumofPages
- messagePoolPageSize
- generalPoolNumofPages

- generalPoolPageSize
- tcpEnabled—This parameter has already been configured by the user application as part of Multimedia Terminal Framework initialization. For more information, see the parameter "tcpEnabled" in the section [Configuration](#) of the [Building Your Application](#) chapter.
- maxConnections
- ePersistencyLevel
- serverConnectionTimeout
- outboundProxyHostName
- eOutboundProxyTransport
- sendReceiveBufferSize
- bUseRportParamInVia
- processingQueueSize
- numOfReadBuffers
- retransmissionT1
- retransmissionT2
- generalLingerTimer
- inviteLingerTimer
- provisionalTimer
- retransmissionT4
- cancelGeneralNoResponseTimer
- cancelInviteNoResponseTimer
- generalRequestTimeoutTimer
- maxElementsInSingleDnsList
- maxSubscriptions
- subsAlertTimer
- subsNoNotifyTimer
- subsAutoRefresh
- processingTaskPriority
- bDisableMerging
- minSE
- sessionExpires
- numOfDnsServers

- pDnsServers
- numOfDnsDomains
- pDnsDomains

Return value

None.

EXAMPLES

Change Stack Configuration

This example shows how configuration parameters are changed in the SIP Stack.

1. Implement callback RvIppSipStackConfigCB

Implement Extension callback and set the required parameters in the SIP Stack.

```
void userSipStackConfig(RvSipStackCfg* stackCfg)
{
    stackCfg->retransmissionT1 = 1000;
    stackCfg->retransmissionT2 = 1000;
}
```

2. Register an Extension callback

```
RvIppSipExtClbks    userSipClbks =
{
    userSipStackConfig,
    NULL,
    NULL
};

rvIppSipRegisterExtClbks(&userSipClbks);
```

REGISTER STACK EVENTS

```
typedef void (*RvIppSipRegisterStackEventsCB) (  
    RvIppSipControlHandle     sipMgrHndl,  
    RvSipStackHandle          stackHandle);
```

Description

Enables the user application to listen for Stack events for which the Multimedia Terminal Framework does not listen.

Invoked

Before the Stack is initialized and after the Terminal Framework has registered its default Stack events.

User application can

Register new Stack events.

Return value

None.

Note It is recommended not to override Stack events to which the Multimedia Terminal Framework listens (listed at the end of this section), but to use the PrexxxCB and PostxxxCB callbacks instead.

EXAMPLE

This example shows how the Stack parser is told to go ahead and process an invalid incoming SIP message. When a message with a syntax error is received, the parser sends the RvSipTransportBadSyntaxMsgEv event to the user application, which in turn tells the parser how it should continue. If the user application instructs the parser to continue and the parser can overcome the error, the Stack will handle the message despite the syntax errors. If the parser cannot overcome the error, the Stack will reject the message.

To achieve this, a new Stack event that the Multimedia Terminal Framework does not listen on should be implemented and registered: RvSipTransportBadSyntaxMsgEv.

1. Implement a new Stack event

Implement a new Stack callback, to be called each time the Stack receives an invalid SIP message.

```
RV_Status RVCALLCONV userSipTransportBadSyntaxMsgEv(
    IN RvSipTransportMgrHandle hTransportMgr,
    IN RvSipAppTransportMgrHandle hAppTransportMgr,
    IN RvSipMsgHandle hMsgReceived,
    OUT RvSipTransportBsAction *peAction)
{
    /*Inform the Stack to continue and process the message*/
    *peAction = RVSIP_TRANSPORT_BS_ACTION_CONTINUE_PROCESS;

    return RV_Success;
}
```

2. Register the new event to the SIP Stack

Register the new Stack event to the SIP Stack to enable listening on this event.

```
void userSipRegisterStackEvents(RvIppSipControlHandle sipMgrHndl,
                                 RvSipStackHandle stackHandle)
{
    RvSipTransportMgrHandle hTransportMgr = userInfo.transportMgr;
    RvSipTransportMgrEvHandlers* appEvHandlers =
        rvIppSipControlGetStackCallbacks(sipMgrHndl);

    /*Set a handle to the transport manager*/
    RvSipStackGetTransportMgrHandle(stackHandle, &hTransportMgr);

    /*Reset the appEvHandlers since not all callbacks are set by
    this application*/
```

```
    memset (appEvHandlers, 0, sizeof (RvSipTransportMgrEvHandlers)) ;  
  
    /*Set application callbacks in the structure*/  
    appEvHandlers->pfnEvBadSyntaxMsg =  
        userSipTransportBadSyntaxMsgEv;  
  
    /*Set the structure in the call-leg manager*/  
    rv = RvSipTransportMgrSetEvHandlers (transportMgr,  
                                         NULL,  
                                         appEvHandlers,  
                                         sizeof (RvSipTransportMgrEvHandlers));  
}  
  
3. Register an Extension callback  
  
RvIppSipExtClbks      userSipClbks =  
{  
    NULL,  
    userSipRegisterStackEvents,  
    NULL,  
    NULL  
};  
  
rvIppSipRegisterExtClbks (&userSipClbks) ;
```

PRE-PROCESS INCOMING NEW CALL

```
typedef RvBool (*RvIppSipPreCallLegCreatedIncomingCB) (
    RvIppSipControlHandle      sipMgrHndl,
    RvSipCallLegHandle         hCallLeg,
    RvSipAppCallLegHandle*    phAppCallLeg,
    void*                      userData);
```

Description

Enables the user application to handle an incoming Invite before the Multimedia Terminal Framework handles it.

Invoked

When the Stack event RvSipCallLegCreatedEv is received and before the Multimedia Terminal Framework processes the event. The event is received every time an Invite message is received.

Use application can

Retrieve information from the message, decide to let the user application process the message, or have the Multimedia Terminal Framework do its default processing.

Return value

False to inform the Multimedia Terminal Framework that the event should be ignored. True to keep the default processing.

POST-PROCESS INCOMING NEW CALL

```
typedef void (*RvIppSipPostCallLegCreatedIncomingCB) (
    RvIppSipControlHandle      sipMgrHndl,
    RvSipCallLegHandle         hCallLeg,
    RvSipAppCallLegHandle*    phAppCallLeg,
    void*                      userData);
```

Description

Enables the user application to handle an incoming Invite after the Multimedia Terminal Framework handles it.

Invoked

When the Stack event RvSipCallLegCreatedEv is received and after the Multimedia Terminal Framework has processed a new incoming Invite.

User application can

Retrieve information from message and objects.

Return value

None.

PRE-PROCESS OUTGOING NEW CALL

```
typedef void (*RvIppSipPreCallLegCreatedOutgoingCB) (
    RvIppSipControlHandle     sipMgrHndl,
    RvSipCallLegHandle        hCallLeg,
    char*                     to,
    char*                     from,
    void*                     userData);
```

Description

This callback indicates that a new Invite message is about to be built.

Invoked

After the call-leg object is created and before the Multimedia Terminal Framework adds headers to the call-leg object. This event is invoked every time the Multimedia Terminal Framework establishes a new outgoing call.

User application can

Change the TO and FROM headers of the outgoing new Invite. The "to" and "from" values returned by the user application are used to build TO and FROM headers in the outgoing Invite.

Return value

None.

POST-PROCESS OUTGOING NEW CALL

```
typedef void (*RvIppSipPostCallLegCreatedOutgoingCB) (
    RvIppSipControlHandle     sipMgrHndl,
    RvSipCallLegHandle        hCallLeg,
    void*                     userData);
```

Description

This callback indicates that a new Invite message is about to be sent.

Invoked

After the Multimedia Terminal Framework adds headers to the call-leg object, and before the message is sent out.

User application can

Change any of the default headers in an outgoing new Invite, or add new ones.

Return value

None.

PRE-PROCESS CALL-LEG STATE CHANGED

```
typedef RvBool (*RvIppSipPreStateChangedCB) (
    RvIppSipControlHandle           sipMgrHndl,
    RvSipCallLegHandle              hCallLeg,
    IN RvSipAppCallLegHandle        hAppCallLeg,
    IN RvSipCallLegState           eState,
    IN RvSipCallLegStateChangeReason eReason,
    void*                           userData);
```

Description

This callback indicates that a call-leg state has changed.

Invoked

When the Stack event RvSipCallLegStateChangedEv is received and before the Terminal Framework processes the event. This event is invoked every time the call-leg state changes (as a result of an incoming message or internal events).

User application can

Retrieve information, let the Multimedia Terminal Framework do its default processing, or have the user application handle the event.

Return value

False to inform the Multimedia Terminal Framework to ignore the event, or True to keep the default processing.

POST-PROCESS CALL-LEG STATE CHANGED

```
typedef void (*RvIppSipPreStateChangedCB) (
    RvIppSipControlHandle           sipMgrHndl,
    RvSipCallLegHandle              hCallLeg,
    IN RvSipAppCallLegHandle        hAppCallLeg,
    IN RvSipCallLegState           eState,
    IN RvSipCallLegStateChangeReason eReason,
    void*                           userData);
```

Description

This callback indicates that a call-leg state has changed.

Invoked

When the Stack event RvSipCallLegStateChangedEv is received and after the Terminal Framework processes the event. This event is invoked every time the call-leg state changes (as a result of an incoming message or internal events).

User application can

Retrieves information from the call-leg or message after it is processed by the Multimedia Terminal Framework.

Return value

None.

Note

When eState equals RVSIP_CALL_LEG_STATE_DISCONNECTED, hAppCallLeg is no longer valid and is set to NULL.

PRE-PROCESS MESSAGE TO SEND

```
typedef RvBool (*RvIppSipPreMsgToSendCB) (
    RvIppSipControlHandle      sipMgrHndl,
    RvSipCallLegHandle         hCallLeg,
    IN RvSipAppCallLegHandle   hAppCallLeg,
    IN RvSipMsgHandle          hMsg,
    void*                      userData);
```

Description

This callback indicates that a SIP message is about to be sent and has not yet been processed.

Invoked

When the Stack event RvSipCallLegMsgToSendEv is received and before the Terminal Framework modifies the message. This event is invoked every time a new call-leg message is about to be sent.

User application can

Modify the outgoing SIP message: Change or add headers, change or add a body message. The user application can also decide to let the Multimedia Terminal Framework do its default processing or have the user application handle the event.

Return value

False to inform the Multimedia Terminal Framework to ignore the event, or True to keep the default processing.

POST-PROCESS MESSAGE TO SEND

```
typedef void (*RvIppSipPostMsgToSendCB) (
    RvIppSipControlHandle      sipMgrHndl,
    RvSipCallLegHandle         hCallLeg,
    IN RvSipAppCallLegHandle   hAppCallLeg,
    IN RvSipMsgHandle          hMsg,
    void*                      userData);
```

Description

This callback indicates that a SIP message is about to be sent, and has already been processed by the Multimedia Terminal Framework.

Invoked

When the Stack event RvSipCallLegMsgToSendEv is received and after the Terminal Framework modifies the message. This event is invoked every time a new call-leg message is about to be sent.

Use application can

Retrieve information from the message, modify the outgoing SIP message after default processing by the Multimedia Terminal Framework.

Return value

None.

PRE-PROCESS MESSAGE RECEIVED

```
typedef RvMtfMsgProcessType (*RvIppSipPreMsgReceivedCB) (
    RvIppSipControlHandle      sipMgrHndl,
    RvSipCallLegHandle         hCallLeg,
    IN RvSipAppCallLegHandle   hAppCallLeg,
    IN RvSipMsgHandle          hMsg,
    void*                      userData);
```

Description

This callback indicates that a message has been received and has not yet been processed.

Invoked

When the Stack event RvSipCallLegMsgReceivedEv is received and before the Terminal Framework modifies the message. This event is invoked every time a new call-leg message is received.

User application can

Handle incoming SIP messages and decide whether to let the Multimedia Terminal Framework do its default processing, or to let the user application handle the event.

Return value

RvMtfMsgProcessType can be set to one of the following values:

RV_MTF_IGNORE_BY_STACK

This value indicates to both the SIP Stack and the Multimedia Terminal Framework to ignore the message. When this value is returned, the callback RvIppSipPreCallLegCreatedIncomingCB() should return False as well. Otherwise, a memory leak will occur (when RvIppSipPreCallLegCreatedIncomingCB() is called, resources—of the SIP connection—will be allocated but not released).

RV_MTF_IGNORE_BY_MTF

This value indicates to the Multimedia Terminal Framework to ignore the message, but the message will still be processed by the SIP Stack.

RV_MTF_DONT_IGNORE

This value indicates to both the SIP Stack and the Multimedia Terminal Framework to process the message.

POST-PROCESS MESSAGE RECEIVED

```
typedef void (*RvIppSipPostMsgReceivedCB) (
    RvIppSipControlHandle      sipMgrHndl,
    RvSipCallLegHandle         hCallLeg,
    IN RvSipAppCallLegHandle   hAppCallLeg,
    IN RvSipMsgHandle          hMsg,
    void*                      userData);
```

Description

This callback indicates that a message has been received and has been processed by the Multimedia Terminal Framework.

Invoked

When the Stack event RvSipCallLegMsgReceivedEv is received and after the Terminal Framework has modified the message. This event is invoked every time a new call-leg message is received.

User application can

Handle an incoming SIP message and retrieve information from the message.

Return value

None.

REGISTERCLIENT STATE CHANGED

```
typedef RvBool (*RvIppSipPreRegClientStateChangedCB) (
```

RvIppSipControlHandle	sipMgrHndl,
RvSipRegClientHandle	hRegClient,
RvIppTerminalHandle	mdmTerminalHndl,
RvSipRegClientState	eState,
RvSipRegClientStateChangeReason	eReason,
void* userData);	

Description

This callback indicates a change in state of RegClient and enables the user application to be informed of the status of a Registration to the SIP server.

Invoked

When a Stack event RvSipRegClientStateChangedEv is invoked and before the default processing of Multimedia Terminal Framework.

User application can

Retrieve information on Registration status, send a new Registration request based on RegClient state, handle the event or inform the Multimedia Terminal Framework to ignore the event.

Return value

False to inform the Multimedia Terminal Framework to ignore the event, or True to keep the default processing.

EXTENSION APIs AND CALLBACK FILES

When implementing Extension callbacks, the user application has access to the following functions:

Functions	Description	File
Multimedia Terminal Framework interface		rvmmdm.h, rvMdmControlApi.h
SIP Control objects	APIs of the objects available in the callback. Refer to the Multimedia Terminal Framework Reference Guide for the complete list of APIs.	rvSipControlApi.h
SIP Stack APIs	SIP Stack APIs are available according to Stack restrictions and limitations (as explained in the Stack documentation). Note: APIs that change the state of the call are not allowed to be used in Extension callbacks.	SIP Stack API files (see the SIP Stack documentation)
User data	A pointer to the user application data, which can be set by calling functions of type: rvXXXGetUserData() and rvXXXSetUserData() for the objects available in the callback.	rvmmdm.h, rvmmdmobjects.h
User application functions		

SIP STACK EVENTS ON WHICH THE MULTIMEDIA TERMINAL FRAMEWORK LISTENS

Call-Leg Events

- RvSipCallLegCreatedEv
- RvSipCallLegStateChangedEv
- RvSipCallLegMsgToSendEv
- RvSipCallLegMsgReceivedEv
- RvSipCallLegModifyStateChangedEv
- RvSipCallLegReferStateChangedEv
- RvSipCallLegReferNotifyEv
- RvSipCallLegFinalDestResolvedEv—Only when STUN is enabled
- RvSipCallLegTrancRequestRcvdEv

Transaction Events

- RvSipTransactionCreatedEv
- RvSipTransactionStateChangedEv
- RvSipTransactionMsgToSendEv
- RvSipTransactionMsgReceivedEv
- RvSipTransactionOpenCallLegEv

RegClient Events

- RvSipRegClientStateChangedEv
- RvSipRegClientMsgToSendEv
- RvSipRegClientMsgReceivedEv
- RvSipRegClientFinalDestResolvedEv—Only when STUN is enabled

Authenticator Events

- RvSipAuthenticatorMD5Ev
- RvSipAuthenticatorSharedSecretEv
- RvSipAuthenticatorGetSharedSecretEv

Transport Events

- RvSipTransportBufferReceivedEv—Only when STUN is enabled

Subscriber Events

- RvSipSubsFinalDestResolvedEv—Only when STUN is enabled

Transmitter Events

- RvSipTransmitterStateChangedEv—Only when STUN is enabled

SIP Control

5

SAMPLE APPLICATION

5.1 WHAT'S IN THIS CHAPTER

This chapter describes the sample application that is provided by the Multimedia Terminal Framework:

- Introduction
- Configuration
- GUI Application
- Running the Sample Application
- Structure and Objects
- Sample Code
- EPP
- Integrating MTF without EPP
- IPV6
- Message Waiting Indication

5.2 INTRODUCTION

As part of Terminal Framework support, RADVISION provides a sample application that is in fact a complete reference implementation. The sample application simulates physical terminations through a GUI application that connects to it through a simple protocol.

The sample application aims to demonstrate how the RADVISION Multimedia Terminal Framework is used to build a Terminal Framework or a Residential Gateway.

WORK ENVIRONMENTS

As illustrated in [Figure 5-1](#), the Multimedia Terminal Framework sample application is composed of two applications:

- MtfSipSample application—this is the integration with the Multimedia Terminal Framework. It includes Multimedia Terminal Framework initializations, configurations, callback implementations, etc. Runs on any supported operating system.
- GUI application—handles user interface graphics and media engine. Runs on Windows only.

The two applications can run on the same machine. Alternatively, the GUI application can run on one machine and the MtfSipSample application can run remotely on a different machine. This separation enables running MtfSipSample on any operating system while using the same GUI application. The protocol between them is called EPP; it is a proprietary protocol. For more information, see the sections [EPP](#) and [Structure and Objects](#) in this chapter.

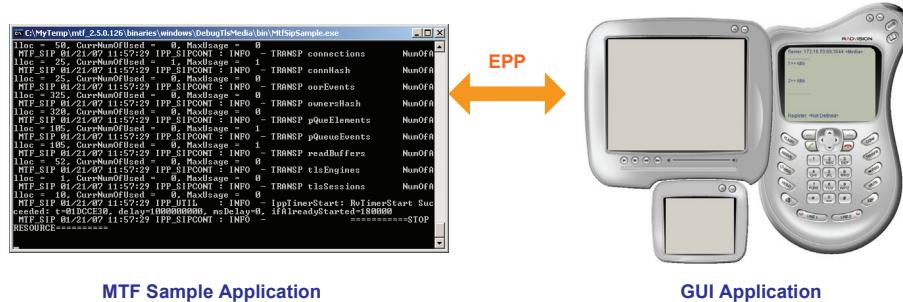


Figure 5-1 *Sample Application*

REQUIREMENTS

- Speakers or a headset should be plugged into the machine on which the GUI application will run
- A microphone should be plugged into the machine on which the GUI application will run

5.3 CONFIGURATION

This section describes how the MtfSipSample application is configured.

CONFIGURATION FILE

The configuration file name is *sippPhone.cfg*. It is structured as an *.ini* file.

The configuration file is located in one of the following:

- Windows, Linux and WinMobile—the SIP directory
- vxWorks and MonteVista—the FTP directory on the remote machine

Configuration file parameters

The configuration file includes:

- MTF configuration parameters
- SIP Stack configuration
- MTF sample application configuration parameters

Table 5-1 lists all MtfSipSample configuration parameters. For each parameter it is indicated whether it is an MTF parameter or a sample application parameter.

Note A parameter line that starts with ";" will be ignored and set to the default, except in the Media Parameters sections, in which a comment is indicated by the "#" sign.

Table 5-1

#	Parameter Name	MTF Parameter or Sample Parameter	Reference to RvIppSipPhoneCfg (MTF) or Description (Sample)
General Parameters			
1	LocalAddress	MTF	localAddress. You may configure 0.0.0.0 instead of your actual IP address. (In this case, the first host in the local hosts list will be chosen.)
2	UiAlias	Sample	UI Termination ID. This parameter is used as termination ID when registering a UI termination to the Multimedia Terminal Framework

Configuration

Table 5-1

3	MaxCallLegs	MTF	maxCallLegs
4	ReferTimeout	MTF	referTimeout
5	enableSdpLogs	Sample	When set to True, SDP log messages will be printed to the log file.
6	ConnectOn180	MTF	connectOn180
7	autoAnswer	Sample	When Set to True, the incoming call will be answered automatically (an offHook event will be sent to the MTF). When the remote party sends BYE, an onHook event is sent to the MTF.
8	DialToneDuration	MTF	dialToneDuration
9	UserDomain	MTF	userDomain
10	CallWaitingReply	MTF	callWaitingReply
11	WatchdogInterval	MTF	watchdogTimeout
12	DtmfRelay	MTF	outOfBandDtmf

Internal

13	EppUseTcp	Sample	When set to True, EPP will work in TCP mode. When set to False, EPP will work in UDP mode. EPP must work in TCP mode in Windows Mobile. For other operating systems it is recommended to work in UDP mode. Setting the value of this parameter requires matching the configuration of EPP_USE_TCP in the Ipp.Xml file.
14	EppPort	Sample	Port number for communication with GUI. Default: 3044
15	EppIp	Sample	IP address for communication with GUI. Possible values: IPV4 or IPV6. Relevant when working with IPV6 only. If so, this parameter must be set to the IPV4 local address. May be ignored when working with IPV4. Default: Same as LocalAddress parameter.
16	localIpPrefix	Sample	Used to find the local address when localAddress is set to 0.0.0.0.

Table 5-1

17	localIpMask	Sample	Used to find the local address when localAddress is set to 0.0.0.0.
SIP Servers			
18	RegistrarAddress	MTF	registrarAddress
19	RegistrarPort	MTF	registrarPort
20	username	MTF	username
21	password	MTF	password
22	AutoRegistration	MTF	autoRegister
23	RegistrationExpire	MTF	registrationExpire
24	UnregistrationExpire	MTF	unregistrationExpire
25	MaxRegisterClients	MTF	maxRegClients
26	OutboundProxyAddress	MTF	outboundProxyAddress
27	OutboundProxyPort	MTF	outboundProxyPort
28	SubsServerName	Sample	IP address or domain name of Subscribe Server (for MWI).
STUN Parameters			
29	StunServerAddress	Sample	IP address or domain name of the STUN Server.
30	StunServerPort	Sample	Port number of the STUN Server.
31	StunNeedMask	Sample	IP address mask for addresses for which the STUN should be used.
32	StunClientResponsePort	Sample	Port number for receiving replies from the STUN Server.
Presentation Parameters			
33	callerIdPermission	Sample	This parameter is not in use.

Configuration

Table 5-1

34	calleeIdPermission	Sample	This parameter is not in use.
35	presentationName	Sample	Used to configure a termination with Caller ID.
TLS Parameters			
36	stackTlsAddress	SIP Stack	localTlsAddresses
37	stackTlsPort	SIP Stack	localTlsPorts
38	stackNumOfTlsAddresses	MTF	numOfTlsAddresses
39	certDepth=1	MTF	certDepth
40	privateKeyType	MTF	privateKeyType
41	privateKeyFileName	Sample	Absolute path and name of the private key certificates file to be read by openSSL.
42	tlsMethod	MTF	tlsMethod
43	caCertFileName	Sample	Absolute path and name of the CA certificates file to be read by openSSL.
44	tlsPostConnectAssertFlag	Sample	Used to implement the RvIppSipTlsPostConnectionAssertionCB() callback. Possible values: 1—indicates that the connection will be asserted successfully (the callback returns True). 0—indicates that the assertion has failed (the callback returns False).
IPP Logging Parameters			
45	Parameters in the section [IppLogOptions]	Sample	The sample application uses these parameters when calling IppLogInit() to configure logging of various Multimedia Terminal Framework modules. Possible values: ERROR, EXCEPTION, WARNING, INFO, DEBUG, ENTER, LEAVE, SYNC

Table 5-1**SIP Logging Parameters**

46	Parameters in the section [SIPLogOptions]	MTF	The sample application uses these parameters to configure logging of various SIP Stack modules. Possible values: ERROR ,EXCEPTION, WARNING, INFO, DEBUG, ENTER, LEAVE, SYNC
----	--	-----	---

Media Capabilities

47	Parameters in section [MediaParameter 0]	Sample	The sample application uses this parameter when calling rvMdmTermClassAddMediaCapabilities() to load media capabilities to the Multimedia Terminal Framework. Example: v=0 o=rtp/1 \$ \$ IN IP4 \$ s=- c=IN IP4 \$ t=\$ \$ m=audio \$ RTP/AVP 0 a=rtpmap:0 G711/8000 a=ptime:144 a=SilenceSupp:on
----	---	--------	--

Media for Re-Invite

48	Parameters in section [MediaParameter 1]	Sample	The sample application uses this parameter when calling rvMdmTermModifyMedia() to send a Re-Invite message. Example: v=0 o=rtp/1 \$ \$ IN IP4 \$ s=- c=IN IP4 \$ t=\$ \$ m=audio \$ RTP/AVP 0 a=rtpmap:0 G711/8000 a=ptime:144 a=SilenceSupp:on
----	---	--------	--

Configuration

Table 5-1

Phone Numbers

49	Parameters in section [PhoneNumbers]	Sample	The sample application uses this list to map between phone number and destination address when implementing the callback RvMdmTermMapDialStringToAddressCB(). Example: 1001 ip-001@172.20.77.1 2001 an-001@172.20.77.1
----	---	--------	---

SIP Stack Parameters

50	Parameters in the section [SIPStackParameters]	Sample	The sample application uses these parameters to override the default values set by the Multimedia Terminal Framework by implementing RvIppSipStackConfigCB (see the function <i>userSipStackConfig()</i> in IppSipSample).
----	---	--------	--

5.4 GUI APPLICATION

OVERVIEW

The GUI application simulates the terminal endpoint. The GUI application code is based on C and C++/MFC and runs on Windows only.

The GUI application handles the following:

- **User interface**—all graphics, including buttons, indicators, text display, etc.
- **Events and signals**—whenever the user presses a key, the GUI application sends event messages to the Multimedia Terminal Framework. When signal messages are received from MtfSipSample, the GUI application updates the user interface (text display, indicators, tones, etc.).
- **Media processing and media engine**—the GUI application processes media messages sent by MtfSipSample for performing media operations, such as opening a new media stream, connecting media streams, modifying existing media streams, etc.

Note The GUI application simulates an IP Phone terminal. It will simulate an analog terminal in future versions of the Multimedia Terminal Framework.



Figure 5-2 *GUI Application*

MEDIA CAPABILITIES

- Audio codec G711 mu-law
- Jitter buffer

Note In this version, video codecs are not supported by the GUI application.

CONFIGURING THE GUI APPLICATION

The configuration file name is *IPP.xml*, which is structured as an XML file. This file should be located in the same directory as *GUI.exe*.

GUI Application parameters should be configured as follows.



To configure the GUI Application

1. Configure the following IP addresses (IPV4 only):
 - CLIENT_ADDR—GUI local address. You may set 0.0.0.0. as default. In this case, the first on local hosts will be chosen.
 - CLIENT_PORT_GUI port number—can be any available port number.
 - SERVER_ADDR—IP address on which the sample application is running.
 - SERVER_PORT—port number on which the sample application listens. Should be the same as the parameter EppPort in the sample application configuration.
 - EPP_USE_TCP—when the value is set to 1, EPP is indicated to work in TCP mode. When the value is set to 0, EPP will work in UDP mode. Working in TCP mode is mandatory for Windows Mobile. For all other operating systems it is recommended to work in UDP mode. Setting the value to 1 requires a matching configuration of the parameter EppUseTcp in *sipphone.cfg*.

2. To allow running without video:

- ❑ In the IP Phone section, under <REGISTRATION>, remove the following lines:
<UNREG_MSG message="unregister vt/cam ippone "/>
<REG_MSG message="register vt/cam ippone "/>

When these lines are removed, the Register message will not be sent from the GUI application to the Multimedia Terminal Framework sample application, and the function rvMdmTermMgrRegisterPhysicalTerminationAsync() will not be called for the video termination.

5.5 RUNNING THE SAMPLE APPLICATION

OVERVIEW

This section explains how to run the Multimedia Terminal Framework sample application:

- [Basic Configuration and Setup](#)
- [Getting Started](#)—Registration, basic outgoing/incoming calls
- [Call Control Features](#)—Transfer, Call Forward, etc.

For other features supported by the Multimedia Terminal Framework, such as Authentication, TLS, Session timer, PRACK, etc., see the [Features](#) chapter.

BASIC CONFIGURATION AND SETUP

Requirements

- Plug headset and microphone into the machine on which the GUI is run
- Establish two or three sets (i.e., MtfSipSample and GUI application) of the sample application so that you can make calls between them, or run one set versus an application by a different vendor.

Configuration

1. Change the configuration in *sipphone.cfg* as follows:
 - Make sure the parameter **LocalAddress** is set to 0.0.0.0, or set it to your local IP address.
 - Configure the remaining parameters, all of which are optional (see the section [Configuration](#)).
2. Allowing a call to be established between two MTF sample application sets can be configured in two different ways:
 - **Option 1: Local Mapping**

The phone number and IP address of the destination are configured in *sipphone.cfg* (phone numbers section):

- ◆ Configure **UiAlias** = <your username>
- ◆ Configure phone numbers and SIP URI of the destination party, while the username of the SIP URI is the same as the destination **UiAlias** parameter.

- ◆ Run the MTF sample application and register the GUI application to *MtfSipSample* (see [Getting Started](#)).

For example:

Configuration of A:

Local Address = 0.0.0.0 or 172.16.70.80

Udp Port = 5060

UiAlias = 1001

[PhoneNumbers]

1002 1002@172.16.70.70:5070

Configuration of B:

Local Address = 0.0.0.0 or 172.16.70.70

Udp Port = 5070

UiAlias = 1002

[PhoneNumbers]

1001 1001@172.16.70.80

Now A can dial B by dialing 1002, and B can dial A by dialing 1001.

Option 2: Server Mapping

Both parties should be registered with the same SIP Registrar:

- ◆ Configure UiAlias as digits only.
- ◆ Configure the same Registrar address in both local and remote parties.
- ◆ Configure the phone number and UiAlias of the destination. There is no need to configure the IP address, as the message is sent to the Registrar, which then routes it to the destination.
- ◆ Run the MTF sample application and register the GUI application to *MtfSipSample* (see [Getting Started](#)).

Running the Sample Application

- ◆ Make sure both parties are registered with the SIP Registrar.

For example:

Configuration of A:

```
UiAlias = 1001  
RegistrarAddress=Radvision.com  
[PhoneNumbers]  
1002 1002
```

Configuration of B:

```
UiAlias = 1002  
RegistrarAddress=Radvision.com  
[PhoneNumbers]  
1001 1001
```

Now A can dial B by dialing 1002, and B can dial A by dialing 1001.

Tip

To enable running more than one sample application (set of MtfSipSample and GUI application) on the same machine, for each set of MtfSipSample and GUI application configure a different EPP port number in the following parameters: EppPort (*sipphone.cfg*) and SERVER_PORT (*IPP.xml*).

For example:

Configuration of A:

```
EppPort=3044  
SERVER_PORT=3044
```

Configuration of B:

```
EppPort=4044  
SERVER_PORT=4044
```

GETTING STARTED

Registration

Initial state: None.

Table 5-2

Step #	Description	GUI Indications	SIP indications
1	Run MtfSipSample	<ul style="list-style-type: none"> On Windows, a console window opens. On Windows Mobile, a window opens with EXIT button. On any other OS, <i>MtfSipSample</i> process starts running. 	N/A
2	Run GUI application (<i>Gui.exe</i>)	Text display appears: Server: <ip address>:<port><Media> IP address and port of MtfSipSample as configured in <i>IPP.xml</i> in SERVER_ADDR and SERVER_PORT parameters. <Media> is a constant string for internal use.	N/A
3	Right-click GUI	Floating menu is opened.	N/A
4	Choose Register*	Text display appears: 1>>Idle 2>>Idle	If the Registrar address has been configured, a Register message is sent out.

* Registration is done automatically when the GUI application initiates. If MtfSipSample is already running when the GUI application is initialized, there is no need to perform steps 3 and 4.

Running the Sample Application

Basic Outgoing Call

Initial state: The GUI application is registered with MtfSipSample.

Table 5-3

Step #	Description	GUI Indications	SIP indications
1	Press Off Hook or Line 1 or Line 2	<ul style="list-style-type: none">• Line 1 is on• Dial tone is heard• Text Display: 1>>Dial:	N/A
2	Dial destination phone number	<ul style="list-style-type: none">• Line 1 is on• Text Display: 1>>Dial: 1001	Invite message is sent out.
3	Wait until remote party answers	<ul style="list-style-type: none">• Line 1 is on• Ringback tone is heard• Text Display: 1>>Calling: 1001	Optional: A provisional message is received (100 or 180 or 183).
4	Call is connected	<ul style="list-style-type: none">• Ringback tone stops• Text Display: 1>>Talking: 1001	200 OK is received.

Basic Incoming Call

Initial state: The GUI application is registered with MtfSipSample.

Table 5-4

Step #	Description	GUI Indications	SIP indications
1	N/A	<ul style="list-style-type: none">• Line 1 blinks• Ringing tone is heard• Text Display: 1>>Call From: 1002	<ul style="list-style-type: none">• New Invite message arrives• 180 Ringing is sent out
2	Press off Hook or Line 1: Call is connected	<ul style="list-style-type: none">• Line 1 is on• Ringing tone stops• Text Display: 1>>Talking: 1001	200 OK message is sent out.

CALL CONTROL FEATURES

Hold/Unhold

Initial state: Call is connected on Line 1 or 2.

Table 5-5

Step #	Description	GUI Indications	SIP indications
1	Press Hold button	<ul style="list-style-type: none"> • Line 1 blinks • Hold indicator is on • Text Display: 1>> On Hold 	<ul style="list-style-type: none"> • Invite (a=sendonly) is sent out • 200 OK is received
2	Press Line 1 button	<ul style="list-style-type: none"> • Line 1 is on • Hold indicator is off • Text Display: 1>>Talking: 1001 	<ul style="list-style-type: none"> • Invite (a=sendrecv) is sent out • 200 OK is received

Incoming Call Waiting

Initial state: Call is connected on Line 1.

Table 5-6

Step #	Description	GUI Indications	SIP indications
1	N/A	<ul style="list-style-type: none"> • Line 1 is on • Line 2 blinks • Call Waiting tone is heard • Text Display: 1>>Talking: 1001 • Text Display: 2>>Call From:1002 	<ul style="list-style-type: none"> • New Invite message arrives • 180 Ringing is sent out
2	Press Hold button*	<ul style="list-style-type: none"> • Line 1 blinks • Line 2 blinks • Hold indicator is on • Text Display: 1>> On Hold • Text Display: 2>>Call From:1002 	<ul style="list-style-type: none"> • Invite (a=sendonly) is sent out to party on Line 1 • 200 OK is received from party on Line 1

Running the Sample Application

Table 5-6

3	Press Line 2 button**	<ul style="list-style-type: none">• Line 1 blinks• Line 2 is on• Hold indicator is on• Call Waiting tone stops• Text Display: 1>> On Hold• Text Display: 2>>Talking:1002 <ul style="list-style-type: none">• 200 OK is sent out to party on Line 2
---	-----------------------	---

* If Hold is not pressed at this point, the call on Line 1 will be disconnected when Line 2 is pressed.

** You can toggle between the lines by pressing Hold + Line 1 or 2.

Mute/Unmute

Initial state: Call is connected on Line 1 or 2.

Table 5-7

Step #	Description	GUI Indications	SIP indications
1	Press Mute button (to mute the call)	Line 1 is on Mute indicator is on	N/A
2	Press Mute button (to unmute the call)	Line 1 is on Mute indicator is off	N/A

Call Forward

Initial state: Phone is idle.

Table 5-8

Step #	Description	GUI Indications	SIP indications
1	Choose Call Forward type: 1. Press Settings button (located next to the Minimize button). 2. Click Call Control tab. 3. In Call Forward Type, choose the required type: Unconditional, Busy, or No Reply.	Call Forward type appears selected in Call Control tab.	N/A
2	Press CFW button	Dial tone is heard	N/A
3	Dial destination phone number*	<ul style="list-style-type: none"> • Dial tone stops • Text Display: CFU: 1002 	N/A

* The phone number should be configured. See the section [Getting Started](#).



To cancel Call Forward

④ Press the **CFW** button.

The CFW Text display disappears.

Running the Sample Application

Blind Transfer

Initial state: Call is connected on Line 1.

Table 5-9

Step #	Description	GUI Indications	SIP indications
1	Choose the Transfer type: 1. Press Settings button (located next to the Minimize button). 2. Click Call Control tab. 3. In Transfer Type, choose Blind.	Blind appears selected in Call Control tab.	N/A
2	Press TRANSFR button.	<ul style="list-style-type: none">• Line 1 blinks• Line 2 is on• Hold indicator is on• Dial tone is heard• Text Display: 1>> On Hold• Text Display: 2>>Dial:	<ul style="list-style-type: none">• Invite (a=sendonly) is sent out to the party on Line 1.• 200 OK is received from the party on Line 1.
3	Dial destination phone number.	<ul style="list-style-type: none">• Line 1 blinks• Line 2 is on• Hold indicator is on• Dial tone stops• Text Display: 1>> On Hold• Text Display: 2>>Idle	<ul style="list-style-type: none">• A Refer message is sent out to the party on Line 1.
4	Wait a few seconds...	<ul style="list-style-type: none">• Line 1 blinks• Line 2 is off• Hold indicator is on• Text Display: 1>> On Hold• Text Display: 2>>Idle	<ul style="list-style-type: none">• 202 Accepted is received from the party on Line 1.
5	Press Line 1.	Line 1 is idle	<ul style="list-style-type: none">• BYE is sent out to the party on Line 1.• 200 OK is received from the party on Line 1.

Attended Transfer

Initial state: Call is connected on Line 1.

Table 5-10

Step #	Description	GUI Indications	SIP indications
1	Choose Transfer type: 1. Press Settings button (located next to Minimize button). 2. Click Call Control tab. 3. In Transfer Type, choose Attended.	Attended appears selected in Call Control tab.	N/A
2	Press TRANSFR button	<ul style="list-style-type: none"> • Line 1 blinks • Line 2 is on • Hold indicator is on • Dial tone is heard • Text Display: 1>> On Hold • Text Display: 2>>Dial: 	<ul style="list-style-type: none"> • Invite (a=sendonly) is sent out to party on Line 1 • 200 OK is received from party on Line 1
3	Dial destination phone number	<ul style="list-style-type: none"> • Line 1 blinks • Line 2 is on • Hold indicator is on • Dial tone stops • Ringback tone is heard • Text Display: 1>> On Hold • Text Display: 2>>Calling:1002 	<ul style="list-style-type: none"> • Invite is sent out to party on Line 2 • Optional: 180 Ringing is received from party on Line 2
4	Wait until remote party answers	<ul style="list-style-type: none"> • Line 1 blinks • Line 2 is on • Hold indicator is on • Text Display: 1>> On Hold • Text Display: 2>>Talking:1002 	<ul style="list-style-type: none"> • 200 OK is received from party on Line 2

Running the Sample Application

Table 5-10

5	Press TRANSFR button	<ul style="list-style-type: none">• Line 1 is off• Line 2 is off• Hold indicator off• Text Display: 1>> On Hold• Text Display: 2>>Talking:1002 <ul style="list-style-type: none">• REFER message is sent out to party on Line 1• 202 Accepted is received from party on Line 1• Optional – Notify is received from party on Line 1• BYE is received from party on Line 2• BYE is sent out to party on Line 1• 200 OK is sent to party on Line 2• 200 OK is received from party on Line 1
---	----------------------	--

Semi-Attended Transfer

Initial state: Call is connected on Line 1.

Table 5-11

Step #	Description	GUI Indications	SIP indications
1	Choose Transfer type: 1. Press Settings button (located next to Minimize button). 2. Click Call Control tab. 3. In Transfer Type, choose Attended.	Attended appears selected in Call Control tab.	N/A
2	Press TRANSFR button	<ul style="list-style-type: none">• Line 1 blinks• Line 2 is on• Hold indicator is on• Dial tone is heard• Text Display: 1>> On Hold• Text Display: 2>>Dial:	<ul style="list-style-type: none">• Invite (a=sendonly) is sent out to party on Line 1• 200 OK is received from party on Line 1

Table 5-11

3	Dial destination phone number	<ul style="list-style-type: none"> • Line 1 blinks • Line 2 is on • Hold indicator is on • Dial tone stops • Ringback tone is heard • Text Display: 1>> On Hold • Text Display: 2>>Calling:1002 	<ul style="list-style-type: none"> • Invite is sent out to party on Line 2 • Optional: 180 Ringing is received from party on Line 2
4	Press TRANSFR button	<ul style="list-style-type: none"> • Line 1 blinks • Line 2 is off • Hold indicator is off • Text Display: 1>> On Hold • Text Display: 2>>Calling:1002 	<ul style="list-style-type: none"> • Cancel is sent out to party on Line 2 • REFER message is sent out to party on Line 1 • 202 Accepted is received from party on Line 1 • Optional: Notify is received from party on Line 1 • BYE is sent out to party on Line 1 • 200 OK is received from party on Line 1
5	Press Line 1	Line 1 is idle	N/A

Transfer on a Held Call

Initial state: Call is connected on Line 1.

Table 5-12

Step #	Description	GUI Indications	SIP indications
1	Press Hold	<ul style="list-style-type: none"> • Line 1 blinks • Hold indicator is on 	<ul style="list-style-type: none"> • Invite (a=sendonly) is sent out to party on Line 1 • 200 OK is received from party on Line 1
2	Press Line 2	<ul style="list-style-type: none"> • Line 1 blinks • Hold indicator is on • Line 2 is on • Dial tone is heard 	N/A

Running the Sample Application

Table 5-12

3	Dial destination phone number	<ul style="list-style-type: none">• Line 1 blinks• Line 2 is on• Hold indicator is on• Dial tone stops• Ringback tone is heard• Text Display: 1>> On Hold• Text Display: 2>>Calling:1002	<ul style="list-style-type: none">• Invite is sent out to party on Line 2• Optional: 180 Ringing is received from party on Line 2
4	Wait until remote party answers	<ul style="list-style-type: none">• Line 1 blinks• Line 2 is on• Hold indicator is on• Text Display: 1>> On Hold• Text Display: 2>>Talking:1002	<ul style="list-style-type: none">• 200 OK is received from party on Line 2
5	Choose Transfer type: 1. Press Settings button (located next to Minimize button). 2. Click Call Control tab. 3. In Transfer Type, choose Attended.	Attended appears selected in Call Control tab	N/A
6	Press TRANSFR button	<ul style="list-style-type: none">• Line 1 is off• Line 2 is off• Hold indicator off• Text Display: 1>> On Hold• Text Display: 2>>Talking:1002	<ul style="list-style-type: none">• REFER message is sent out to party on Line 1• 202 Accepted is received from party on Line 1• Optional: Notify is received from party on Line 1• BYE is received from party on Line 2• BYE is sent out to party on Line 1• 200 OK is sent to party on Line 2• 200 OK is received from party on Line 1

3-Way Conference

Initial state: Call is connected on Line 1.

Table 5-13

Step #	Description	GUI Indications	SIP indications
1	Press CONF button	<ul style="list-style-type: none"> • Line 1 blinks • Line 2 is on • Dial tone is heard 	<ul style="list-style-type: none"> • Invite (a=sendonly) is sent out to party on Line 1 • 200 OK is received from party on Line 1
2	Dial destination phone number	<ul style="list-style-type: none"> • Line 1 blinks • Line 2 is on • Hold indicator is on • Dial tone stops • Ringback tone is heard • Text Display: 1>> On Hold • Text Display: 2>>Calling:1002 	<ul style="list-style-type: none"> • Invite is sent out to party on Line 2 • Optional: 180 Ringing is received from party on Line 2
3	Wait until remote party answers	<ul style="list-style-type: none"> • Line 1 blinks • Line 2 is on • Hold indicator is on • Text Display: 1>> On Hold • Text Display: 2>>Talking:1002 	<ul style="list-style-type: none"> • 200 OK is received from party on Line 2
4	Press CONF button	<ul style="list-style-type: none"> • Line 1 is on • Line 2 is on • Text Display: 1>> On Hold • Text Display: 2>>Talking:1002 	<ul style="list-style-type: none"> • Invite (a=sendrecv) is sent out to party on Line 1 • 200 OK is received from Line 1



To disconnect a Conference call

◎ Press the CONF button.

Running the Sample Application

3-Way Conference on a Held Call

Initial state: Call is connected on Line 1.

Table 5-14

Step #	Description	GUI Indications	SIP indications
1	Press Hold	<ul style="list-style-type: none">• Line 1 blinks• Hold indicator is on	<ul style="list-style-type: none">• Invite (a=sendonly) is sent out to party on Line 1• 200 OK is received from party on Line 1
2	Press Line 2	<ul style="list-style-type: none">• Line 1 blinks• Hold indicator is on• Line 2 is on• Dial tone is heard	N/A
3	Dial destination phone number	<ul style="list-style-type: none">• Line 1 blinks• Line 2 is on• Hold indicator is on• Dial tone stops• Ringback tone is heard• Text Display: 1>> On Hold• Text Display: 2>>Calling:1002	<ul style="list-style-type: none">• Invite is sent out to party on Line 2• Optional: 180 Ringing is received from party on Line 2
4	Wait until remote party answers	<ul style="list-style-type: none">• Line 1 blinks• Line 2 is on• Hold indicator is on• Text Display: 1>> On Hold• Text Display: 2>>Talking:1002	<ul style="list-style-type: none">• 200 OK is received from party on Line 2
5	Press CONF button	<ul style="list-style-type: none">• Line 1 is on• Line 2 is on• Text Display: 1>> On Hold• Text Display: 2>>Talking:1002	<ul style="list-style-type: none">• Invite (a=sendrecv) is sent out to party on Line 1• 200 OK is received from party on Line 1



To disconnect a Conference call

◎ Press the CONF button.

Re-Invite

Initial state: Call is connected on Line 1.

Table 5-15

Step #	Description	GUI Indications	SIP indications
1	Prepare SDP message: 1. Press Settings button (located next to Minimize button). 2. Click Dynamic Media Change tab. 3. Enter SDP message in text window.	<ul style="list-style-type: none"> • Line 1 is on • Text Display: 1>>Talking:1002 	N/A
2	In Dynamic Media Change tab, press Send Modify Media button.	<ul style="list-style-type: none"> • Line 1 is on • Text Display: 1>>Talking:1002 	Invite is sent out with new SDP

Unregister

Initial state: GUI is registered with MtfSipSample.

Table 5-16

Step #	Description	GUI Indications	SIP indications
1	Right-click GUI.	Floating menu opens	N/A
2	Choose Unregister.	None	If Registrar address is configured, Register message is sent out.

Shutdown

Initial state: Any state.

Table 5-17

Step #	Description	GUI Indications	SIP indications
1	Right-click GUI.	Floating menu opens	N/A
2	Choose Unregister and shutdown.	<ul style="list-style-type: none"> • MtfSipSample window is closed • GUI interface stays the same 	If connected calls exist, a BYE message is sent out for each call.

Running the Sample Application

Warm Restart

Initial state: Any state.

Table 5-18

Step #	Description	GUI Indications	SIP indications
1	Right-click GUI.	Floating menu opens	N/A
2	Choose Unregister and restart.	GUI interface is refreshed and returns to Idle state	If connected calls exist, a BYE message is sent out for each call.

5.6 STRUCTURE AND OBJECTS

MtfSipSample Structure

This section describes the structure of the Multimedia Terminal Framework sample application. [Figure 5-3](#) illustrates the main components in the Multimedia Terminal Framework sample application.

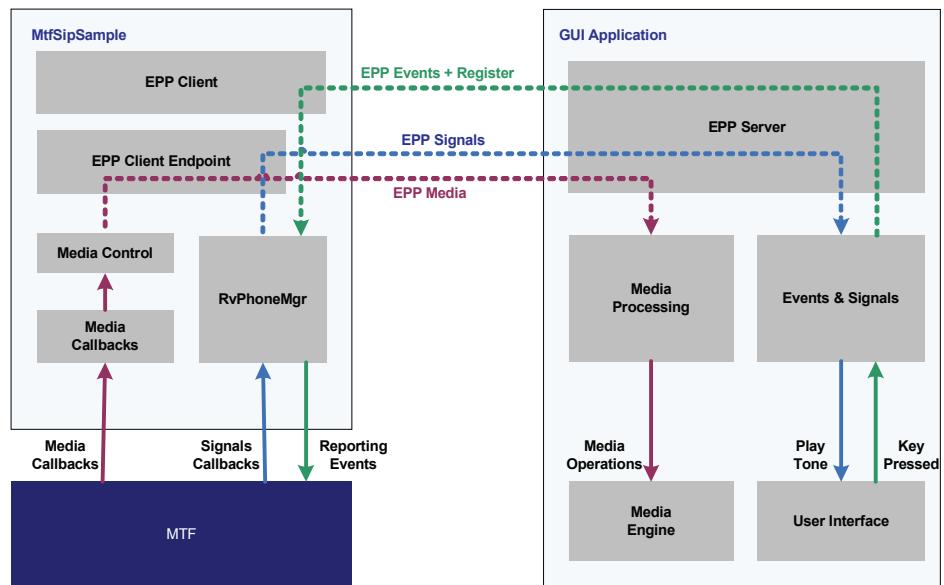


Figure 5-3 Sample Application Structure

RvPhoneMgr

This module includes:

- Implementation of signal callbacks invoked by the Multimedia Terminal Framework—for each signal callback, this module builds an EPP message and sends it out to the GUI application through the EPP client endpoint.
- Reporting events to the Multimedia Terminal Framework—whenever an EPP event message is received by the EPP client, this module parses the message, builds a Multimedia Terminal Framework event, and sends the event to the Multimedia Terminal Framework.

See the files `IppSipSamplePhonelink.c` and `IppSamplePhoneLink.c`.

Media Callbacks

This module includes implementations for media callbacks invoked by the Multimedia Terminal Framework. For each media callback, this module processes the media parameters received in the callback, builds an EPP media message and sends it out to the GUI application through the Media Control library and the EPP client. See the file `MtfMediaCallback.c`.

Media Control

This library is used for sending media messages from `MtfSipSample` to the GUI application, indicating to the GUI application to perform media operations. This library converts SDP objects to textual messages that are sent over EPP. This library is part of the Multimedia Terminal Framework sample application. You do not need this library in your application if your application does not use EPP.

EPP Client

This module listens for events and Register/Unregister messages received from the GUI application. When an event message arrives, it passes the event to `IppSipSamplePhoneLink.c`, which parses the message and sends the event to the Multimedia Terminal Framework. For more information, see the [EPP](#) section.

EPP Client Endpoint

This module is used for sending signal messages to the GUI application by `IppSipSamplePhoneLink.c` and for sending media messages by `MtfMediaCallbacks.c`. For more information, see the [EPP](#) section.

GUI Application

For information, see the [GUI Application](#) section.

MTFSIPSAMPLE OBJECTS

This section describes the objects in MtfSipSample and the relations between them, as illustrated in [Figure 5-4](#) and [Figure 5-5](#).

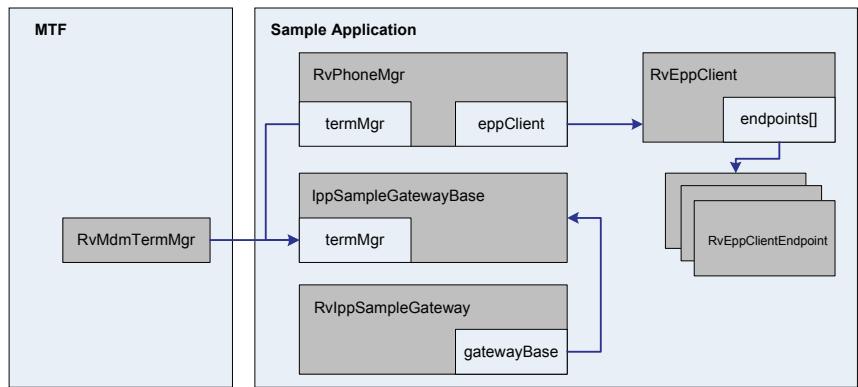


Figure 5-4 MtfSipSample objects

The following objects are described in [Figure 5-4](#):

RvMdmTermMgr

This object is the MTF Manager and is part of the Multimedia Terminal Framework. The pointer to this object is stored in the Multimedia Terminal Framework, while the object is stored in IppSampleGatewayBase.

IppSampleGatewayBase

This object stores the MTF Manager object, RvMdmTermMgr, which is obtained during initialization. It also includes all configuration parameters and the database of the Multimedia Terminal Framework, which is protocol independent.

RvlppSampleGateway

This object stores all protocol-specific information of MtfSipSample, as well as a pointer to IppSampleGatewayBase.

RvPhoneMgr

This object includes pointers to the MTF Manager, RvMdmTermMgr, and to the EPP client. It also contains callback implementations of the EPP client that are invoked whenever an EPP message arrives from the GUI application (rvPhoneMgrOnEvent, rvPhoneMgrOnRegister, rvPhoneMgrOnUnregister).

RvEppClient

This object represents the EPP client. It contains a list of all RvEppClientEndpoints and is responsible for receiving EPP register/unregister and event messages from the GUI application.

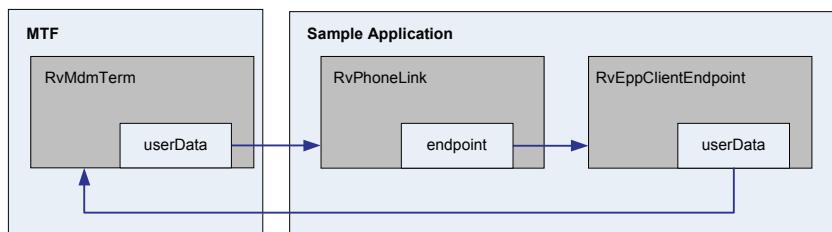


Figure 5-5 *MtfSipSample* termination objects

The following objects are described in [Figure 5-5](#):

RvMdmTerm

This object is part of the Multimedia Terminal Framework and represents a physical termination. When the sample application registers a physical termination to the Multimedia Terminal Framework, it receives a pointer to this object.

RvPhoneLink

This object is associated with the physical terminal object of the Multimedia Terminal Framework, RvMdmTerm. Hence, the sample application sets a pointer to RvPhoneLink as user data of the termination. It is constructed when a register message arrives from the GUI application, and is destructed when an unregister message arrives from the GUI application.

In general, RvPhoneLink will hold any information of the application that is associated with the Multimedia Terminal Framework physical terminal. In this version, RvPhoneLink contains only a pointer to RvEppClientEndpoint.

When the sample application handles a register message received from the GUI application, it constructs a new RvPhoneLink object and registers a physical termination to the Multimedia Terminal Framework, while passing a pointer to the new RvPhoneLink object as user data.

When integrating the Multimedia Terminal Framework, you may configure your own application data instead of setting a pointer to RvPhoneLink, and remove the EPP client.

RvEppClientEndpoint

This object is responsible for sending EPP signal messages from MtfSipSample to the GUI application. For more information, see the section [EPP](#).

5.7 SAMPLE CODE

This section refers to the sample code in the Multimedia Terminal Framework sample application. It includes pointers to the relevant Multimedia Terminal Framework callbacks and integration steps; it does not offer full documentation of the code.

This section may be of help to you if you are using the Multimedia Terminal Framework sample application as a basis for your application. In this case, it is strongly recommended that you also follow the guidelines in the section [Integrating MTF without EPP](#).

MtfSipSample includes two directories:

- Common directory—this directory includes all protocol-independent modules
- SIP directory—this directory includes all SIP-specific modules

The tables below contain general explanations about the files of each directory as well as references to the relevant Multimedia Terminal Framework integration steps.

Table 5-19 Pointers in the Common directory

#	Pointers in Sample Application	Usage	Reference in MTF intergration steps
IppSampleGatewayBase.c			
1	Utility functions Sample application utility functions for loading parameters from the configuration file (<i>sipphone.cfg</i>).	You may use these functions only if you use <i>sipphone.cfg</i> as your configuration file.	N/A
2	Call Forward callbacks This file includes implementations of Call Forward callbacks: <ul style="list-style-type: none">• rvIppSampleCfwActivateCompletedCB()• rvIppSampleCfwDeactivateCompletedCB()	You may change these implementations according to the requirements of your application.	See Call Forward section for details on Call Forward callbacks.
3	Two MTF media callbacks This file includes implementations of two MTF media callbacks: <ul style="list-style-type: none">• rvIppSampleSelectTermination()• rvIppSampleGatewayDeleteEphTerm()	You may change these implementations according to the requirements of your application.	See chapter Building Your Application , Step 4: Integrating Media for more information about the following callbacks: <ul style="list-style-type: none">• RvMdmTermMgrSelectTerminationCB()• RvMdmTermMgrDeleteEphTermCB()

Table 5-19 Pointers in the Common directory

4	Set user data This function rvIppSampleGatewayBaseRegisterMdmCallbacks() sets RvIppSampleGatewayBase as the sample application user data.	Set the user data of your application in MTF. The user data will be associated with RvTermMgr.	See the MTF Reference Guide for more information about this API: • rvMdmTermMgrSetUserData()
IppSampleMediaNegotiate.c			
5	Media negotiation This file is responsible for media negotiation during call establishment or while a call is connected. The main function is MtfMediaNegotiate().	You may change this function according to the requirements of your application.	See chapter Building Your Application, Step 4: Integrating Media for more information about the following callbacks: • RvMdmTermCreateMediaCB() • RvMdmTermModifyMediaCB() Your application should perform media negotiation in these two callbacks.
IppSamplePhoneLink.c			
6	Reporting events This file includes a function that is an implementation of the EPP callback rvPhoneMgrOnEvent(). The callback is invoked whenever an event message is received from the GUI application. The callback implementation converts the EPP message to an MTF event and sends it to the Multimedia Terminal Framework.	You may change this function to the one that is invoked by your application when the user presses a key.	See chapter Building Your Application, Step 3: Reporting Events for more information about rvMdmTermProcessEvent()
7	Signal callbacks This file includes implementations for MTF signal callbacks: • rvPhoneMgrStartSignalIppCB() • rvPhoneMgrStopSignalIppCB() • rvPhoneMgrStartSignalAnlgCB() • rvPhoneMgrStopSignalAnlgB(). The implementations convert an MTF signal to EPP signal message and sends it to GUI application.	You may change implementations of these functions to call directly to your terminal for playing tones, displaying text or any other change of the user interface required by the signal callbacks.	See chapter Building Your Application, Step 2: Playing Signals for more information about signal callbacks.

Sample Code

Table 5-19 Pointers in the Common directory

8	Unregister termination This file includes a function that is an implementation of the EPP callback <code>rvPhoneMgrOnUnregister()</code> . The callback is invoked whenever an Unregister message is received from the GUI application. The callback implementation unregisters the termination from the MTF.	You may change this function to the one that is invoked by your application when the termination is deactivated.	See chapter Shutting Down: 1. Unregister Terminations .
---	---	--	---

IppSampleRtpLink.c

9	RvRtpLink object This file implements the RvRtpLink object, which is an internal sample application object. RvrtpLink is associated with the RTP termination in the Multimedia Terminal Framework. RvRtpLink represents the RTP session. Its pointer will be received as user data in media callbacks called by the Multimedia Terminal Framework.	You may replace RvRtpLink with the object in your application that represents an RTP session.	See chapter Building Your Application, Step 4: Integrating Media .
---	--	---	--

IppSampleUserCbkcs.c

10	MDM extensibility callbacks This file shows examples of implementations of MDM extensibility callbacks.	You may change these implementations according to the requirements of your application.	As described in the chapter Extensibility, MDM Control .
----	---	---	--

IppSampleVideoUserCbkcs.c

11	Fast Update callbacks This module implements Video callbacks used for Fast Update.	You may change these implementations according to the requirements of your application.
----	--	---

MtfMediaCallbacks.c

12	Media callbacks This file implements Multimedia Terminal Framework media callbacks.	You may change these implementations according to the requirements of your application.	See chapter Building Your Application, Step 4: Integrating Media .
----	---	---	--

Table 5-19 Pointers in the Common directory**MtfMediaCallbacksHelper.c****13 Media utility functions**

This file provides utility functions for the MtfMediaCallbacks module (building messages, etc.)

These utility functions are used for the MTF sample application specific implementation for media processing. You may use them as an example for processing SDP messages.

N/A

Table 5-20 Pointers in the SIP directory

#	Pointers in Sample Application	Usage	Reference in MTF
IppSampleSipMain.c			
1	The Main functions This file includes the main functions of MtfSipSample: vxMain() for vxWorks, WinMain() for Windows Mobile, and main() for all other operating systems.	You may change this file according to the requirements of your application.	N/A
IppSampleSipGateway.c			
2	MTF initialization & configuration This file includes initialization and configuration of the Multimedia Terminal Framework. See function: rvIppSipSampleGatewayConstruct()	You may change this file according to the requirements of your application.	See chapter Building Your Application, Step 1: Initialization .
3	MTF destruction This file includes destruction of the Multimedia Terminal Framework. See function rvIppSipSampleGatewayDestruct()	You may change this file according to the requirements of your application.	See chapter Shutting Down .
IppSampleSipMWI.c			
4	Message Waiting Indication(MWI) This file demonstrates how Message Waiting Indication can be implemented above the Multimedia Terminal Framework.	You may change this file according to the requirements of your application.	See section Message Waiting Indication .

Sample Code

Table 5-20 Pointers in the SIP directory

IppSampleSipPhoneLink.c

5 Register termination This file includes an implementation of an EPP callback: rvPhoneMgrOnRegister(). The callback is invoked whenever a Register message is received from the GUI application. The callback implementation registers the termination to the MTF.	<ul style="list-style-type: none">• You may change this function to the one that is invoked by your application when termination is activated.• Set your user data as the last parameter in rvMdmTermMgrRegisterPhysicalTerminationAsync(). This user data will be passed to your application when media callbacks are called.	See chapter Building Your Application, Registering Terminations .
---	---	---

IppSampleSipTls.c

6 TLS configuration This file includes the configuration required for the TLS feature.	You may change this file according to the requirements of your application.	See chapter Features, SIP Features: TLS .
--	---	---

IppSampleSipTlsCibks.c

7 TLS callbacks and openssl. This file includes: <ul style="list-style-type: none">• TLS callback implementations• Integration with openssl	You may change this file according to the requirements of your application.	See chapter Features, SIP Features: TLS .
---	---	---

IppSampleSipUserCibks.c

8 SIP extensibility callbacks This file shows examples of implementations of SIP extensibility callbacks.	You may change these implementations according to the requirements of your application.	See chapter Extensibility, SIP Control .
---	---	--

IppSampleSipUtils.c

9 Utility functions This file includes internal utility functions used by MtfSipSample.	You may use these utility functions if you find them useful for your application.	N/A
---	---	-----

5.8 EPP

OVERVIEW

EPP is a proprietary protocol used by the Multimedia Terminal Framework sample application to communicate with the GUI application. The purpose of this protocol is to enable running the Multimedia Terminal Framework sample application on a non-Windows operating system. EPP messages are text based, sent over UDP from the Multimedia Terminal Framework sample application to GUI application and vice versa.

Note As explained, the sole purpose of EPP is to enable MtfsipSample and the GUI application to run on separate machines. If your application and your terminal run on the same machine (in the same process), there is no need to use EPP. For more information, see [Integrating MTF without EPP](#).

The messages are sent and received by a dedicated EPP thread. As illustrated by Figure 5-3, there are three types of messages:

- **Events and Register messages**—sent from the GUI to MtfsipSample to indicate that the user has pressed a key, or that a termination was registered or unregistered. No reply is sent for events and Register messages.
- **Signals messages**—sent from MtfsipSample to the GUI to indicate that the user interface should be updated (play tone, change text display, etc.). Signal messages are sent in a blocking mode until a reply is received from the GUI: Ack for success, or Nak for failure.
- **Media messages**—sent from MtfsipSample to the GUI to perform media operations (create media streams, etc.)

EPP OBJECTS

EPP Client

The EPP client object listens for events and Register/Unregister messages received from the GUI application:

- For each Register message that arrives from the GUI application, a termination is registered to the Multimedia Terminal Framework, and an EppClientEndpoint object is constructed for sending signals to that terminal. The Register message includes the IP address of the GUI application and the

port number of the terminal to which the signal messages will be sent. The Register message is sent to IppSampleSipPhoneLink.c.

- For each Unregister message, the termination is unregistered from the Multimedia Terminal Framework and an EppClientEndpoint object is destructed. The Register message is sent to IppSampleSipPhoneLink.c.
- For each event message that arrives, an event is sent to IppSamplePhoneLink.c.

The EPP client is constructed during initialization and destructed during shutdown. During initialization, this module opens a UDP port for receiving events and Register messages from the GUI application.

EPP Client Endpoint

The EppClientEndpoint object sends signal and media messages to the GUI application when signal and media callbacks are invoked by the Multimedia Terminal Framework. For each Register message that arrives from the GUI application, an EppClientEndpoint object is constructed. When an Unregister message arrives from the GUI application, the EppClientEndpoint object is destructed. Hence, one EppClientEndpoint object will exist per registered termination:

- An IPPhone/Videophone application will have one EppClientEndpoint object for the UI, and one EppClientEndpoint object for each audio and video termination. All objects will have the same port number. (Not supported in this version.)
- An IAD application will have one EppClientEndpoint for each analog line, each with a different port number.

Every time an EppClientEndpoint object is constructed, it opens one UDP port for sending signals and media messages.

EPP SOCKETS

The EPP library opens two UDP sockets for communication with the GUI application:

- **EPP Client socket**—this socket is opened by the EPP Client for receiving events and Register messages. The port number is configured in two parameters. Both parameters should be configured to the same value:

- The parameter EppPort—configured in MtfSipSample (*sippHONE.cfg*)
- The parameter SERVER_PORT—configured in the GUI application (*IPP.xml*).
Default value: 3044
- **EPP Client Endpoint socket**—this socket is opened by the EPP Client Endpoint for sending signals and media messages and for receiving replies for signal messages. The socket binds to the IP address and port number that are included in the incoming Register message. The port number is configured in GUI application (*IPP.xml*) as parameter CLIENT_PORT.
Default value: 1255

Note In the IAD application, the number of EPP Client Endpoint sockets will be the same as the number of analog lines. (Not supported in this version.)

CALL FLOW EXAMPLE

Figure 5-6 illustrates the call flow between the GUI application and the Multimedia Terminal Framework sample application through EPP. `rvEppClientWrapper()` is an EPP callback that is called every time an EPP message arrives from the GUI. This callback builds a Multimedia Terminal Framework event and sends it to the Multimedia Terminal Framework by calling API `rvMdmTermProcessEvent()`. `rvPhoneMgrStartSignalIppCB()` is an implementation of the Multimedia Terminal Framework callback `RvMdmTermStartSignalCB()`, which is called when signals are sent from the Multimedia Terminal Framework to the user application.

See the tables below for lists of EPP events and signals.

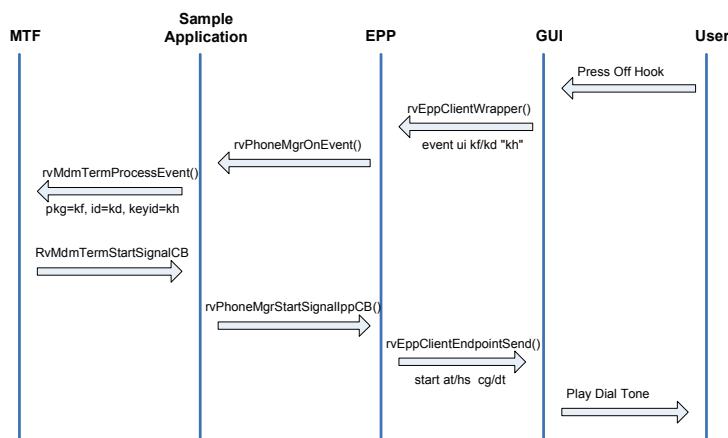


Figure 5-6 EPP Call Flow Example

EPP EVENTS TABLE

Table 5-21 lists EPP messages that will be sent from the GUI application to the Multimedia Terminal Framework sample application whenever the user presses a GUI application key. Two EPP messages are sent for each key pressed: One when the key is pressed down (key down) and one when the key is released (key up). The key up message is sent with the duration of the pressing of the key.

The EPP Message column presents the message to be processed by the Multimedia Terminal Framework. Messages that do not appear in the table are ignored by the Multimedia Terminal Framework.

Example

- When the user presses digit 1, the Multimedia Terminal Framework processes both EPP messages:
 - event ui kp/kd "k1"—the Multimedia Terminal Framework will send a signal to notify the GUI application to start DTMF tone (see [EPP Signals Table](#)).
 - event ui kp/ku "k1","100"—the Multimedia Terminal Framework will send a signal to notify the GUI application to stop the DTMF signal (see [EPP Signals Table](#)).

- When the user presses the Transfer key, only one EPP message will be processed and the second one will be ignored:
event ui kf/kd "kt"—the Multimedia Terminal Framework will process the Transfer event.
ui kf/ku "kt", "100"—the Multimedia Terminal Framework will ignore this message.

When the GUI sends an EPP message to the Multimedia Terminal Framework sample application, the EPP callback `rvEppClientWrapper()` will be called in the Multimedia Terminal Framework sample.

Table 5-21 EPP messages from GUI application to MTF sample application

#	Key Pressed	EPP Message
1	Off Hook	event ui kf/kd "kh"
2	On Hook	event ui kf/ku "kh", "100"
3	Digit 1	event ui kp/kd "k1" event ui kp/ku "k1", "100"
4	Digit 2	event ui kp/kd "k2" event ui kp/ku "k2", "100"
5	Digit 3	event ui kp/kd "k3" event ui kp/ku "k3", "100"
6	Digit 4	event ui kp/kd "k4" event ui kp/ku "k4", "100"
7	Digit 5	event ui kp/kd "k5" event ui kp/ku "k5", "100"
8	Digit 6	event ui kp/kd "k6" event ui kp/ku "k6", "100"
9	Digit 7	event ui kp/kd "k7" event ui kp/ku "k7", "100"
10	Digit 8	event ui kp/kd "k8" event ui kp/ku "k8", "100"

Table 5-21 EPP messages from GUI application to MTF sample application

11	Digit 9	event ui kp/kd "k9" event ui kp/ku "k9", "100"
12	Digit 0	event ui kp/kd "k0" event ui kp/ku "k0", "100"
13	Digit #	event ui kp/kd "ko" event ui kp/ku "ko", "100"
14	Digit *	event ui kp/kd "ks" event ui kp/ku "ks", "100"
15	Line 1	event ui kf/kd "l001"
16	Line 2	event ui kf/kd "l002"
17	Transfer	event ui kf/kd "kt"
18	Conference	event ui kf/kd "kc"
19	Hold	event ui kf/kd "kl"
20	Mute	event ui kf/kd "mu"
21	Speaker	event ui kf/kd "hf"
22	Headset	event ui kf/kd "ht"
23	Registration	register <term id> ipphone <GUI ip address> <GUI port>
24	Unregistration Headset	unregister <term id> ipphone

EPP SIGNALS TABLE

Table 5-22 lists EPP messages sent from the Multimedia Terminal Framework sample application to the GUI application whenever a user indication should be applied on the GUI application (for example: play tone, set indicator, etc.).

The EPP Message column lists the EPP messages to be sent from the Multimedia Terminal Framework sample application, while the GUI application Action column lists the corresponding actions to be performed by the GUI application. When Start messages should be sent, the callback

`rvPhoneMgrStartSignalIppCB()` will be called in the Multimedia Terminal Framework sample application. When Stop messages should be sent, the callback `rvPhoneMgrStopSignalIppCB()` will be called.

Table 5-22 EPP messages from MTF sample application to GUI application

#	EPP Message	GUI application action
1	start ui dis/cld	Clear text display on screen
2	start ui dis/di "Hello","0","0"	Display "Hello" on screen
3	start ui ind/is "l001","on"	Set Line 1 indicator on
4	start ui ind/is "l001","off"	Set Line 1 indicator off
5	start ui ind/is "l001","blink"	Set Line 1 indicator blink
6	start ui ind/is "l002","on"	Set Line 2 indicator on
7	start ui ind/is "l002","off"	Set Line 2 indicator off
8	start ui ind/is "l002","blink"	Set Line 2 indicator blink
9	start ui ind/is "il","on"	Set Hold indicator on
10	start ui ind/is "il","off"	Set Hold indicator off
11	start ui ind/is "mu","on"	Set Mute indicator on
12	start ui ind/is "mu","off"	Set Mute indicator off
13	start ui ind/is "hf","on"	Set Speaker indicator on
14	start ui ind/is "hf","off"	Set Speaker indicator off
15	start ui ind/is "mwi","blink"	Set MWI indicator blink
16	start ui ind/is "mwi","off"	Set MWI indicator off
17	start at/hs cg/dt	Play dialtone.wav
18	stop at/hs cg/dt	Stop dialtone.wav
19	start at/hs cg/bt	Play busy.wav

Table 5-22 EPP messages from MTF sample application to GUI application

20	stop at/hs cg/bt	Stop busy.wav
21	start at/hs cg/wt	Play fastbusy.wav
22	stop at/hs cg/wt	Stop fastbusy.wav
23	start at/hs cg/cw	Play CallWaiting.wav
24	stop at/hs cg/cw	Stop CallWaiting.wav
25	start at/hs cg/cr	Play CallWaitee.wav
26	stop at/hs cg/cr	Stop CallWaitee.wav
27	start at/hs cg/rt	Play ringback.wav
28	stop at/hs cg/rt	Stop ringback.wav
29	start ui ind/is "ir","on"	Play ring.wav
30	start ui ind/is "ir","off"	Stop ring.wav
31	start at/hs dg/k0	Play DTMF_0.wav
32	stop at/hs dg/k0	Stop DTMF_0.wav
33	start at/hs dg/k1	Play DTMF_1.wav
34	stop at/hs dg/k1	Stop DTMF_1.wav
35	start at/hs dg/k2	Play DTMF_2.wav
36	stop at/hs dg/k2	Stop DTMF_2.wav
37	start at/hs dg/k3	Play DTMF_3.wav
38	stop at/hs dg/k3	Stop DTMF_3.wav
39	start at/hs dg/k4	Play DTMF_4.wav
40	stop at/hs dg/k4	Stop DTMF_4.wav
41	start at/hs dg/k5	Play DTMF_5.wav

Table 5-22 EPP messages from MTF sample application to GUI application

42	stop at/hs dg/k5	Stop DTMF_5.wav
43	start at/hs dg/k6	Play DTMF_6.wav
44	stop at/hs dg/k6	Stop DTMF_6.wav
45	start at/hs dg/k7	Play DTMF_7.wav
46	stop at/hs dg/k7	Stop DTMF_7.wav
47	start at/hs dg/k8	Play DTMF_8.wav
48	stop at/hs dg/k8	Stop DTMF_8.wav
49	start at/hs dg/k9	Play DTMF_9.wav
50	stop at/hs dg/k9	Stop DTMF_9.wav
51	start at/hs dg/ks	Play DTMF_Star.wav
52	stop at/hs dg/ks	Stop DTMF_Star.wav
53	start at/hs dg/ko	Play DTMF_ps.wav
54	stop at/hs dg/ko	Stop DTMF_ps.wav

5.9 INTEGRATING MTF WITHOUT EPP

OVERVIEW

This section presents guidelines for removing EPP from the Multimedia Terminal Framework sample application. As explained above, EPP is a proprietary protocol that enables running the Multimedia Terminal Framework sample application and the GUI application on separate machines. If this separation is not required by your application, it is recommended to integrate the Multimedia Terminal Framework without EPP and avoid the unnecessary exchange of UDP messages. The next step will be to integrate your application with your GUI or terminal.

Note After EPP is removed from MtfSipSample as described below, it will no longer run with the GUI application.

GUIDELINES

To integrate the Multimedia Terminal Framework with your application without using EPP, you should modify the following functions in MtfSipSample:

EPP Library

This library includes all EPP-related functions. The entire library can be removed, together with any references to it in MtfSipSample.

Function rvPhoneMgrOnEvent()

The function rvPhoneMgrOnEvent() is a callback implementation of the EPP client. This callback is invoked by the EPP client whenever an EPP event message arrives from the GUI, meaning that a key was pressed in the GUI. This function parses the EPP message and calls the Multimedia Terminal Framework API rvMdmTermProcessEvent() to send the event to the Multimedia Terminal Framework. You should replace this callback with your own callback, event or other that will be responsible for sending the event to the Multimedia Terminal Framework whenever a key is pressed by the user in your terminal endpoint. For more information about reporting events to the Multimedia Terminal Framework, see the chapter [Building Your Application, Step 3: Reporting Events](#).

Function rvPhoneMgrOnRegister()

The function `rvPhoneMgrOnRegister()` is a callback implementation of the EPP client. This callback is invoked by the EPP client whenever an EPP Register message arrives from the GUI, meaning that the user registered the GUI to the Multimedia Terminal Framework sample application (choose Register on the right click menu). This function parses the EPP message and calls the Multimedia Terminal Framework API

`rvMdmTermMgrRegisterPhysicalTerminationAsync()` to register the termination to the Multimedia Terminal Framework. You should replace this callback with your own callback, event or other that will be responsible for registering your terminations to the Multimedia Terminal Framework whenever your terminations become active. For more information about registering terminations to the Multimedia Terminal Framework, see the chapter [Building Your Application, Registering Terminations](#).

Function rvPhoneMgrOnUnregister()

The function `rvPhoneMgrOnUnregister()` is a callback implementation of the EPP client. This callback is invoked by the EPP client whenever an EPP Unregister message arrives from the GUI, meaning that the user unregistered the GUI to the Multimedia Terminal Framework sample application (choose Unregister on the right click menu). This function parses the EPP message and calls the Multimedia Terminal Framework API

`rvMdmTermMgrUnregisterTerminationAsync()` to unregister the termination from the Multimedia Terminal Framework. You should replace this callback with your own callback, event or other that will be responsible for registering your terminations to the Multimedia Terminal Framework whenever your terminations become active. For more information about registering terminations to the Multimedia Terminal Framework, see the chapter [Building Your Application, Registering Terminations](#).

Functions rvPhoneMgrStartSignalXXXCB()

The functions `rvPhoneMgrStartSignalIppCB()` and `rvPhoneMgrStartSignalAnlgCB()` are callback implementations of the Multimedia Terminal Framework callback `RvMdmTermStartSignalCB()`. This callback is invoked by the Multimedia Terminal Framework whenever the user application is required to apply a signal (play a tone, turn on an indicator, etc.) on the terminal. This function builds an EPP signal message and calls the EPP function to send the EPP message to the GUI. You should replace this function

with your own code that will be responsible for applying the signal on your termination. For more information about implementing signal callbacks, see the chapter [Building Your Application, Step 2: Playing Signals](#).

Functions rvPhoneMgrStartSignalXXXCB()

The functions `rvPhoneMgrStopSignalIppCB()` and `rvPhoneMgrStopSignalAnlgCB()` are callback implementations of the Multimedia Terminal Framework callback `RvMdmTermStopSignalCB()`. This callback is invoked by the Multimedia Terminal Framework whenever the user application is required to stop a signal (stop playing a tone, turn off an indicator, etc.) on the terminal. This function builds an EPP signal message and calls the EPP function to send the EPP message to the GUI. You should replace this function with your own code that will be responsible for stopping the playing of the signal on your termination. For more information about implementing signal callbacks, see the chapter [Building Your Application, Step 2: Playing Signals](#).

File MtfMediaCallbacks.c

This file contains callback implementations for Multimedia Terminal Framework media callbacks. The callbacks process media parameters, negotiate media, and then send EPP media message to the GUI. These functions begins with `mcHelperXXX()` (for example: `mcHelperOpenDev()`, `mcHelperConnectRtpToDev()`, etc.) You should replace the part that sends EPP media messages with your own code that is responsible for stopping the playing of the signal on your termination. For more information about implementing media callbacks, see the chapter [Building Your Application, Step 4: Integrating Media](#).

5.10 IPV6

OVERVIEW

When the Multimedia Terminal Framework is compiled with IPV6 support, it can run with either IPV6 or IPV4, but not with both. Therefore, configuration may include IP addresses in either IPV6 or in IPV4 (i.e., either all addresses are configured to IPV4, or all are configured to IPV6). For more information, see the chapter [Features](#), section [IPV6](#).

Note When a Terminal Framework sample application is running with IPV6 and it is configured with IPV6 addresses, only the signaling messages will be sent via IPV6. RTP and EPP messages (internal messages between the sample and the GUI application) are sent via IPV4.

DESCRIPTION



To run the Multimedia Terminal Framework sample via IPV6

1. Compile the Multimedia Terminal Framework sample (in Windows: rvSipPhone.dsw) with the compilation flag RV_IPV6, enabling IPV6 support.
2. Configure the parameters EppIp and EppPort with an IPV4 address for internal communication between the sample and the GUI application.

5.11 MESSAGE WAITING INDICATION

OVERVIEW

The Multimedia Terminal Framework provides a basic sample for implementing Message Waiting Indication (MWI) in the user application (over the Multimedia Terminal Framework layer).



To run the Multimedia Terminal Framework sample with MWI support

- ◎ Compile the Multimedia Terminal Framework sample (in Windows: *rvSipPhone.dsw*) with the compilation flag: `SAMPLE_MWI`.

For sample code, see the files *IppSampleSipMWI.c* and *IppSampleSipMWI.h*.

DESCRIPTION

The MWI sample includes the following:

- **Configuration**

Configuration consists of two groups of parameters:

- **Sample configuration**—see the structure `RvIppSampleSipMwiCfg`:
 - ◆ StackHandle—handle to the SIP Stack
 - ◆ LocalAddress—local IP address
 - ◆ RegistrarAddress—address of the Registrar
 - ◆ RegistrarPort—port of the Registrar
 - ◆ stackUdpPort—port to listen on Stack messages
 - ◆ uiAlias—termination ID to register to Multimedia Terminal Framework
 - ◆ subsServerName—IP address or domain name of the Server to which Subscriptions will be sent.

- **Stack parameters**—see the function `rvIppSampleSipMwiSetStackCfg()`. SIP Stack parameters can be configured in the implementation of the callback `RvIppSipStackConfigCB()`. The following parameters should be configured:
 - ◆ MaxSubscriptions—Set the value of this parameter to the maximum number of simultaneous subscriptions required by your application. Default value: 5.

- ◆ SubsAutoRefresh—Indicates whether the SIP Stack will send a Refresh automatically. Default value: False.

■ Initialization

See the function RvIppSampleSipMwiInit().

The Initialization function registers Stack callbacks, allocates memory, stores configuration parameters, etc. It is called after the SIP Stack is initialized (after (rvIppSipStackInitialize() is called). The sample registers the following Stack events:

- RvSipSubsCreatedEv—exchanges handles with the Stack. In this version, the application handle is set to NULL.
- RvSipSubsNotifyEv—handles incoming notifications (see below under *Handling of incoming notifications*).
- RvSipSubsExpirationAlertEv—sends a refresh to the subscription.
- RvSipSubsStateChangedEv—not functional in this version.
- RvSipSubsMsgReceivedEv—not functional in this version.
- RvSipSubsMsgToSendEv—not functional in this version.

■ Registration of user signals

A new package and signals are registered to the Multimedia Terminal Framework to be used for notifications on the phone when messages arrive. This is done by calling:

- RvMdmTermMgrRegisterUserSignal()
- RvMdmTermMgrSetUserPackage()

■ Subscription

See the function rvIppSampleSipMwiRegisterMdmTerm().

A subscription message is sent to the Server for every registered termination. The first subscription messages is sent when the termination registers to the Multimedia Terminal Framework. Additional refresh messages are sent whenever the Stack event RvSipSubsExpirationAlertEv() is invoked.

■ **Handling of incoming notifications**

See the callback RvSipSubsNotifyEv, state = RVSIP_SUBS_NOTIFY_STATUS_REQUEST_RCVD.

Handle a received Notify message as follows:

- Process the message.

See the function handleNotification(). Check the Messages-Waiting: text in the message body:

- ◆ If Messages-Waiting:yes is included, set indicators on the phone to indicate that messages are waiting.
- ◆ If Messages-Waiting:no is included, set indicators on the phone to indicate that no messages are waiting.

- Call RvSipNotifyAccept() to send a reply.

UNSOLICITED NOTIFY MESSAGES (WITHOUT SUBSCRIBE)

MWI sample code demonstrates how to handle incoming Notify messages after subscribing to the Server. To receive unsolicited Notify messages (i.e., without Subscribing), perform the following steps:

1. Register the SIP Stack callback RvSipTransactionStateChangedEv to listen for incoming unsolicited Notify messages (without Subscribe). See the [Extensibility](#) chapter, or the Multimedia Terminal Framework Sample Application for registering Stack events.
2. In this callback, use SIP Stack APIs to retrieve information, and call the API RvSipTransactionRespond() with 200 OK to send a reply.

Example:

```
Void userSipTransactionStateChangedEv(  
    RvSipTranscHandle          hTransc,  
    RvSipTranscOwnerHandle      hTranscOwner,  
    RvSipTransactionState       eState,  
    RvSipTransactionStateChangeReason eReason)  
{  
    RvStatus           rv      = RV_OK;  
    RvSipMsgHandle     hMsg    = NULL;
```

```
char           buffer[2000];
unsigned int   actlen;

if (eState == RVSIP_TRANS_C_STATE_SERVER_GEN_REQUEST_RCVD)
{
    /* check if this is Notify message*/
    RvSipTransactionGetMethodStr(hTransc, 2000, buffer);
    if (strcpy(buffer, "NOTIFY"))
        return;

    /* Get message */
    rv = RvSipTransactionGetReceivedMsg(hTransc, &hMsg);
    if (rv != RV_OK)
    {
        return;
    }

    /* Get message body */
    rv = RvSipMsgGetBody(hMsg, buffer, 2000, &actlen);
    if (RV_OK != rv)
    {
        printf("Error getting message body\n");
        return;
    }

    /* Check MWI status */
    if (strstr(buffer, "yes"))
        printf("Messages Waiting = YES\n");
    else if (strstr(buffer, "no"))
        printf("Messages Waiting = NO\n");

    /* Respond with 200 OK */
    RvSipTransactionRespond(hTransc, 200, NULL);
}
```

Message Waiting Indication

6

LOGGING

6.1 WHAT'S IN THIS CHAPTER

The logging facility consists of the Multimedia Terminal Framework Log and the Stack Log. This chapter explains how logging is configured for both of these modules:

- Multimedia Terminal Framework
- Stack Log

6.2 MULTIMEDIA TERMINAL FRAMEWORK

The Multimedia Terminal Framework Log is the logging module that produces output for debugging and monitoring the protocol-independent part of the Multimedia Terminal Framework. The user application should call a sequence of APIs to initialize and configure the Multimedia Terminal Framework Log before it can be used.

For an example of how the Multimedia Terminal Framework Log is initialized, see the Multimedia Terminal Framework sample application file IppSampleSipGateway.c, functions loadIPPLogOptions() and rvIppSampleGatewayConstruct().

INITIALIZING THE LOG

The user application can initialize the Multimedia Terminal Framework Log as follows.



To initialize the Multimedia Terminal Framework Log

- ◎ Call the following function:

```
RvStatus IppLogInit(  
    IppLogSourceFilterElm* ippFilters)
```

This function receives an array of IppLogSourceFilterElm structures as a parameter.

The IppLogSourceFilterElm structure contains two fields:

- **char logSrcName[20]**

This field indicates the module name. Possible values of core modules and Terminal Framework modules are listed below:

Name	Module
SOCKET	Socket
THREAD	Thread
MUTEX	Mutex
SEMA4	Semaphore
LOCK	Locking
MEMORY	Memory
TIMER	Timer
QUEUE	Queue
SELECT	Select
CLOCK	Clock
TIMESTAMP	Timestamp
IPP_USERAPP	User application
IPP_CALLCON	Call Control
IPP_MDM	MDM
IPP_MDMCONT	MDM
IPP_SIPCONT	Control
RPOOL	RPOOL
RA	RA

■ **RvLogMessageType messageMask**

This field indicates the logging mask of the module. Possible values are defined in the *rvinterfacesdefs.h* file, as listed below:

RV_LOGLEVEL_EXCEP	Exception
RV_LOGLEVEL_ERROR	Error
RV_LOGLEVEL_WARNING	Warning
RV_LOGLEVEL_INFO	Info
RV_LOGLEVEL_DEBUG	Debug
RV_LOGLEVEL_ENTER	Enter
RV_LOGLEVEL_LEAVE	Leave
RV_LOGLEVEL_SYNC	Thread synchronization

Example

The following example loads logging configuration values from a static buffer and initializes the logging module:

```
static char configBuf[8192] = {
    "SOCKET=ERROR,EXCEPTION,WARNING,INFO \n\
    THREAD=ERROR,EXCEPTION,WARNING \n\
    MUTEX=ERROR,EXCEPTION,WARNING \n\
    SEMA4=ERROR,EXCEPTION,WARNING \n\
    LOCK=ERROR,EXCEPTION,WARNING \n\
    MEMORY=ERROR,EXCEPTION,WARNING \n\
    TIMER=ERROR,EXCEPTION,WARNING \n\
    QUEUE=ERROR,EXCEPTION,WARNING \n\
    SELECT=ERROR,EXCEPTION,WARNING \n\
    CLOCK=ERROR,EXCEPTION,WARNING \n\
    TIMESTAMP=ERROR,EXCEPTION,WARNING \n\
    IPP_UTIL=ERROR,EXCEPTION,WARNING,INFO \n\
    IPP_USERAPP=ERROR,EXCEPTION,WARNING,INFO \n\
    IPP_CALLCON=ERROR,EXCEPTION,WARNING,INFO \n\
    IPP_MDM=ERROR,EXCEPTION,WARNING,INFO \n\
    IPP_MDMCONT=ERROR,EXCEPTION,WARNING,INFO \n\
```

```
IPP_SIPCONT=ERROR,EXCEPTION,WARNING,INFO \n\
RPOOL=ERROR,EXCEPTION,WARNING \n\
RA=ERROR,EXCEPTION,WARNING \n\"
}

static void initIppLogging(char* configBuf)
{
    IppLogSourceFilterElm logOptions[20];
    char     *ptr = NULL;
    char     line[512];
    char     module[80]="";
    filterString[80]="";
    RvStrTok    t;
    RvInt8      filter;
    int         index = -1;

    /* Load values from buffer*/
    while (rvIppSampleUtilGetNextLine(configBuf, &ptr) !=NULL)
    {
        sscanf(ptr, "%s", line);

        /* Read module name and filter string */
        sscanf(ptr, "%s", line);
        rvStrTokConstruct(&t, "= ", line);
        strcpy(module,rvStrTokGetToken(&t));
        strcpy(filterString,rvStrTokGetToken(&t));
        if (!IppSipUtilFilterToInt(
            IPP_API_TYPE, filterString, &filter))
        {
            printf("failed to determine Filter for the
                   following line\n%s", line);
            break;
        }

        /* Fill in logging structure*/
        strcpy( logOptions[index].logSrcName, module) ;
        logOptions[index].messageMask = filter ;
        index++;
    }

    /* Initialize logging */
}
```

```
IppLogInit(logOptions);  
}
```

DEFAULT OUTPUT

The log output is saved by default in a text file under the name of: IppLog.txt.

CLOSING THE LOG

The user application can close the Multimedia Terminal Framework Log as follows.



To close the Multimedia Terminal Framework Log

- ◎ Call the following function:

```
RvStatus IppLogEnd(void)
```

This function receives no parameters and must be called last, after the Multimedia Terminal Framework has been terminated.

For an example of closing the Multimedia Terminal Framework Log, see the Multimedia Terminal Framework sample application file IppSampleSipGateway.c, function rvIppSampleGatewayDestruct().

Example

```
/* Shut down Multimedia Terminal Framework*/
RvMdmTermMgrStop(termMgr);
rvMdmTermMgrDestruct(termMgr);

/* Close logging*/
IppLogEnd();
```

6.3 STACK LOG

SIP STACK LOG

CONFIGURATION

This section describes the Stack Log. Stack logging uses the Stack API for logging purposes.

This section describes the SIP Stack Log.

Configuring the SIP Stack Log is done in the Terminal Framework configuration. For an example of how to configure the SIP Stack Log, see the file IppSampleSipGateway.c, functions loadSIPLogOptions() and rvIppSampleGatewayConstruct().

Note To print incoming and outgoing SIP messages, the SIP Stack Log should include the following settings: Module = TRANSPORT; Level = INFO.



To configure the SIP Stack Log

1. Fill in the SIP Stack logging structure:

Set values of the log module IDs and filter level in the structure **RvIppSipLogOptions**.

See [Table 6-1](#) for a list of possible Module IDs (defined in RvSipStackTypes.h enumeration RvSipStackModule).

2. Fill in the Terminal Framework configuration structure:

Assign SIP Stack logging options in the structure **RvIppSipPhoneCfg**.

3. Initialize the Multimedia Terminal Framework by calling **rvIppSipStackInitialize()**.

Table 6-1 Frequently Used Source Modules

Source Module Name	Description
CMAPI	Conference Manager API—all the entries/exits to/from the Conference Manager API functions when the application called the Stack.
CMAPICB	Conference Manager Callback—all the entries/exits to/from the Conference Manager API callback functions when the Stack called the application.
PERERR	Packed Encoding Rules (PER) Error Messages. Shows a problem with the decoding of incoming messages or encoding of outgoing messages. This usually implies an error in the user program.
TPKTCHAN	TPKT Channel—all messages sent or received on the TPKT channels (H.245 and Q.931), without modifications that the application might have made using the various send/receive hook functions.
UDPCHAN	UDP Channel—all messages sent or received on the UDP channels (RAS).
TPKTWIRE	All messages sent or received on the TPKT channels (H.245 or Q.931) as they were actually sent/received on the network.
UDPWIRE	All messages sent or received on the UDP channels (RAS) as they were actually sent/received on the network.
WATCHDOG	Stack status messages.

Example

```
static char configBuf[8192] = {
    "callLogFilters=INFO,ERROR,EXCEPTION,WARNING
    transactionLogFilters=INFO,ERROR,EXCEPTION,WARNING
    msgLogFilters=ERROR,EXCEPTION,WARNING
    transportLogFilters=INFO,ERROR,EXCEPTION,WARNING
    parserLogFilters=ERROR,EXCEPTION,WARNING
    stackLogFilters=ERROR,EXCEPTION,WARNING
    msgBuilderLogFilters=ERROR,EXCEPTION,WARNING
```

```
authenticatorLogFilters=ERROR,EXCEPTION,WARNING
regClientLogFilters=ERROR,EXCEPTION,WARNING
subscriptionLogFilters=ERROR,EXCEPTION,WARNING
compartmentLogFilters=ERROR,EXCEPTION,WARNING
transmitterLogFilters=ERROR,EXCEPTION,WARNING
ads_rlistLogFilters=ERROR,EXCEPTION,WARNING
ads_raLogFilters=ERROR,EXCEPTION,WARNING
ads_rpoolLogFilters=ERROR,EXCEPTION,WARNING
ads_hashLogFilters=ERROR,EXCEPTION,WARNING
ads_pqueueLogFilters=ERROR,EXCEPTION,WARNING
core_semaphoreLogFilters=ERROR,EXCEPTION,WARNING
core_mutexLogFilters=ERROR,EXCEPTION,WARNING
core_lockLogFilters=ERROR,EXCEPTION,WARNING
core_memoryLogFilters=ERROR,EXCEPTION,WARNING
core_threadLogFilters=ERROR,EXCEPTION,WARNING
core_queueLogFilters=ERROR,EXCEPTION,WARNING
core_timerLogFilters=ERROR,EXCEPTION,WARNING
core_timestampLogFilters=ERROR,EXCEPTION,WARNING
core_clockLogFilters=ERROR,EXCEPTION,WARNING
core_tmLogFilters=ERROR,EXCEPTION,WARNING
core_socketLogFilters=ERROR,EXCEPTION,WARNING
core_portrangeLogFilters=ERROR,EXCEPTION,WARNING
core_selectLogFilters=ERROR,EXCEPTION,WARNING
core_hostLogFilters=ERROR,EXCEPTION,WARNING
core_tlsLogFilters=ERROR,EXCEPTION,WARNING
core_aresLogFilters=ERROR,EXCEPTION,WARNING
adsLogFilters=ERROR,EXCEPTION,WARNING
coreLogFilters=ERROR,EXCEPTION,WARNING"
}
static void loadSIPLogOptions(char* configBuf)
{
    RvIppSipLogOptions logOptions[40];
    char             * ptr = NULL;
    char             line[512];
    char             module[80] = "", filterString[80] = " ";
    RvStrTok         t;
    RvSipStackModule moduleId;
    RvInt8           filter;
    RvStatus          rv;

    While (rvIppSampleUtilGetNextLine(configBuf, &ptr) != NULL)
```

Stack Log

```
{  
    /* read module name and filter string */  
    sscanf(ptr, "%s", line);  
    rvStrTokConstruct(&t, "= ", line);  
    strcpy(module,rvStrTokGetToken(&t));  
    strcpy(filterString,rvStrTokGetToken(&t));  
  
    if ((rv = IppSipUtilStringToStackModule(module, &moduleId))  
        != rvTrue)  
    {  
        printf("failed to determine Module for the following  
              line\n%s", line);  
        break;  
    }  
  
    if ((rv = IppSipUtilFilterToInt(SIP_API_TYPE, filterString,  
                                    &filter)) != rvTrue)  
    {  
        printf("failed to determine filter for the following  
              line\n%s", line);  
        break;  
    }  
  
    logOptions->filters[logOptions->num].moduleId = moduleId ;  
    logOptions->filters[logOptions->num].filter = filter ;  
    logOptions->num++;  
  
}  
}
```

MODULE IDs

The SIP Stack Log uses various module IDs for monitoring SIP Stack behavior. You can set up the log to analyze any of the source identifiers and specify the level of logging required for each one independently. The main source identifiers that the log analyzes, are:

- RVSIP_CALL
- RVSIP_TRANSACTION
- RVSIP_MESSAGE
- RVSIP_TRANSPORT
- RVSIP_PARSER
- RVSIP_STACK

- RVSIP_MSGBUILDER
- RVSIP_AUTHENTICATOR
- RVSIP_REGCLIENT
- RVSIP_SUBSCRIPTION
- RVSIP_COMPARTMENT
- RVSIP_TRANSMITTER
- RVSIP_ADS_RLIST
- RVSIP_ADS_RA
- RVSIP_ADS_RPOOL
- RVSIP_ADS_HASH
- RVSIP_ADS_PQUEUE
- RVSIP_CORE_SEMAPHORE
- RVSIP_CORE_MUTEX
- RVSIP_CORE_LOCK
- RVSIP_CORE_MEMORY
- RVSIP_CORE_MEMORY
- RVSIP_CORE_THREAD
- RVSIP_CORE_QUEUE
- RVSIP_CORE_TIMER
- RVSIP_CORE_TIMESTAMP
- RVSIP_CORE_CLOCK
- RVSIP_CORE_TM
- RVSIP_CORE_SOCKET
- RVSIP_CORE_PORTRANGE
- RVSIP_CORE_SELECT
- RVSIP_CORE_HOST
- RVSIP_CORE_TLS
- RVSIP_CORE_ARES
- RVSIP_CORE
- RVSIP_ADS

The SIP Stack Log allows you to set any combination of the following filters (defined in the file *RvSipCommonTypes.h*):

- RVSIP_LOG_DEBUG_FILTER

- RVSIP_LOG_INFO_FILTER
- RVSIP_LOG_WARN_FILTER
- RVSIP_LOG_ERROR_FILTER
- RVSIP_LOG_EXCEP_FILTER
- RVSIP_LOG_LOCKDBG
- RVSIP_LOG_ENTER_FILTER
- RVSIP_LOG_LEAVE_FILTER

DEFAULT OUTPUT

By default, the SIP Stack Log prints to the file "SipLog.txt".

ATTACHING A CALLBACK

To use a different logging mechanism, bind a new callback to the SIP Stack Log with Extension APIs.

**To attach a callback**

1. Implement a print function of the type:

```
typedef void (
    RVCALLCONV * RvSipStackPrintLogEntryEv) (
        IN void*           context,
        IN RvSipLogFilters filter,
        IN const RV_CHAR*  *formattedText);
```

2. Implement a user callback of the type:

```
typedef void (*RvIppSipStackConfigCB) (
    RvSipStackCfg* pStackCfg);
```

3. Get the Stack events structure by calling:

```
rvIppSipControlGetStackCallbacks()
```

4. Set the SIP Stack configuration:

```
userSipClbks.stackConfigF = userSipStackConfig;
```

5. Register user callbacks:

```
RvIppSipRegisterExtClbks();
```

Example

```
/*Step 1*/
void RVCALLCONV userSipPrintLogEntryEv(
    IN void*           context,
    IN RvSipLogFilters filter,
    IN const RV_CHAR*  *formattedText)
{
```

```
    printf ("%s\n",formattedText) ;
}
/*Step 2*/
void userSipStackConfig(RvSipStackCfg* stackCfg)
{
    stackCfg->pfnPrintLogEntryEvHandler =
    userSipPrintLogEntryEv;
}

/*Step 3*/
userSipClbks = rvIppSipControlGetStackCallbacks(sipMgrHndl);

/*Step 4*/
userSipClbks.stackConfigF = userSipStackConfig;

/*Step 5*/
rvIppSipRegisterExtClbks(&userSipClbks);
```

Stack Log

7

SHUTTING DOWN

7.1 WHAT'S IN THIS CHAPTER

This chapter explains how to shutdown the Multimedia Terminal Framework:

- **Shutting Down**

7.2 SHUTTING DOWN

The Multimedia Terminal Framework must be shut down in any of the following cases:

- **Before you shut down your application**

Shutting down the Multimedia Terminal Framework allows you to shutdown your application gracefully and release Multimedia Terminal Framework resources.

- **Before you perform a Warm Restart**

Warm Restart allows you to restart the Multimedia Terminal Framework without restarting your application. Before performing a Warm Restart, shutdown the Multimedia Terminal Framework by performing the steps below, and then re-initialize the Multimedia Terminal Framework by performing the steps explained in [Step 1: Initialization](#).

Perform the following steps to shutdown the Multimedia Terminal Framework:

1. Unregister Terminations

Applications that want to shutdown the Multimedia Terminal Framework should first unregister all terminations. Since this process is sometimes asynchronous, the application should wait until the process ends for all terminations before shutting down. Each termination unregisters separately. After a termination

unregisters from the Multimedia Terminal Framework, it can no longer send events or receive calls from the Multimedia Terminal Framework. The user application unregisters a termination by calling one of the following functions:

```
RvBool rvMdmTermMgrUnregisterTermination(
    RvMdmTermMgr*           mgr,
    RvMdmTerm*              term,
    RvMdmServiceChange*     sc)

RvBool rvMdmTermMgrUnregisterTerminationAsync(
    RvMdmTermMgr*           mgr,
    RvMdmTerm*              term,
    RvMdmServiceChange*     sc)
```

Parameters:

mgr—A pointer to the Termination Manager.

term—A pointer to the termination.

sc—This parameter should be set to NULL.

Once one of these functions is called, the unregistration process begins.

The unregistration process is asynchronous. If the termination is registered with a Registrar, the Multimedia Terminal Framework will send an Un-registration message to the Registrar and wait for a reply. The "UnregistrationExpire" configuration parameter defines the timeout (in seconds) to wait for a reply. The process will continue only when the timeout expires or when a reply is received from the Proxy. When either of these circumstances occurs, the Multimedia Terminal Framework will call the following callback to notify the user application that the un-registration process has ended:

```
void RvMdmTermUnregisterTermCompletedCB(
    RvMdmTerm*      term,
    RvMdmError*    mdmError)
```

If the termination is not registered with Registrar, this callback will be called immediately. This callback is called separately for each termination. The Multimedia Terminal Framework then releases all termination resources.

2. Shutting down the Multimedia Terminal Framework

This is done as follows:

- 1. Stop processing events by calling:**

```
void rvMdmTermMgrStop(
    RvMdmTermMgr*          mgr,
    RvMdmServiceChange*     sc)
```

Parameters:

mgr—A pointer to the Termination Manager.

sc—This parameter should be set to NULL.

- 2. Release the Multimedia Terminal Framework resources by calling:**

```
void rvMdmTermMgrDestruct(
    RvMdmTermMgr*          mgr)
```

Parameters:

mgr—A pointer to the Termination Manager.

- 3. Shutdown the system by calling:**

```
rvIppSipSystemEnd()
```

Note The user application should not call IppLogEnd() during the shutdown process, as it is already called by the Multimedia Terminal Framework.

Shutting Down

PART 3: FEATURES

8

FEATURES

8.1 WHAT'S IN THIS CHAPTER

This chapter describes features available in the current version:

- Common Features
- Protocol-Specific Features

8.2 COMMON FEATURES

This section describes the features that are supported in the Multimedia Terminal Framework:

- Caller ID
- Call Waiting
- Hold
- Call Transfer:
 - Attended and Semi-Attended Transfer
 - Blind Transfer
- Three-Way Conference
- Headset/Handsfree
- Mute
- Warm Restart
- IPv6
- SIP Features

CALLER ID

When a new call arrives, the Multimedia Terminal Framework provides the user application with information on the caller. The user application uses this information to display the caller data on the phone screen. This information can

be retrieved by calling Terminal Framework APIs, and is displayed by implementing an extension callback as described below. When an outgoing call is established, the Multimedia Terminal Framework includes information about the user in the outgoing message, based on configuration parameters.

DISPLAY

By default, the Terminal Framework displays the name and address of the caller. Use the following Extension callback to override the default display:

```
void* RvIppMdmDisplayCB(
    RvIppConnectionHandle connHndl,
    RvIppTerminalHandle terminalHndl,
    RvCCTerminalEvent event,
    RvCCEventCause cause,
    void* displayData);
```

In implementing this callback, you can call each of the APIs described below to retrieve the information. For further information on the Display callback, see the [Extensibility](#) chapter.

IMPLEMENTATION

The parameters described below make up the FROM header of the incoming Invite message. The FROM header appears in the following format, based on [RFC 3261](#):

CallerId <sip:CallerName@CallerAddress>
--

INCOMING CALL

When a new Invite message arrives, the information described below becomes available to the user application.

Caller Address

This parameter indicates the caller IP address or the domain name, and can be retrieved by calling the following API:

```
RvBool rvIppMdmConnGetCallerAddress(
    RvIppConnectionHandle connHndl,
    char* callerAddress,
    unsigned int callerAddressLen);
```

Caller Name

This parameter indicates the caller name, and can be retrieved by calling the following API:

```
RvBool rvIppMdmConnGetCallerName(
```

```
RvIppConnectionHandle    connHndl,
char*                  callerName,
unsigned int           callerNameLen);
```

Caller ID

This parameter indicates the caller display name, and can be retrieved by calling the following API:

```
RvBool  rvIppMdmConnGetCallerId(
    RvIppConnectionHandle    connHndl,
    char*                  callerId,
    unsigned int           callerIdLen);
```

Caller Number

This parameter indicates the caller phone number. It is not relevant to SIP Phones and therefore remains empty.

OUTGOING CALL

The following parameters are added to the outgoing Invite message:

Caller Name

This parameter is set to the value of the termination ID as it was configured when the termination was registered to the Multimedia Terminal Framework. For more information, see the [Registering Terminations](#) section in the [Building Your Application](#) chapter.

Caller ID

This parameter is set to the value of the presentation name as it was configured by the user application during initialization, by calling either *rvMdmTermPropertiesSetPresentationInfo()* or *rvMdmTermSetPresentationInfo()*.

Caller Address

This parameter is set as follows:

- If the RegistrarAddress parameter is configured with a domain name, the Caller Address will be set to RegistrarAddress.

- If RegistrarAddress is configured with an IP address or if it is not configured at all, the Caller Address will be set to userDomain.

For more information, see the [Configuration](#) section in the [Building Your Application](#) chapter.

CALL WAITING

Call Waiting informs a served user of an incoming call while the served user is engaged in one or more other calls. The served user then has a choice of either accepting or ignoring the waiting call.

IMPLEMENTATION**OUTGOING CALL**

When a provisional response Queued (182) reply is received to an Invite message, a Call Waiting signal will be played.

INCOMING CALL

When a new Invite is received and there is at least one available line in the termination, a Queued (182) provisional response will be sent as a reply by default. A Ringing (180) reply can be sent if the configuration parameter `callWaitingReply` is set to `RV_REPLY_RINGING`. For more information, see the chapter [Building Your Application](#), section [Configuration](#). If no lines are available, a Busy Here (486) response is sent.

HOLD

Hold allows the served user to temporarily interrupt an existing call with other users. While the call is on hold, the served user may perform other actions such as originate, accept other calls, conference, or use other services without impacting the call on hold. Subsequently, the served user may re-establish (retrieve) his part of the call.

To support this feature:

- Implement the media callback *RvMdmTermModifyMediaCB* as described in the [Building Your Application](#) chapter, section [Step 4: Integrating Media](#).
- Send a Hold event each time the user presses the Hold key:
`pkg = "kf"`
`id = "ku"` when the key is up and `"kd"` when the key is down.
`keyid = "kt"`
- Send a Line event each time the user presses the Line key:
`pkg = "ki"`
`id = "ku"` when the key is up and `"kd"` when the key is down.
`keyid = "l001"` for Line 1, `"l002"` for Line 2, etc.

IMPLEMENTATION

In SIP, Hold is implemented by sending or receiving a Re-Invite message that includes an SDP message indicating to the other party to change the media stream to one-way mode, thus putting the call on Hold.

To unhold the call, an additional Re-Invite is sent or received that includes an SDP message indicating to the other party to change the media stream to two-way mode.

User Steps	Results
1. Setup a call on Line 1	Call may be incoming or outgoing.
2. Press Hold	Media with B is disconnected. Hold indicator is turned on.
3. Press Line 1	Media with B is connected. Hold indicator is turned off.

The Multimedia Terminal Framework supports two implementations:

1. **RFC 3264 Offer/Answer model**—All streams in the SDP indicate a send-only direction, so that no media is played or sent to the remote party. To unhold, the direction is set to send-receive.
2. **RFC 2543**—The session description is the same as in the original invitation (or response), but the "c" destination addresses for the media streams to be put on Hold are set to zero (0.0.0.0). To unhold, the IP addresses are set as in the original invitation.

Note *RFC 2543* is no longer valid but is supported for backward compatibility for incoming messages only.

When a Re-Invite is received, the Terminal Framework checks for both implementations. If at least one exists, the call is put on Hold. When a Re-Invite is sent out, it will include only the *RFC 3264* implementation.

CALL TRANSFER

ATTENDED AND SEMI-ATTENDED TRANSFER

Call Transfer enables the served user (User A, the transferring user) to transfer an existing call with another party (User B, the far-end user) into a new call between User B and a third party (User C, the transferred-to user) selected by User A. There are three types of Transfer: Attended Transfer (Transfer with Consultation), Semi-Attended Transfer, and Blind Transfer. The Multimedia Terminal Framework supports all three types of Transfer.

This section explains the following types of Transfer:

- **Attended Transfer**—Enables the transferring User A to transfer an existing call with User B (primary call) into a new call between User B and a User C selected by User A (secondary call). As an intermediate step ("consultation phase"), User A puts User B on Hold, establishes a separate call with User C ("consultation call"), then drops the consultation call, causes Users B and C to be connected in a new secondary call, and drops out of the call. User A is not involved in the secondary call beyond the consultation phase; the primary call ends.
- **Semi-Attended Transfer**—The process of Semi-Attended Transfer is very similar to that of Attended Transfer described above. The difference between the two is that Semi-Attended Transfer, as opposed to Attended Transfer, enables User A to complete the Transfer before the consultation call is connected. The consultation call will not be connected during the Transfer process. If the call is not connected when User A chooses to complete the Transfer, the consultation call will be canceled and the Transfer process will continue as described in the preceding paragraph.

Note From the point of view of the end user, Semi-Attended Transfer will behave like Blind Transfer if User A completes the Transfer before the consultation call is connected.

The sections below illustrate two ways in which a user can achieve a successful Attended and Semi-Attended Transfer. Note that the difference between Attended and Semi-Attended Transfer is whether the consultation call is connected or not (Step 4 in Transfer on an Active Call; Step 5 in Transfer on a Held Call).

TRANSFER ON AN ACTIVE CALL

User Steps	Results	Comments
1. Setup a call with B	A and B are connected.	Call with B may be incoming or outgoing.
2. Press Transfer	Call with B is put on Hold: --Media with B is disconnected. --Hold indicator is turned on. Dial tone is played on Line 2.	
3. Dial digits	Start to setup a call with C.	If no ringing tone is heard after digits are dialed, the user can go back to the primary call by pressing Line 1.
4. Wait for the call to be answered (Optional)	Call with C is connected.	Attended Transfer
5. Press Transfer	Calls with B and C are disconnected. B and C are connected.	Second Transfer can be pressed at any time after the user completes dialing.
6. Go On-hook	The line is released.	By pressing On-hook or line.

TRANSFER ON A HELD CALL

User Steps	Results	Comments
1. Setup a call with B	A and B are connected.	Call with B may be incoming or outgoing.
2. Press Hold	Media with B is disconnected. Hold indicator is turned on.	
3. Press Line 2	Dial tone is played on Line 2.	
4. Dial digits	Start to setup a call with C.	If no ringing tone is heard after digits are dialed, the user can go back to the primary call by pressing Line 1.
5. Wait for the call to be answered (Optional)	Call with C is connected.	Attended Transfer
6. Press Transfer	Calls with B and C are disconnected. B and C are connected.	
7. Go On-hook	The line is released.	By pressing On-hook or line.

To support this feature:

- Send a Transfer event each time the user presses the Transfer key:

pkg = "kf"

id = "ku" when the key is up and "kd" when the key is down.

keyid = "kt"

IMPLEMENTATION

Transfer is implemented by a REFER message and a Replaces header based on *RFC 3891* (SIP Replaces header) and *draft-ietf-sipping-cc-transfer-03.txt*, sections 6.1, 6.3, 6.6 (figure 12) and 7.

The figures on the pages that follow demonstrate the call flow of Transfer in SIP for parties A, B and C. The left-hand column shows the events received from the user application and the signals sent to the user application. The right-hand column shows the outgoing and incoming network messages.

Common Features

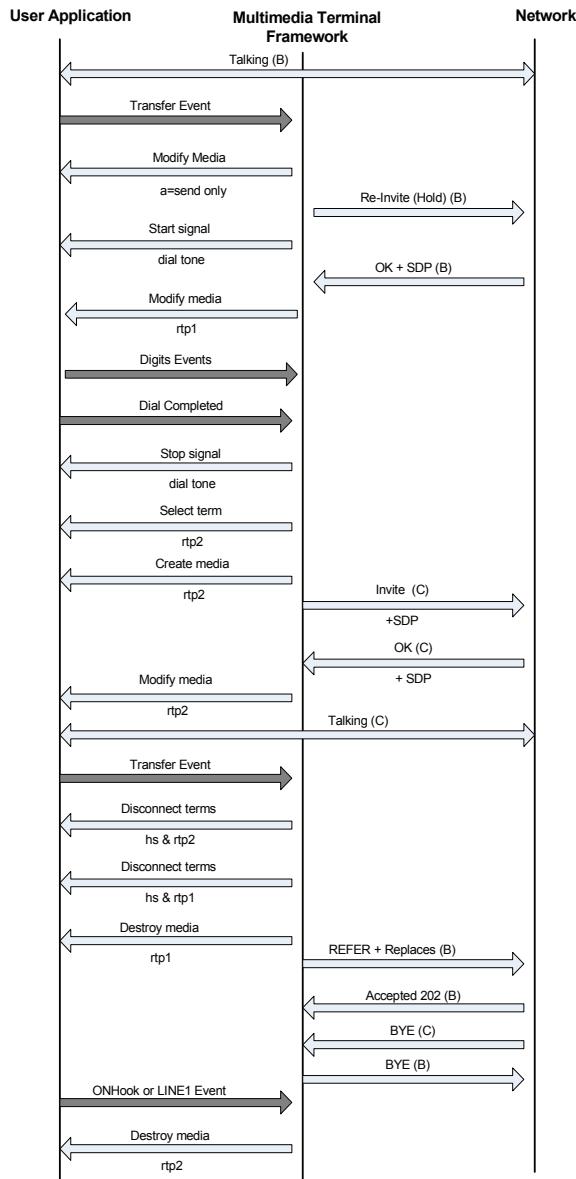


Figure 8-1 Attended Transfer—User A (SIP)

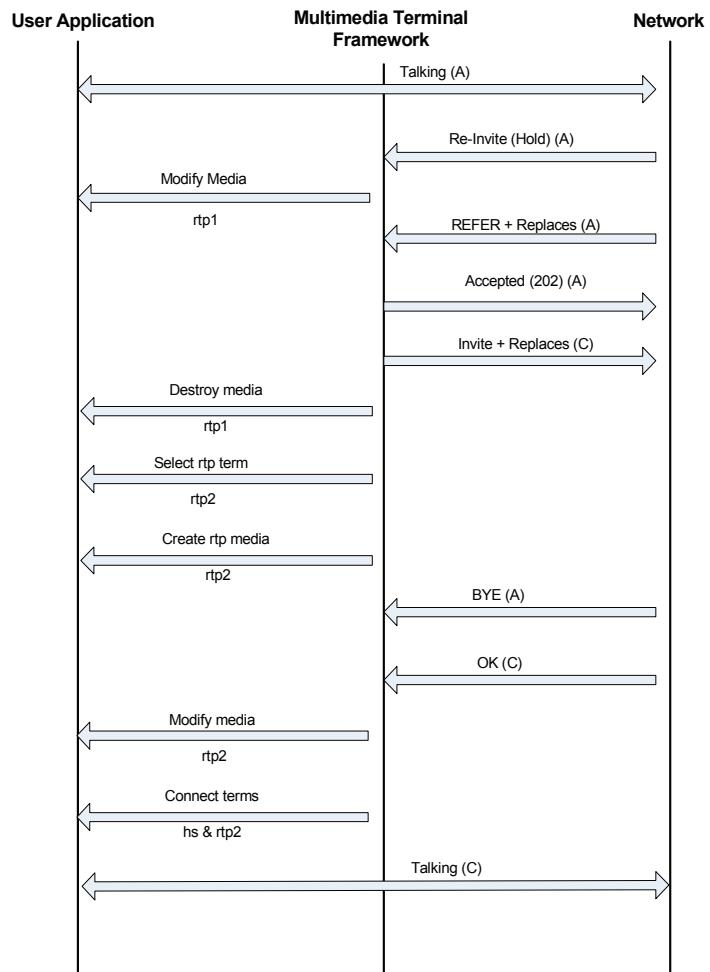


Figure 8-2 Attended Transfer—User B (SIP)

Common Features

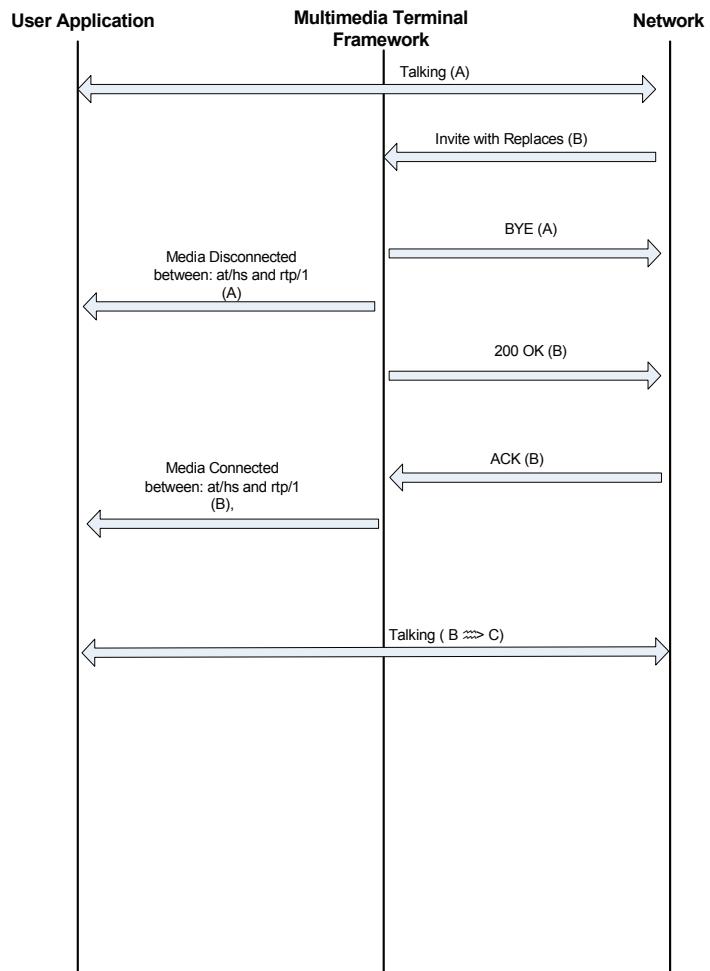


Figure 8-3 Attended Transfer—User C (SIP)

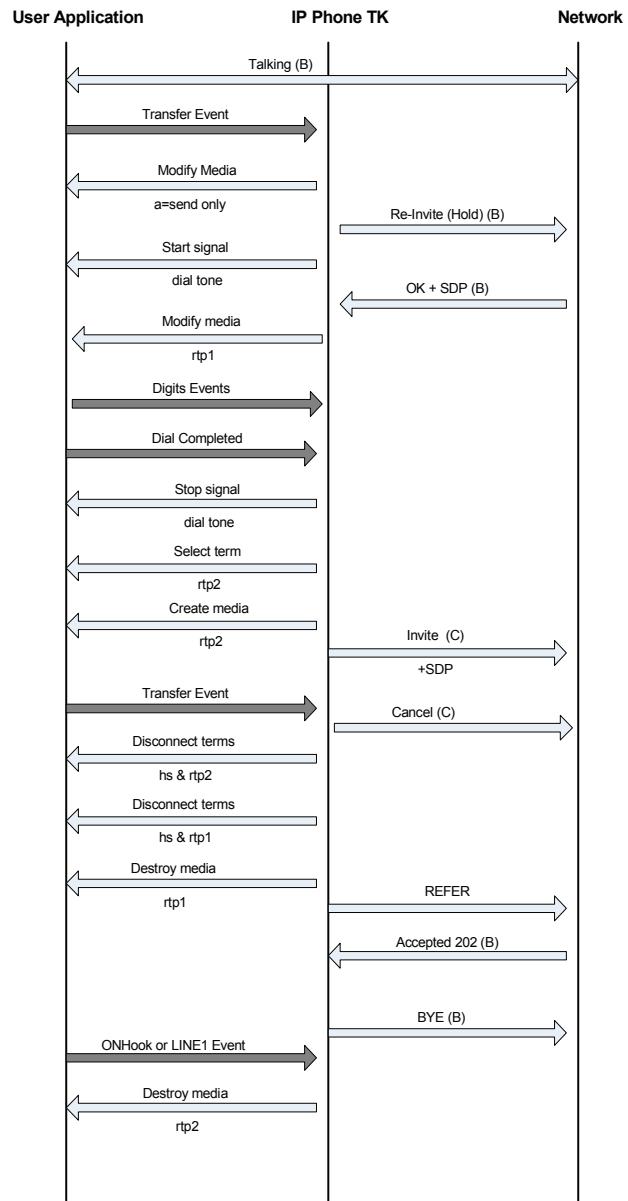


Figure 8-4 Semi-Attended Transfer—User A (SIP)

Common Features

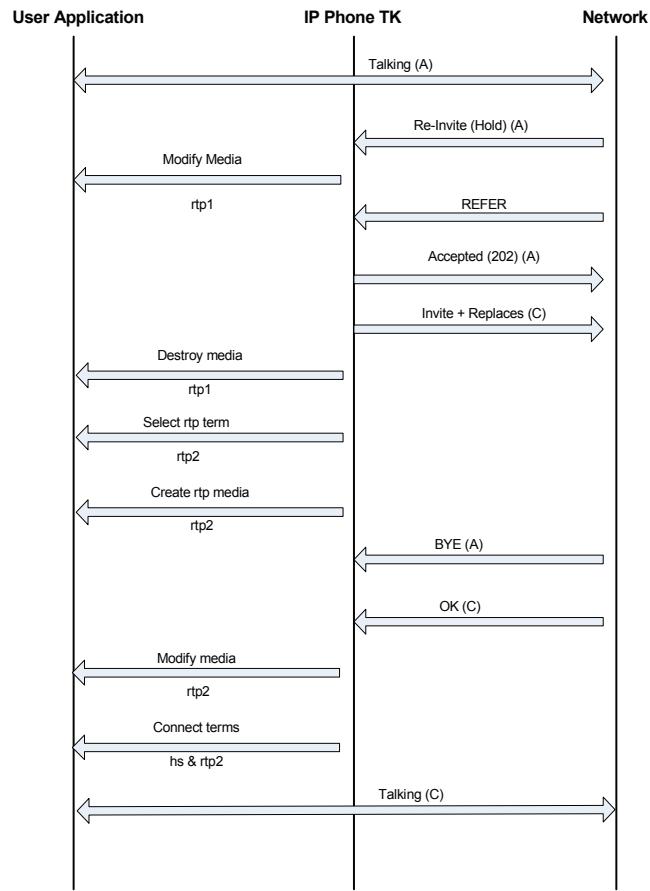


Figure 8-5 *Semi-Attended Transfer—User B (SIP)*

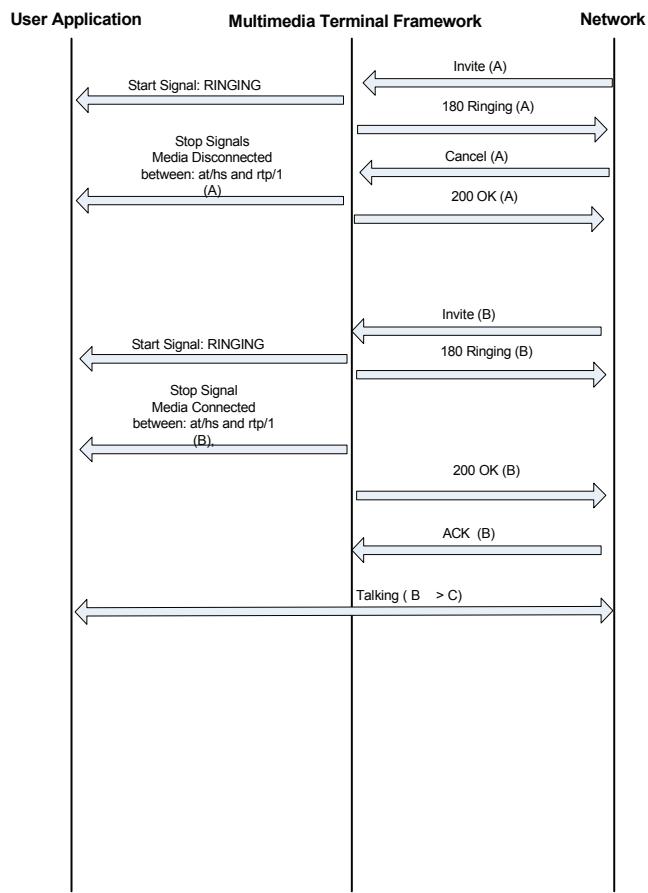


Figure 8-6 *Semi-Attended Transfer—User C (SIP)*

BLIND TRANSFER

Blind Transfer enables the transferring User A to transfer an existing call with User B (primary call) into a new call between User B and a User C selected by User A, while skipping the consultation phase. User A does not establish a call with User C but causes Users B and C to be connected in a new secondary call and then drops out of the call; the primary call ends.

Note From the point of view of the end user, Blind Transfer may also be supported by using the Transfer key. See [Attended and Semi-Attended Transfer](#) for details.

The section below illustrates how a successful Blind Transfer can be achieved.

PERFORMING A BLIND TRANSFER

User Steps	Results	Comments
1. Setup a call with B	A and B are connected.	Call with B may be incoming or outgoing.
2. Press Blind Transfer	Call with B is put on Hold: --Media with B is disconnected. --Hold indicator is turned on.	
3. Dial digits	Digits are mapped to destination address.	After dialing digits, if no ringing tone is heard, the user can go back to the primary call by pressing Line 1.
4. Go On-Hook	Call with B is disconnected. B and C are connected.	By pressing On-hook or line.

To support this feature:

- Send a Blind Transfer event each time the user presses the Blind Transfer key:
pkg = "kf"
id = "ku" when the key is up and "kd" when the key is down.
keyid = "kbt"

IMPLEMENTATION

Transfer is implemented by a REFER message, based on draft-ietf-sipping-cc-transfer-03.txt, section 5.1.

The figures below demonstrate the call flow of Transfer in SIP for parties A, B and C. The left-hand column shows the events received from the user application and the signals sent to the user application. The right-hand column shows the outgoing and incoming network messages.

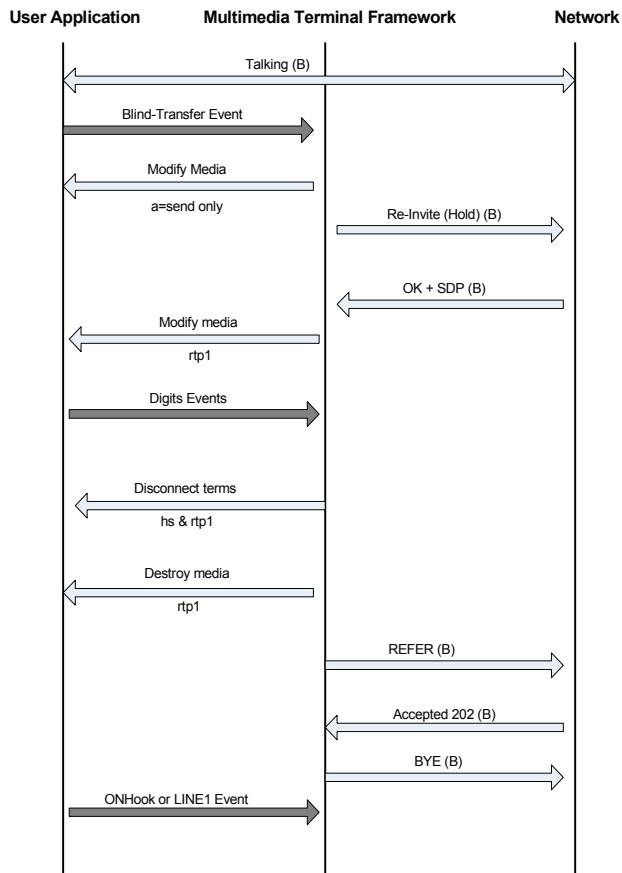


Figure 8-7 Blind Transfer—User A (SIP)

Common Features

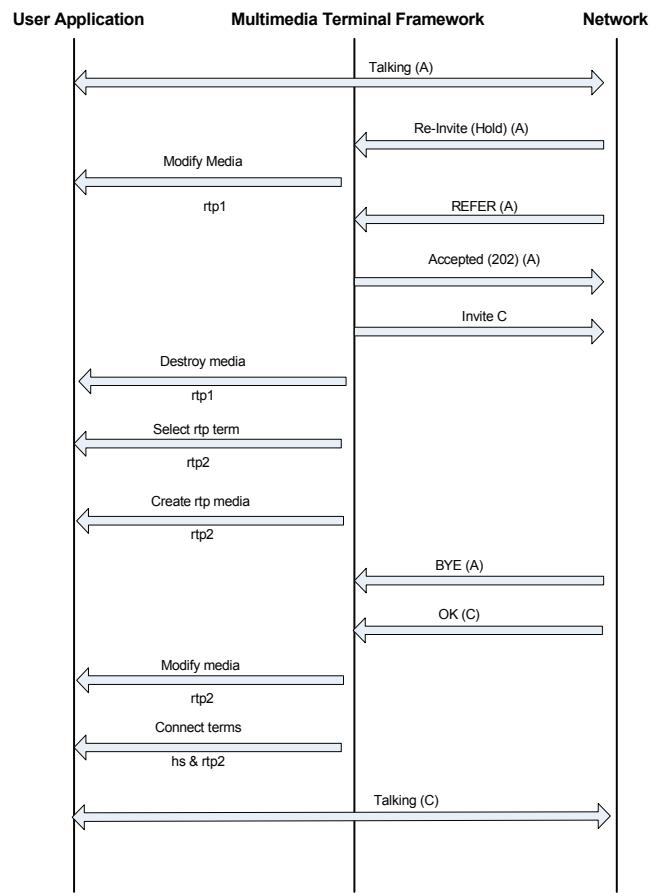


Figure 8-8 *Blind Transfer—User B (SIP)*

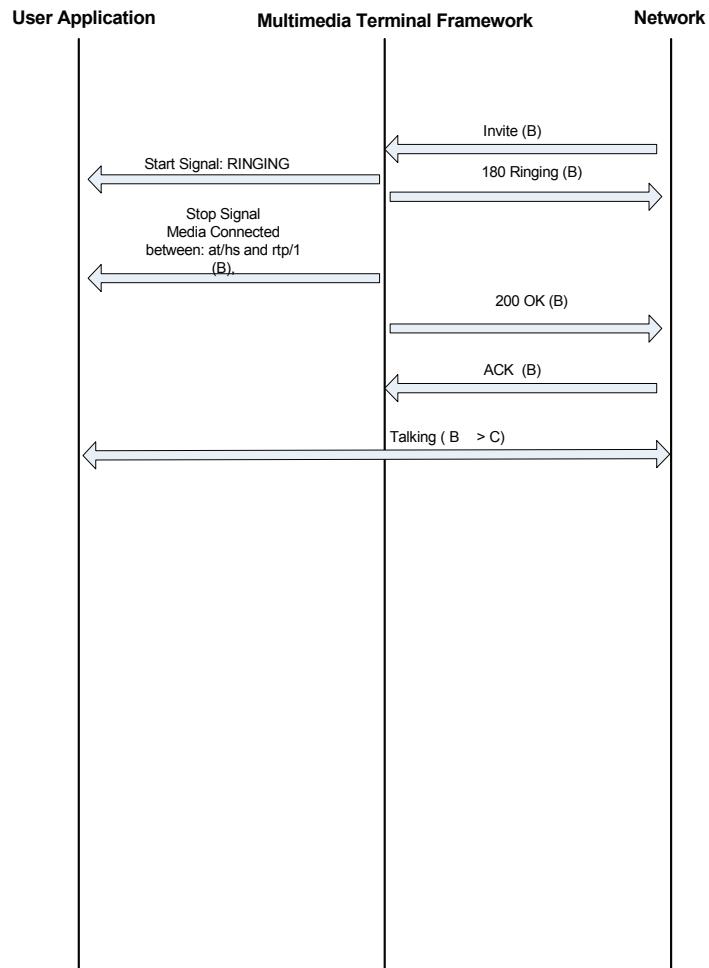


Figure 8-9 *Blind Transfer—User C (SIP)*

THREE-WAY CONFERENCE

A Three-Way Conference allows three users to be connected simultaneously in a single call. In a Three-Way Conference, the party creating the conference does the media mixing so that the RTP streams of both remote parties are connected through the RTP stream of the party creating the Conference.

To support this feature:

- Hardware requirements—To enable the media flow between all three parties of the call, the hardware should be able to mix the RTP input of both remote parties.
- Send a Conference event each time the user presses the Conference key:
`pkg = "kf"`
`id = "ku" when the key is up and "kd" when the key is down.`
`keyid = "kc"`

When the user presses the Conference key, the active call is put on Hold and a new line becomes active. The user can complete the Conference only after the new call is established. The sections below illustrate two ways in which the user can create a successful Conference.

To disconnect the Conference Call, the user can do one of the following:

- Go On-hook by pressing On-hook or Line
- Press the Conference key

Both calls will be disconnected as a result.

PERFORMING CONFERENCE ON AN ACTIVE CALL

User Steps	Results
1. Setup a call with B	Call with B may be incoming or outgoing.
2. Press Conference	Call with B is put on Hold: --Media with B is disconnected. --Hold indicator is turned on. Dial tone is played on Line 2.
3. Setup a call with C	Dial digits and wait for the call to be answered.
4. Press Conference	Call with B is Unheld: --Media with B is connected. --Hold indicator is turned off. All three parties are connected.

PERFORMING CONFERENCE ON A HELD CALL

User Steps	Results
1. Setup a call with B	Call with B may be incoming or outgoing.
2. Press Hold	Media with B is disconnected. Hold indicator is turned on.
3. Press Line 2	Dial tone is played on Line 2.
4. Setup a call with C	Dial digits and wait for the call to be answered.
5. Press Conference	Call with B is Unheld: --Media with B is connected. --Hold indicator is turned off. All three parties are connected.

Common Features

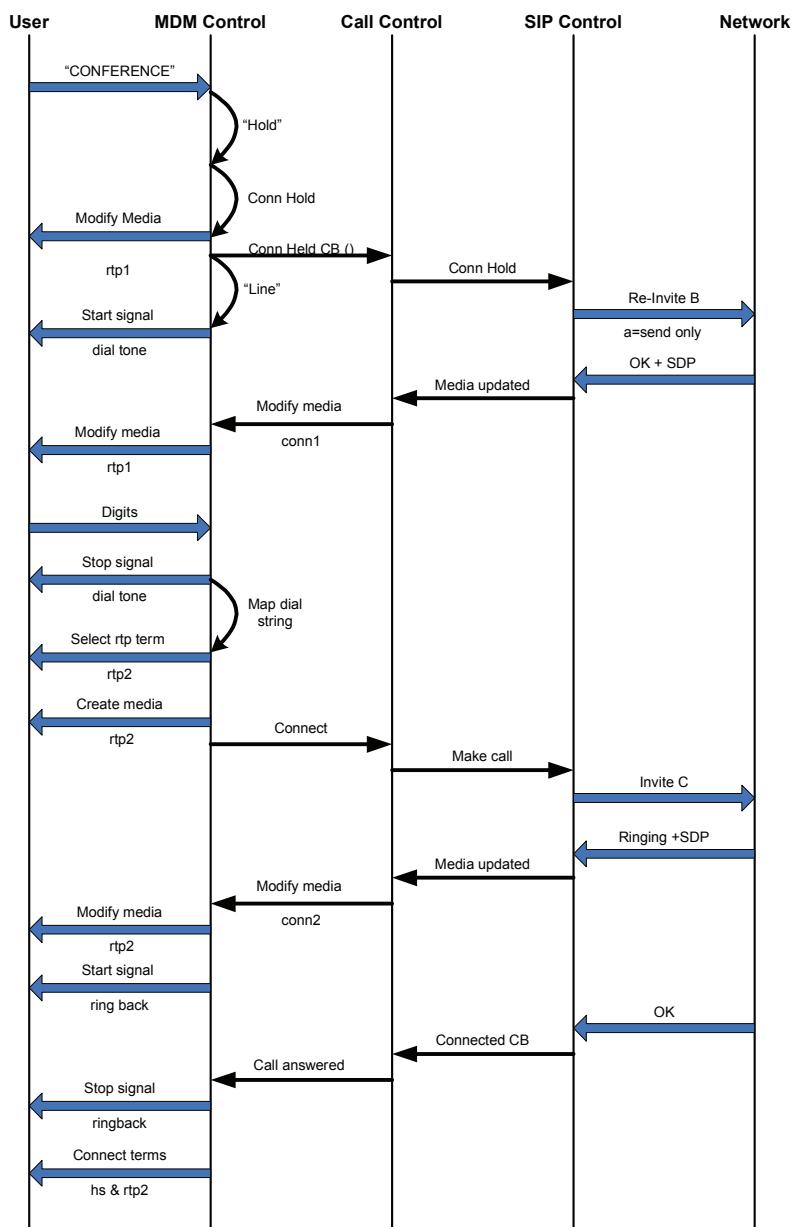


Figure 8-10 Three-Way Conference—SIP Implementation

HEADSET/HANDSFREE

The Headset/Handsfree feature enables the user to direct the output media stream to different AT terminations, according to the key(s) pressed by the user. When setting a new call, one of the AT terminations is selected as active, according to hardware events. When all terminations are available, the default one will be the speaker. The media events and the media from the remote side are delivered to the selected (active) termination. The active AT termination will be connected to the RTP termination during the call.

During a call, the active state can be transferred from one termination to another by pressing Control. Which AT termination is associated with the active call context, or where the media from the remote side will be redirected, is determined by the state of two function keys (Speaker key and Headset key) and the Hook Switch.

Note The Headset/Handsfree is not supported in Residential Gateways. To support this feature, a UI termination should be registered.

To support this feature:

- Hardware requirements:
Headset and/or Handsfree Transducer devices must be supported.
- Register new terminations:
Two more terminations must be registered: Headset and Handsfree, with the following termination Ids: "at/ht" and "at/hf". An AT termination that is not registered will not be supported.
- Send events:
Send Headset/Handsfree events whenever the user presses the corresponding keys:
pkg = "kf"
id = "ku" when the key is up and "kd" when the key is down.
keyid = "hf" for Handsfree and "ht" for Headset.

Common Features

The following table illustrates the state machine for the Handset (hs), Handsfree (hf), and Headset (ht) AT terminations and the function keys and hook switch state. The table details the actions induced by a given key press or hook state transition.

Current State	Speaker Key Transition	Headset Key Transition	Hook Switch Transition
Idle Speaker key = OFF Headset key = OFF Hookswitch = on-hook	Initiate call with Handsfree (hf). Turn on speaker indicator.	Initiate call with Headset (ht). Turn on Headset indicator.	Initiate call with Handset (hs).
Handset Active Speaker key = OFF Headset key = OFF Hookswitch = off-hook	Move media from Handset (hs) to Handsfree (hf). Turn on speaker indicator.	Move media from Handset (hs) to Headset (ht). Turn on Headset indicator.	Terminate call.
Handsfree Active (1) Speaker key = ON Headset key = OFF Hookswitch = on-hook	Terminate call. Turn off speaker indicator.	Move media from Handsfree (hf) to Headset (ht). Turn off Speaker indicator. Turn on Headset indicator.	Move media from Handsfree (hf) to Handset (hs). Turn off speaker indicator.
Handsfree Active (2) Speaker key = ON Headset key = OFF Hookswitch = off-hook	Move media from Handsfree (hf) to Handset (hs). Turn off Speaker indicator.	Move media from Handsfree (hf) to Headset (ht). Turn off Speaker indicator. Turn on Headset indicator.	No effect.
Headset Active (1) Speaker key = OFF Headset key = ON Hookswitch = on-hook	Move media from Headset (ht) to Handsfree (hf). Turn off Headset indicator. Turn on Speaker indicator.	Terminate call. Turn off Headset indicator.	Move media from Headset (ht) to Handset (hs). Turn off Headset indicator.
Headset Active (2) Speaker key = OFF Headset key = ON Hookswitch = off-hook	Move media from Headset (ht) to Handsfree (hf). Turn off Headset indicator. Turn on Speaker indicator.	Move media from Headset (ht) to Handset (hs). Turn off Headset indicator.	No effect.

MUTE

The Mute feature enables the user to block all outgoing media during a call. The user will receive media, but no media will be sent out to the remote party. Whenever the user presses the Mute key, a Mute event will be sent to the Multimedia Terminal Framework. The Terminal Framework Call Control toggles the Mute states from ON to OFF and processes the event according to the following rules:

- The Mute event has effect only if it is sent during an active call.
- Sending a Mute event during a call that is on Hold will have no effect.
- Pressing Hold, Conference, or Transfer while a call is in the Mute state cancels Mute.

IMPLEMENTATION

The Multimedia Terminal Framework implements this feature by disconnecting the media streams and reconnecting them in receive-only mode.

To support this feature:

- Send a Mute event every time the user presses the Mute key:
Pkg= "kf"
Id = "kd" and "ku"
Parameters: keyid= "mu"

User Steps	Results
1. Setup a call with B	Call with B may be incoming or outgoing.
2. Press Mute	Media mode is changed to receive-only. Mute indicator is turned on.
3. Press Mute	Media mode is changed to send-receive. Mute indicator is turned off.

WARM RESTART

Warm Restart enables you to restart the Multimedia Terminal Framework without shutting down your application. This allows you to change the configuration of the Multimedia Terminal Framework or the Stack during runtime of your application. When you restart the Multimedia Terminal Framework, the Protocol Stack is also restarted.



To perform Warm Restart:

- 1. Shutdown the Multimedia Terminal Framework**

For more information, see the chapter [Building Your Application](#), section [Initialization Steps](#).

- 2. Change the configuration**

For more information, see the chapter [Building Your Application](#), section [Configuration](#).

- 3. Initialize the Multimedia Terminal Framework**

For more information, see the chapter [Building Your Application](#), section [Initialization Steps](#).

IPv6

COMPILING AND CONFIGURING IPv6

The RADVISON Multimedia Terminal Framework supports IP version 6 (IPv6). The Multimedia Terminal Framework can be configured to listen to IPv6 or IPv4 addresses, and to receive and send messages using IPv6 or IPv4 packets.

The network type is determined by two variables:

1. The `RV_NET_TYPE` flag. This flag may be configured in the file "usrconfig.h", or (on Unix platforms) by compiling with `IPV6=on`.

When the pre-processor flag `RV_NET_TYPE` is defined as `RV_NET_IPV4`, the Multimedia Terminal Framework supports IPv4 only.

When the pre-processor flag `RV_NET_TYPE` is defined as `RV_NET_IPV6`, the Multimedia Terminal Framework will support either IPv4 or IPv6, depending on the configuration of the `LocalAddress` parameter at startup.

2. The `LocalAddress` parameter in the configuration (see [Configuration](#)). In `LocalAddress`, you can configure either an IPv6 or an IPv4 address. However, all IP addresses in the configuration should be of the same type as `LocalAddress`. A configuration example:

IPv6: `LocalAddress= [fec0:20::20b:dbff:fea4:a6c5]%`1

IPv4: `LocalAddress=172.20.77.1`

The following table shows the possible combinations of these two variables:

<code>RV_NET_TYPE =</code> <code>RV_NET_IPV6</code>	<code>RV_NET_TYPE =</code> <code>RV_NET_IPV4</code>
<code>LocalAddress = IPv6</code>	<code>IPv6</code>
<code>LocalAddress = IPv4</code>	<code>IPv4</code>

LIMITATIONS

- The Multimedia Terminal Framework currently supports only one local address. Multiple IP addresses are not supported.
- IPv6 is not supported on the WinCE platform.
- The user application is responsible for the consistency of IP addresses in the Multimedia Terminal Framework configuration. For example, if `LocalAddress` has been specified

as IPv6, then all parameters that define IP addresses (Proxy/Registrar, dialplan IP addresses, etc.) must also be IPv6 addresses.

The Redial feature allows the user to dial the sequence of digits that was last dialed on the phone. To use this feature, the user should go off-hook and then press the Redial button. This will result in last-dialed digits being dialed again.

- Send a Redial event each time the user presses the Redial key:

pkg = "kf"

id = "ku" when the key is up and "kd" when the key is down.

keyid = "redial"

8.3 PROTOCOL-SPECIFIC FEATURES

SIP FEATURES

The Multimedia Terminal Framework features described below are available only for the specified protocol:

- SIP Features

This section describes features that are available only for the SIP protocol:

- Registration
- Registration Refresh
- Unregistration
- Outbound Proxy
- Client Authentication
- STUN
- Dynamic Media Change
- TLS
- Out-of-Band DTMF
- Call Forward
- Session Timer
- PRACK
- Early Media
- Distinctive Ringing
- Advanced DNS

REGISTRATION

The user application can register its termination to a Registrar. The Registrar address is configured in the parameter "registrarAddress". A termination can be registered to a Registrar in two different ways: through automatic registration or through manual registration.

Automatic Registration

The Terminal Framework Registration mechanism automatically sends a Registration request to a Registrar for every UI and Analog termination. A Registration will be sent only if the RegistrarAddress parameter has been configured. After the initial request is sent, more requests will be sent periodically (Registration Refresh). Sending the initial Registration request depends on the value of the configuration parameter autoRegister. If the value is True, the initial request is sent automatically when you register the termination to the Multimedia Terminal Framework, by calling:

- `rvMdmTermMgrRegisterPhysicalTermination()` or

- `rvMdmTermMgrRegisterPhysicalTerminationAsync()`

If the value is False, the initial request is not sent and you can use an API to send it. See [Manual Registration](#) for further details. For more information on the autoRegister parameter, see the section [Configuration](#) in the [Building Your Application](#) chapter.

[Manual Registration](#)

Calling one of the following APIs will send a Register message to the configured Registrar:

- `rvMdmTermMgrRegisterTermToNetwork_0`
This API sends a registration message for a single termination.
- `rvMdmTermRegisterAllTermsToNetwork_0`
This API sends registration messages to all terminations.

REGISTER MESSAGE

The headers of the Register message sent by the Multimedia Terminal Framework will contain the following values, which are based on the following parameters:

- **Scheme**—When TLS is disabled, this field will be set to "sip:", and when TLS is enabled, this field will be set to "sips:".
- **RegistrarAddress**—The value of this field is taken from the configuration parameter: "RegistrarAddress".
- **RegistrarPort**—The value of this field is taken from the configuration parameter: "RegistrarPort".
- **RegistrarAddress**—The value of this field is taken from the configuration parameter: "RegistrarAddress".
- **TermId**—The value of this field is taken from argument "id" when `rvMdmTermMgrRegisterPhysicalTerminationAsync()` or `rvMdmTermMgrRegisterPhysicalTermination()` is called.
- **RegistrationExpire**—The value of this field is taken from the configuration parameter: "RegistrationExpire".

REGISTER REPLY

The Multimedia Terminal Framework handles the following replies:

- **OK (200)**—This reply indicates that Registration has succeeded. The Multimedia Terminal Framework will check whether a new Expires value is offered according to the following:
 - Go through the list of Contact headers in the reply and choose the one that was sent in the request (Register requests always include one Contact).
 - Get the Expires value from the Contact header.
 - Check whether this value is different from the original Expires value sent in the request. If yes, replace the original value with the new one. Further Registration refresh messages will be sent according to the new value.
- **Unauthorized (401)/Proxy Authentication Required (407)**—This reply indicates that Registration has failed, since a Registration request must be authenticated. The Multimedia Terminal Framework will send another Registration request, which includes Authentication credentials. These credentials will be sent with every refresh request.
- **Interval Too Brief (423)**—This reply indicates that Registration has failed, since the Expires value sent in the request is too small and offers a minimum value. The Multimedia Terminal Framework will retrieve the minimum value from the "Min-Expires" header and will send a new Register request with the new value in the Expires header. Further Registration refresh requests will be sent according to the new Expires value.

REGISTRATION REFRESH

After the initial Registration request is sent, a timer is set, and each time the timer expires an additional request (re-Registration request) will be sent. The re-Registration request is sent regardless of the value of the autoRegister parameter and regardless of the response from the Registrar. If the Registrar responds with Unauthorized or Proxy Authentication Required (401/407), another Registration request is sent including Authentication credentials. These credentials will be sent with every additional request.

UNREGISTRATION

The user application can unregister from the Registrar in two different ways: through automatic unregistration or through manual unregistration. The Unregister message is similar to the Register message except for the Expires header value, which is set to zero:

```
Expires: 0
```

Automatic Unregistration

As part of the shutdown process, the user application unregisters terminations from the Multimedia Terminal Framework. When a termination unregisters from the Multimedia Terminal Framework, an Unregister message is sent to the Registrar. For more information, see the chapter [Building Your Application](#).

Manual Unregistration

The following API enables the user application to manually unregister a termination from the SIP Registrar that it configured at initialization:

```
RvBool rvMdmTermMgrUnregisterTermFromNetwork_(
    RvMdmTermMgr*     mgr,
    RvMdmTerm*        term);
```

This callback is defined in the file *rvmmdm.h*.

Parameters

mgr—A pointer to the Termination Manager object.

term—A pointer to the termination to be unregistered.

Return Value

Returns False if sending an Unregister request to the Registrar failed, or True if it was successful.

This API sends an Unregistration request to the Registrar that is configured in the parameter "registrarAddress". After the message is sent, the Multimedia Terminal Framework waits for a reply. If authentication is required, the Multimedia Terminal Framework will send another message with authentication. When a final reply is received from the Registrar, or if the timeout configured in the parameter "unregisterTimeout" expires, the callback RvMdmTermUnregisterTermFromNetworkCompletedCB() is called to indicate that the process has been completed.

It is important to wait for the callback to be called before sending another Register/Unregister request to the Registrar, i.e., before calling any of the following functions:

- rvMdmTermMgrUnregisterTermination()
- rvMdmTermMgrUnregisterTerminationAsync()
- rvMdmTermMgrRegisterPhysicalTermination()
- rvMdmTermMgrRegisterPhysicalTerminationAsync()
- rvMdmTermMgrRegisterAllTermsToNetwork_()
- rvMdmTermMgrRegisterTermToNetwork_()

To use this API, the user application should perform the following steps:

1. Implement callback

```
void RvMdmTermUnregisterTermFromNetworkCompletedCB (
    RvMdmTerm*      term,
    RvmdmError*     mdmError)
```

2. Register callback

```
void rvMdmTermClassRegisterUnregisterTermFromNetwork
CompletedCB(
    RvMdmTermClass*   c,
    RvMdmTermUnregisterTermFromNetworkCompletedCB  un
registerTermFromNetworkCompletedF);
```

3. Call Unregister API

```
RvBool rvMdmTermMgrUnregisterTermFromNetwork (
    RvMdmTermMgr*   mgr,
    RvMdmTerm*      term);
```

OUTBOUND PROXY

When the Multimedia Terminal Framework is configured with an Outbound Proxy, all outgoing messages (including Register messages) will be sent to it, and are directed by the Outbound Proxy to the remote party. This parameter is supported with either an IP address or a domain name. If both an Outbound Proxy and a Registrar have been configured, the Register message is sent to the Outbound Proxy, which in turn forwards it to the Registrar.

CLIENT AUTHENTICATION

When your application sends an Invite or a Register message, it may be required to provide authentication information. If the Terminal Framework receives an Unauthorized response or a Proxy Authentication Requested (401/407) response, it will send an additional message with Authentication credentials, including two configurable parameters: username and password.

These parameters may be configured per application or per terminal:

- To configure the parameters per application, set the values during initialization. For more information, see the chapter [Building Your Application](#), section [Configuration](#).

- To configure the parameters per application, set the values during initialization. For more information, see the chapter [Building Your Application](#), section [Initialization Steps](#).
- To configure the parameters per termination, set the values during registration of the terminal. For more information, see the chapter [Building Your Application](#), section [Registering Terminations](#).

If the parameters are not configured, Authentication credentials will be sent with an empty user name and password.

SIP methods that are automatically authenticated by the Multimedia Terminal Framework:

- REGISTER
- INVITE
- RE-INVITE
- PRACK—requires the SIP Stack to be compiled with the compilation flag `PRACK_AUTHENTICATION_SUPPORT`
- REFER
- BYE

STUN

STUN is a light-weight protocol that allows an application to discover the presence and types of NATs and firewalls between itself and the public Internet. It also enables the application to determine the public IP addresses allocated to it by the NAT. (For more information, see *RFC 3489 STUN—Simple Traversal of UDP Through NATs*.) The Multimedia Terminal Framework allows the user application to integrate a STUN client on top of the Multimedia Terminal Framework, enabling it to successfully make calls from behind NATs.

The STUN feature is supported for Call-leg, RegClient and Subscribe objects. Other SIP objects are not relevant for the Multimedia Terminal Framework.

IMPLEMENTATION

The Multimedia Terminal Framework STUN Manager listens on SIP Stack events of outgoing messages and invokes user callbacks whenever a private address needs to be resolved to a public one. When the address is resolved, the STUN Manager will replace the relevant fields that contain a local address with a resolved address in the outgoing message.

The following fields in SIP messages are replaced as a result of STUN queries:

- **SIP Headers**

- ❑ Via header—IP address and port are replaced. When the port number does not exist, the STUN request is sent with port 5060.
- ❑ Contact header—IP address and port are replaced. When the port number does not exist, the STUN request is sent with port 5060.
- **SDP Body**
 - ❑ Connection Data field ("c=")—IP address is replaced. This is done both at the session level and the media description level.
 - ❑ All RTP ports are replaced.
 - ❑ RTCP port—The Multimedia Terminal Framework supports a reference that adds the RTCP attribute to SDP. (For more information, see *RFC 3605—RTCP Attribute in SDP*.)
When the user application specifies an RTP port (and implicitly assumes that the RTCP port is RTP+1), a STUN request is sent with <ip, rtcp port> and the result is set in the rtpc: attribute in SDP.
- **Handling special cases**
 - ❑ Handling *RFC 2543 Hold*—When sending a Hold message as specified in old SIP *RFC 2543*, the 0.0.0.0 will not be replaced.
 - ❑ Handling port 0—When a media stream is closed by setting the port to 0, as specified in *RFC 3264—An Offer/Answer Model with SDP*, the port number remains 0 and will not be replaced.

Using the RADVISION STUN Client Toolkit

If you are using the RADVISION STUN Client Toolkit, it is strongly recommended that you use the code in the Multimedia Terminal Framework sample application that handles integration with the RADVISION STUN Client. Your application is required to do the following (as illustrated in [Figure 8-11](#)):

- Implement one callback—RvIppStunIsAddressResolveNeededCB. This callback indicates to the Multimedia Terminal Framework whether or not to perform address resolution for this address.

- Send STUN queries for RTP/RTCP addresses—Address resolution is required for RTP/RTCP addresses as well as for SIP addresses. While STUN queries for SIP addresses are handled by the Multimedia Terminal Framework, STUN queries for RTP/RTCP addresses should be handled by the user application. The reason for this is that STUN queries should be sent from the port on which the address resolution is needed, and only the user application has access to the ports on which its media streams were opened. The Multimedia Terminal Framework handles the STUN replies and replaces the relevant fields in the outgoing SDP message.

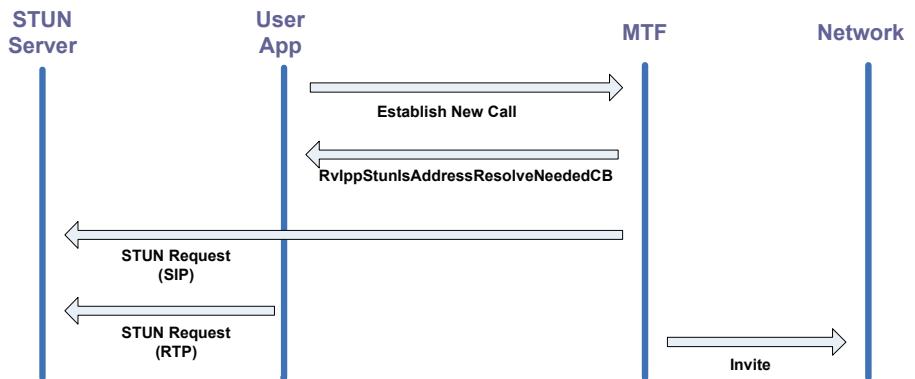


Figure 8-11 Using the RADVISION STUN Client Toolkit

The steps below describe the implementation required from your application.

Note It is strongly recommended not to make changes in the implementation code besides the ones mentioned in the steps below.

Using a non-RADVISION STUN Client

If you are integrating your own or a third-party STUN Client, you need to follow the steps below. As illustrated in [Figure 8-12](#), Multimedia Terminal Framework STUN callbacks are invoked when a SIP message is about to be sent. Your application should implement these callbacks by sending STUN

queries (using your STUN Client). When a reply is received from the STUN Server, the Multimedia Terminal Framework will replace the addresses in the relevant fields of the outgoing message before sending it out.

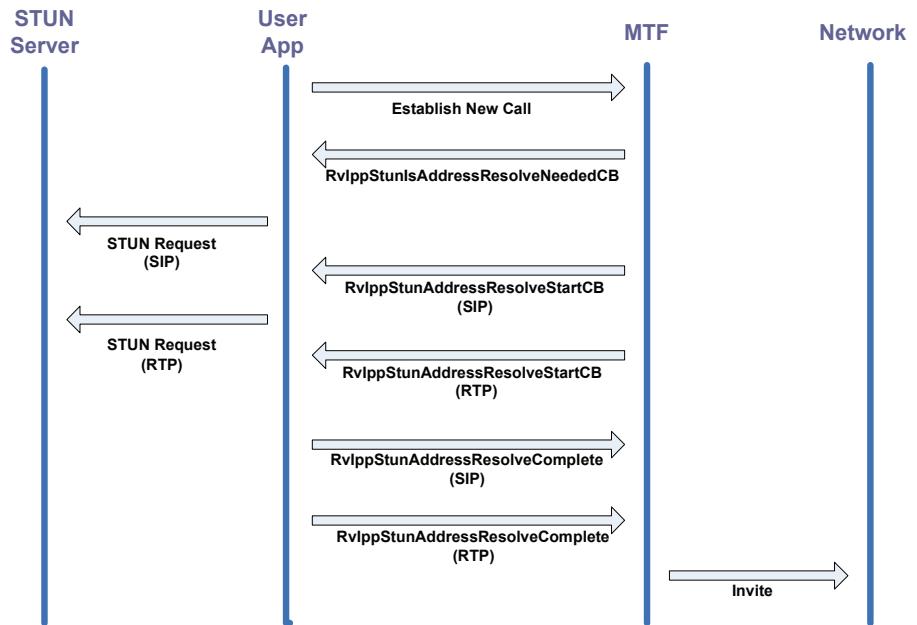


Figure 8-12 Using a non-RADVISION STUN Client Toolkit

To support this feature:



Step 1: Implement callback `RvIppStunIsAddressResolveNeededCB()`

This user callback is invoked once in each outgoing message to decide whether the private address should be resolved to a public one or not.

If you are using the RADVISION STUN Client

This is the only callback you need to implement. Change the existing implementation of `stunIsAddressResolveNeededCB()` to the requirements of your application.

If you are using a different STUN Client

The user application returns True if the address needs to be resolved, or False if not. If False is returned, no further STUN callbacks will be invoked before the message is sent out.

Example

The following example checks if the call is destined inside the private network. If so, there is no need to perform STUN queries and the callback returns False.

```
RvBool stunIsAddressResolveNeededCB(
    IN RvAddress* addrDest)
{
    /*
     *      test addrDest vs. gw->stunNeedMask
     */
    char     szDest[64], *ptrDest, *ptrMask;

    RvAddressGetString( addrDest, sizeof(szDest), szDest);
    ptrDest = szDest;
    ptrMask = gw.stunNeedMask;

    while( *ptrDest != '\0' && *ptrMask != '\0')
    {
        if ( *ptrMask == 'x')
            /*they are identical in mask part*/
            return rvFalse;
        else if(*ptrDest != *ptrMask)
            return rvTrue;

        ptrDest++;
        ptrMask++;
    }
    /*they are identical*/
    return rvFalse;
}
```



Step 2: Implement callback RvIppStunAddressResolveStartCB()

This user callback is invoked for every <ip,port> pair in the outgoing message that needs to be resolved and replaced.

If you are using the RADVISION STUN Client

Use the implementation described in the Multimedia Terminal Framework sample application in the function **stunClientSendRequest()**. It is recommended not to make changes in the implementation code.

If you are using a different STUN Client

When this callback is invoked, the user application should send one or more STUN queries to resolve private addresses. In the implementation of this callback, the application should build and send STUN queries to begin the address resolution process, i.e., querying the STUN Server. The callback is invoked once for each SIP header and SDP header. RTP/RTCP addresses STUN queries should be sent for both SIP and RTP/RTCP addresses:

- STUN queries for SIP addresses—STUN query for SIP addresses should be sent via the SIP Stack port. To send the STUN query, it must use the function **RvIppStunMgrGetSendMethod()** to retrieve the sending function, and use the sending function for message sending.
- STUN queries for RTP/RTCP addresses—STUN query for RTP/RTCP addresses should be sent via the RTP Stack, by using RTP Stack APIs.

This callback can be implemented asynchronously, meaning that it will return before a STUN Server reply is received so that the Multimedia Terminal Framework thread will not be blocked.

Example

The following example demonstrates an implementation of the callback **RvIppStunAddressResolveStartCB()**.

```
RvStatus userSipStunclientSendRequest (
    RvIppStunAddrData *addrData,
{
    RvStatus rv;
    RV_CHAR strIp[64];
    RV_CHAR *stunAddress;
    RvUint intIp;
```

```
StunAddress4 dest,from;
SendMethod method;
char buf[1024];

stunAddress = (char *)rvStringGetData(&gw.stunServerAddress);

if (stunAddress == NULL)
{
    rv = RV_ERROR_BADPARAM;
    goto err_exit;
}

intIp = inet_addr(stunAddress);
if (intIp == INADDR_NONE)
{
    rv = RV_ERROR_BADPARAM;
    goto err_exit;
}
dest.addr = intIp;
dest.port = gw.stunServerPort;

RvAddressGetString(&(addrData->inAddr),sizeof(strIp),strIp);
from.addr = inet_addr(strIp);
if (from.addr == INADDR_NONE)
{
    rv= RV_ERROR_BADPARAM;
    goto err_exit;
}
from.port = RvAddressGetIpPort(&(addrData->inAddr));

/* Retrieve SIP Stack message sending function */
RvIppStunMgrGetSendMethod (gw.stunMgrHndl, &method);
/* Build STUN query according to RFCstun */
buildSTUNQuery(from,buf);
/* Send STUN query by SIP message sending function */
method->sendMsgCB( method->usrData, szIp, dest.port, buf, len
);

return rv;
}
```



Step 3: Implement callback RvIppAddressResolveReplyCB()

This callback is invoked when a reply from the STUN Server is received.

If you are using the RADVISION STUN Client

Use the implementation described in the Multimedia Terminal Framework sample application in the function **stunClientReplyReceived()**. It is recommended not to make changes in the implementation code.

If you are using a different STUN Client

After a query is sent to the STUN Server, by default the STUN Server will send a response to the originator of the request—in this case, to the SIP Stack port. This callback is invoked when a reply from the STUN Server is received.

Note The user application can instruct the STUN server to send the responses to a different port by filling in the ResponseAddress field in the STUN query.

The user application should call the following API to notify the Multimedia Terminal Framework about the resolved address:

```
RvStatus RvIppStunAddressResolveComplete(
    RvIppStunAddrData*     addrData,
    RvBool                 bStatusOK)
```

with these parameters:

addrData—This structure contains the resolved IP address and port.

bStatusOK—This parameter indicates whether the address was resolved successfully or not.



Step 4: Initialize the STUN Manager

If you are using the RADVISION STUN Client

Use the implementation described in the Multimedia Terminal Framework sample application in the function **stunClientInit()**. This function initializes and configures both the RADVISION STUN Client and the Multimedia Terminal Framework STUN Manager. It is recommended not change the implementation code, except for the configuration parameters.

If you are using a different STUN Client

The STUN Manager is initialized by calling the following function:

```
RvIppStunMgrHandle RvIppStunMgrCreate(
    IN RvIppStunMgrParam*     param);
```

The RvIppStunMgrParam configuration structure should be set with pointers to the callback implementations (see Steps 1-3 above).

Example

```
RvIppStunMgrParam      param;
param.natTimeout      = gw->stunNATTTimeout;
param.msgTimeout       = 3; /* wait three seconds for STUN replies
before giving up */

param.isAddressResolveNeededCB      =
stunIsAddressResolveNeededCB;
param.handleAddressResolutionReplyCB = stunReplyCB;
param.addressResolveStartCB= userSipStunclientSendRequest;

gw->stunMgrHndl = RvIppStunMgrCreate ( &param);
```



Step 5: Send STUN queries for RTP/RTCP addresses

If you are using the RADVISION STUN Client

Your application is required to implement two procedures to replace RTP and RTCP addresses in outgoing SDP messages:

- **transmitStunReqOverRtp()**—for the RTP address
- **transmitStunReqOverRtcp()**—for the RTCP address

These two procedures should send a STUN query to the STUN Server over RTP/RTCP sockets to resolve the public IP address and port number of the RTP/RTCP media streams. These queries can be sent by the RTP application only, as it alone binds the RTP/RTCP sockets.

The user application needs the following parameters for sending STUN queries:

- **STUN message** (parameters **msg** and **msgSz**)—a buffer and its size, this is the actual STUN query that the user application should send to STUN Server. The buffer should be sent as is. This buffer is received as an argument of the two procedures.

- **RTP or RTCP port** (parameters **rtpPort** and **rtpcPort**)—the port number through which the user application should send the STUN query. See sample code in the procedures to extract it.
- **STUN server IP address** (parameter **stunSrvaddr**)—the IP address of the STUN Server, i.e., the destination address to which the user application should send the STUN query. See sample code in the procedures to extract it.
- **STUN server port number** (parameter **stunSrvPort**)—the port number of the STUN Server. See sample code in the procedures to extract it.

The STUN Server replies will be received and handled by the Multimedia Terminal Framework STUN integration. When the replies are received, the Multimedia Terminal Framework STUN integration will update the relevant SDP fields in the outgoing message.

If you are using a different STUN Client

You should already be covered by the implementation of `IppStunAddressResolveStartCB()` in Step 2.

ON SHUTDOWN

If you are using the RADVISION STUN Client

Use the implementation described in the Multimedia Terminal Framework sample application in the function `stunClientEnd()`. This function destructs both the RADVISION STUN Client and the Multimedia Terminal Framework STUN Manager. It is recommended not to make changes in the implementation code.

If you are using a different STUN Client

Terminate STUN Manager. The STUN Manager must be terminated after the Multimedia Terminal Framework shuts down.

Example

```
rvMdmTermMgrStop (&gw->gatewayBase) ;
rvMdmTermMgrDestruct (&gw->gatewayBase.termMgr) ;
RvIppStunMgrDelete ( gw->stunMgrHndl) ;
```

LIMITATIONS

The following limitations exist:

- Only SIP requests are allowed to be asynchronous. SIP responses **must** invoke RvIppStunAddressResolveComplete inside the implementation of RvIppStunAddressResolveStartCB.
- The refreshing of NAT bindings is **not** implemented at this stage.
- The STUN feature does not work on non-dialog messages, such as INFO.

CODE FLOW WHEN USING A NON-RADVISION STUN CLIENT

Figure 8-13 illustrates a successful STUN scenario between the user application and the Multimedia Terminal Framework when your application does **not** use the RADVISION STUN Client.

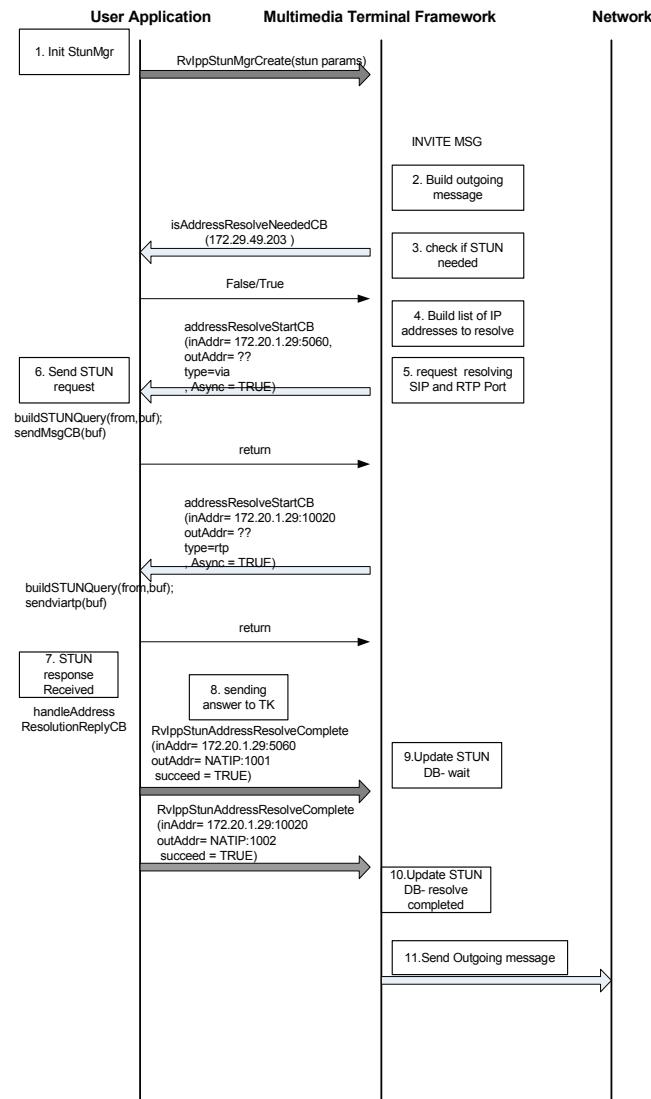


Figure 8-13 Successful STUN Scenario

The steps below show a code flow with STUN for a successful basic call when using a non-RADVISION STUN Client.

Table 8-1 Typical Code Flow with STUN when using a non-RADVISION STUN Client

Step	Performed by	Description
1. Initialize the STUN Manager	User application	Call the function RvIppStunMgrCreate.
2. Build an outgoing message	Multimedia Terminal Framework	After the user dials digits, the Multimedia Terminal Framework builds an outgoing SIP message. The message will not be sent until all addresses of the message are resolved.
3. Check if STUN address resolution is needed	Multimedia Terminal Framework + user application	The isAddressResolveNeededCB is invoked. In this function the user application decides whether or not to replace IP addresses for this SIP message.
4. Build a list of RvAddress pairs to be resolved	Multimedia Terminal Framework	The Multimedia Terminal Framework parses the outgoing SIP and SDP message and adds a list of RvAddresses that require resolving to its internal database.
5. Request address resolution	Multimedia Terminal Framework	Multimedia Terminal Framework calls the following callback once for every address that needs to be resolved in the outgoing message.
6. STUN client builds STUN request	User application	In the implementation of RvIppStunAddressResolveStartCB, the user application code should build a STUN message to send with the STUN Manager-supplied SendMSgCB.
7. STUN response is received	User application	STUN responses to the SIP query are handled in the callback RvIppAddressResolveReplyCB.

Table 8-1 *Typical Code Flow with STUN when using a non-RADVISION STUN Client*

Step	Performed by	Description
8. Send a response to Multimedia Terminal Framework	User application	The user application calls the function RvIppStunAddressResolveComplete() to inform the Multimedia Terminal Framework that address resolution is complete.
9. Update database and wait	Multimedia Terminal Framework	The Multimedia Terminal Framework updates its database and waits for the resolution of the remaining <ip-address,ports> pairs that relate to the same message.
10. Update final <ip-address,port>	Multimedia Terminal Framework	Final pair is received.
11. Resume sending outgoing SIP message	Multimedia Terminal Framework	The Multimedia Terminal Framework informs the SIP Stack to send out the message.

DYNAMIC MEDIA CHANGE

The Dynamic Media Change feature enables one party of a call to change the parameters of existing media during the call. The remote party can either accept the request (as a result, the parameters of the existing media will be changed on both sides) or reject the request (the parameters will remain the same).

The Multimedia Terminal Framework supports sending and receiving Modify Media requests during a call.

Notes:

1. When Modify Media is initiated by the user (Outgoing Request), only the media of the active call will be affected.
2. This feature is blocked if another Modify Media process (initiated by an outgoing or incoming request) is still being processed.
3. This feature is disabled during Transfer.

IMPLEMENTATION

The Dynamic Media Change feature is implemented by a Re-Invite message with SDP body according to the Offer/Answer model (*RFC 3264*).

Outgoing Request

Sending a Modify Media request is an asynchronous process (see Figure 8-14).

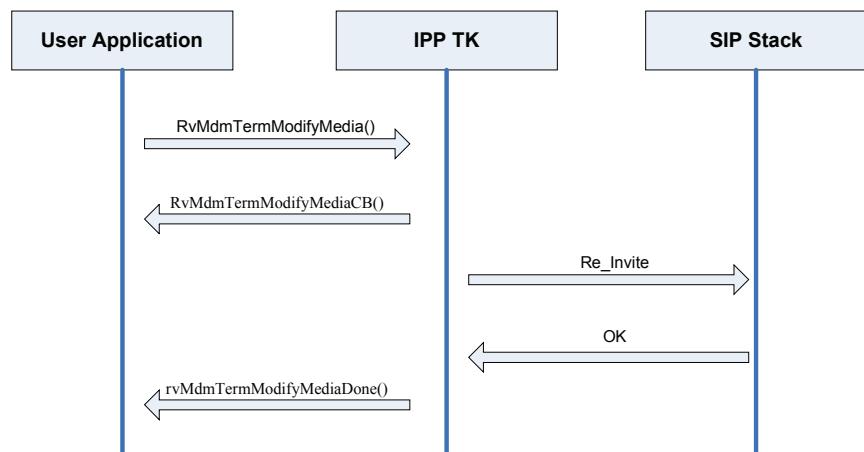


Figure 8-14 *Outgoing Request*

Incoming Request

Receiving a request to modify media is a synchronous process (see [Figure 8-15](#)).

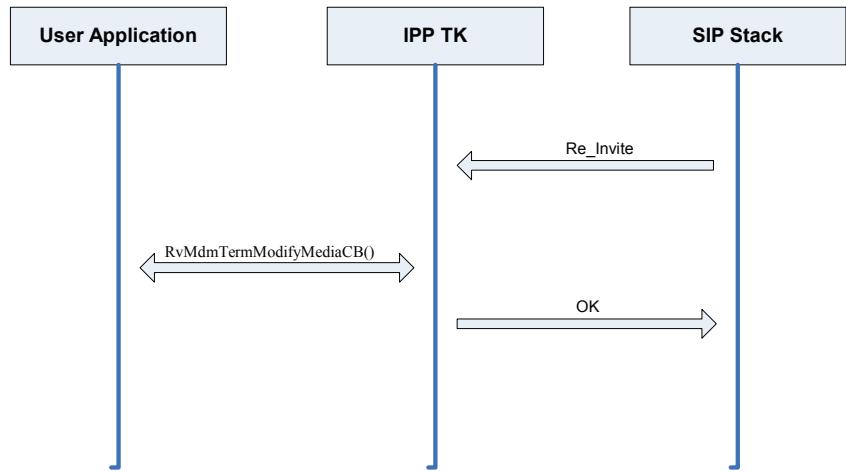


Figure 8-15 Incoming Request

To use this functionality, the user should consider Step 2 (below) for the implementation of the callback `rvMdmTermModifyMediaCB()`. This callback should have been implemented already in the call establishment process. For the Modify Media feature, the callback is called during a connected call.

To support this feature:

OUTGOING REQUEST

The process of sending a Modify Media request during a call is outlined here:



Step 1: Implement callback **RvMdmTermModifyMediaCompletedCB**

When the Multimedia Terminal Framework receives a final reply from the remote party, it notifies the user by calling the following callback function:

```
void rvMdmTermModifyMediaDone_(
    RvMdmTerm*           term,
    RvBool                status,
    RvMdmMediaDescriptor* media,
    RvMdmMediaStreamDescr* streamDescr,
    RvMdmTermReasonModifyMedia reason);
```

with the following parameters:

term—Pointer to the UI or Analog termination.

status—Status of the process: True in case of success, False in case of failure. If True, the user can stop listening with the old media parameters and keep the new parameters open. If False, the user can stop listening with the new media parameters and keep the old ones open.

media—The new local parameters (received from the user) and the parameters received from the remote party.

streamDescr—Output parameter. After reviewing the parameters received by the remote party (in the Media parameter), the user may choose a subset of the new parameters that were offered by him. Once this is done, the object will contain the parameters of the media the user actually opened.

reason—If the status is False, this parameter will include the reason for the failure.



Step 2: Call the following function to start the process:

```
RvBool rvMdmTermModifyMedia(
    RvMdmTerm*   term,
    RvSdpMsg*    sdpMsg);
```

with the following parameters:

term—Pointer to the UI or Analog termination.

sdpMsg—SDP object that contains the new media parameters.

This function puts an event in the MDM queue and returns immediately so as to allow the event to be processed in the context of the Multimedia Terminal Framework thread. The Multimedia Terminal Framework sends the remote party a Re-Invite message with the new media parameters.

INCOMING REQUEST

No actions are required from the user application to support incoming requests for dynamic media change. The following callback is already implemented by the user application as part of media integration (see the chapter [Building Your Application](#), section [Step 4: Integrating Media](#)):

```
RvMdmTermModifyMediaCB(
    RvMdmTerm*           term,
    RvMdmMediaStream*    media,
    RvMdmMediaStreamDescr* streamDescr,
    RvMdmError*          mediaError);
```

indicating to the user to change media parameters of an existing media stream. The callback is called with the following parameters:

term—A pointer to termination.

media—Object that includes the parameters received from the remote party.

streamDescr—Output parameter. This object includes the media parameters the user has opened; a subset of the parameters offered by the remote party in the Media parameter.

mediaError—This parameter is not used.

Return value:

True if the user accepts the request, or False if the user rejects the request (for example, if no supported codecs were found in the received parameters). When this callback returns True, the Multimedia Terminal Framework sends a 200/OK reply to the remote party. When this callback returns False, the Multimedia Terminal Framework sends a 606/MediaNotAcceptable reply to the remote party.

CALL FLOW

Outgoing Request

The table below describes the flow of sending a request for media change during a call. The call is connected to the remote party and wants to change media parameters.

Step	Performed by	
Open new media		Start listening with new media parameters while listening also with the old ones.
Call function rvMdmTermModifyMedia()	User application	The new parameters will be passed to this function as arguments.
Send Re-Invite message	Multimedia Terminal Framework	The message will include the new parameters received from the user application.
Receive a reply from remote party	Multimedia Terminal Framework	Reply of OK (200) indicates that the remote party has accepted the request.
Invoke callback RvMdmTermModifyMediaCompletedCB()	Multimedia Terminal Framework + user application	The status argument indicates whether or not the remote party has accepted the request.

Incoming Request

The table below describes the flow of receiving a request for media change during a call.

Step	Performed by	
Re-Invite message is received	Multimedia Terminal Framework	
Process Re-Invite message	Multimedia Terminal Framework	Check whether the message indicates a request for Modify Media or a request for Hold/Unhold.
Invoke callback RvMdmTermModifyMediaCB()	Multimedia Terminal Framework + User application	New parameters as received from the remote party will be passed through this callback. The user application will check whether these parameters can be supported and return True or False accordingly.

TLS

TLS is a security mechanism that operates on the Transport layer, on top of TCP Transport. By using TLS as connection transport, a SIP entity can send and receive data in a secure and authenticated manner.

TLS, together with the commonly-used Public Key Infrastructure certification distribution mechanism, achieves the following goals:

- Guarantees the identity of a remote computer
- Transmits messages to the remote computer in a secure encrypted manner
- Uses pairs of asymmetrical encryption keys to guarantee the identity of the remote computer. The public key of each remote computer is published in a certificate.

RFC 3261 defines the use of TLS as a transport mechanism by using the "sips:" scheme. When using the "sips:" scheme in a URI (or any other header that indicates the next hop of a message, such as Route, Via, etc.) *RFC 3261* mandates the transport to be TLS. For this reason TLS will not guarantee a secure delivery end-to-end, but only to the next hop.

The RADVISION SIP Stack uses an open source library called "openSSL" that provides TLS and encryption services. For more information about openSSL, see the openSSL project website at <http://www.openssl.org>.

IMPLEMENTATION

The TLS implementation is based on *RFC 3261* and is implemented as an add-on in the Multimedia Terminal Framework.

To support this feature:



Step 1: Install openSSL

The "openSSL" library must be downloaded and installed. Note that the RADVISION SIP Stack supports openSSL version ssl 9.7.D. Please follow the instructions given in the website <http://www.openssl.org>.

Example

To set openSSL in Windows, follow these instructions:

1. Go to <http://www.openssl.org>.
2. Download version 9.7.D.
3. Extract the package.

4. Follow directions in README and INSTALL.W32 to install openSSL.
5. Add to tools->options->directories, win32, include files the path to openssl/inc32.
6. Add to tools->options->directories, win32, library files the path to openssl/out32.dll.
7. Close and reopen msVC++.

Now you can compile the Multimedia Terminal Framework with TLS over openSSL.



Step 2: Compile with TLS flag

To support the TLS feature, Multimedia Terminal Framework must be compiled with a TLS compilation flag:

- To compile in Windows: Choose the TLS configuration in the build (debug or release).
- In operating systems other than Windows: Use the command line: "make all TLS=on".



Step 3: Implement callback RvIppTlsGetBufferCB

The following callback:

```
RvIppTlsGetBufferCB (
    IN   RvIppTlsBufferType      bufferType,
    OUT  RvChar                  *tlsBuffer,
    OUT  RvUInt32                *tlsBufferLen);
```

is invoked during the TLS initialization to receive TLS data from the user application. For example, IPP_TLS_SERVER_KEY_BUFFER.

Example

The following example is an implementation of this callback:

```
RvBool userSipTlsGetBuffer(
    IN RvIppTlsBufferType      bufferType,
    OUT RvChar*                *tlsBuffer,
    OUT RvUInt32*               *tlsBufferLen)
{
    static RvUInt32 caFileIndex = 0;
    BIO*          inKey       = NULL;
    BIO*          inCert      = NULL;
    EVP_PKEY*     pkey        = NULL;
    X509*         x509        = NULL;
    RvChar*        privKey     = NULL;
    RvChar*        keyEnd;     privKeyLen = 0;
    RvInt          cert;       cert = NULL;
    RvChar*        certEnd;    certLen = 0;
    RvInt          char;       caCertFileName[FILENAME_LEN];
    *tlsBufferLen = 0;

    /* loading key */
    switch (bufferType)
    {
        case IPP_TLS_SERVER_KEY_BUFFER:
            privKey = tlsBuffer;
            keyEnd = privKey;
            inKey = BIO_new(BIO_s_file_internal());
            if (BIO_read_filename(inKey, gw.keyTlsCfg.privateKeyFileName) <= 0)
                return rvFalse;

            pkey = PEM_read_bio_PrivateKey(inKey, NULL, NULL, NULL);
            if (NULL == pkey)
                return rvFalse;

            privKeyLen = i2d_PrivateKey(pkey, NULL);
            privKeyLen = i2d_PrivateKey(pkey, (unsigned char**) &keyEnd);
            BIO_free(inKey);
            *tlsBufferLen = privKeyLen;
            break;

        case IPP_TLS_SERVER_CA_BUFFER:
            cert = tlsBuffer;
            certEnd = cert;
            inCert = BIO_new(BIO_s_file_internal());
            if (BIO_read_filename(inCert, gw.keyTlsCfg.privateKeyFileName) <= 0)
                return rvFalse;

            x509 = PEM_read_bio_X509(inCert, NULL, NULL, NULL);
            if (NULL == x509)
                return rvFalse;

            certLen = i2d_X509(x509, NULL);
            certLen = i2d_X509(x509, (unsigned char**) &certEnd);
            BIO_free(inCert);
            *tlsBufferLen = certLen;
            break;

        case IPP_TLS_CA_BUFFER:
            /* get a CA filenames from the GW list of names */
            if (!getCAFFileName(gw.keyTlsCfg.caCertFileName[caFileIndex],

```

```

        - (char *)&caCertFileName)) || (caFileIndex == TLS_MAX_CA))

        return rvFalse;

        /* for adding the approving CA to the list of certificates */
        cert = llsBuffer;
        certEnd=cert;
        certLen    = 0;
        x509      = NULL;
        memset(cert,0,sizeof(cert));

        /* loading certificate */
        inCert=BIO_new(BIO_s_file_internal());

        if (BIO_read_filename(inCert, caCertFileName) <= 0)
            return rvFalse;

        x509=PEM_read_bio_X509(inCert,NULL,NULL,NULL);
        certLen = i2d_X509(x509,NULL);
        certLen = i2d_X509(x509,(unsigned char **)&certEnd);
        BIO_free(inCert);
        *llsBufferLen = certLen;
        caFileIndex++;
        break;

    } /* End Switch */

    return rvTrue;
}

```



Step 4: Implement callback RvIppSipTlsPostConnectionAssertionCB

The following callback:

```
RvIppSipTlsPostConnectionAssertionCB(
    IN  RvSipTransportConnectionHandle      hConnection,
    IN  RvSipTransportConnectionAppHandle   hAppConnection,
    IN  RvChar*                           strHostName,
    IN  RvSipMsgHandle                   hMsg,
    OUT RvBool*                          pbAsserted)
```

is invoked to override Stack default behavior in post-connection assertion. Once a connection has completed the handshake, make sure the certificate presented was indeed issued for the address to which the connection was made. The Stack may make this assertion automatically. However, if for some reason the application wants to override this assertion, it can implement this callback.

In the following example the application always overrides the Stack decision with a configuration parameter.

Example

This example is an implementation of the callback:

```
void userSipTlsPostConnectionAssertion(
    IN RvSipTransportConnectionHandle      hConnection,
    IN RvSipTransportConnectionAppHandle   hAppConnection,
    IN RvChar*                           strHostName,
    IN RvSipMsgHandle                   hMsg,
    OUT RvBool*                         pbAsserted)
{
    /* Behavior as defined in the configuration file.*/
    *pbAsserted = gw.transportTlsCfg.tlsPostConnectAssertFlag;
}
```



Step 5: Register TLS Callbacks

The user application must register TLS callbacks mentioned above by calling the following function:

```
void rvIppSipTlsRegisterExtClbks(RvIppSipTlsExtClbks* clbks)
```

The function is defined in `rvSipTlsApi.h` and should be called before `rvIppSipStackInitialize()`.

Example

```
RvIppSipTlsExtClbks     userSipTlsExtClbks =
{
    userSipTlsGetBuffer,
    userSipTlsPostConnectionAssertion
};

void userRegisterIppSipTlsExt(void)
{
    rvIppSipTlsRegisterExtClbks(&userSipTlsExtClbks);
}
```



Step 6: Configure TLS

The following parameters are related to the TLS configuration. TLS parameters are configured in two places: the Multimedia Terminal Framework configuration and the SIP Stack configuration.

Multimedia Terminal Framework TLS Parameters

The TLS parameters are configured as part of Multimedia Terminal Framework configuration, when calling `rvIppSipInitConfig()`. The values should be set in the structure `RvIppTransportTlsCfg`, which is included in the structure `RvIppSipPhoneCfg`:

- `StackTlsPort`—Secure port for TLS. Default: 5064.
- `CertDepth`—Defines the depth that an engine will consider legal in a certificate chain to which it is presented. Default: 5.
- `TlsMethod`—The SSL methods that will be used in the application's TLS engines. Default: 2.
(`RVSIP_TRANSPORT_TLS_METHOD_TLS_V1`)
- `PrivateKeyType`—Informs the engine of its private key. Default: 0 (`RVSIP_TRANSPORT_PRIVATE_KEY_TYPE_RSA_KEY`).
- `PrivateKeyFileName`—The relative path to the private key filename. For example: `D:\vxworks\evgeny-gx260.key-cert.pem`.
- `CaCertFileName`—The relative path to the file that contains the TLS trusted certificate authority. For example: `D:\vxworks\cacert.pem`.
- `TlsPostConnectAssertFlag`—A value of 1 indicates that the connection will be asserted even if it is invalid. A value of 0 indicates not to assert when the certificate is invalid.

A TLS attack can be performed as follows: Computer mallice.com holds the valid certificate of p.com and displays it in the handshake process. The certificate is valid so the handshake will be completed successfully. If the attack is preformed, the user might deliver data to an unauthenticated party. To prevent this, the SIP Stack compares parameters from within the certificate to the connection destination (usually the URI or the Route headers). If this comparison fails, the flag value overrides the decision of the SIP Stack and completes the TLS connection establishment.

SIP Stack TLS Parameters

The TLS parameters are configured as part of SIP Stack configuration by calling extension API `RvIppSipStackConfigCB()`:

- `MaxTlsSessions`—The maximum number of TLS sessions. A TLS session is the TLS equivalent on a TCP connection and contains TLS data required to manage the TLS connection. The SIP Stack will be able to handle a maximum of `maxTlsSessions` concurrent TLS connections. Default :10.

- `numOfTlsAddresses`—The number of TLS addresses on which the application wants to listen. Setting this number to 0 means that the application does not want to listen to any TLS addresses. It is the responsibility of the application to allocate two arrays with (`numOfTlsAddresses`) cells that contain addresses and corresponding ports. Default: 1.
- `localTlsAddresses`, `localTlsPorts`—Local TLS addresses on which the SIP Stack will listen. These arrays must be allocated according to the size given in `numOfTlsAddresses`. Each of the entries of `localTlsAddresses` must be allocated as well to contain the requested IP address. For example: 172.20.1.123; 5064.
- `numOfTlsEngines`—The maximum number of TLS engines. TLS engines are used to give a set of properties to a TLS connection. Note that the Multimedia Terminal Framework supports only one engine, which is rolled as both Client and Server engine.

Notes:

- The Multimedia Terminal Framework forces the `tcpEnabled` parameter to be 1.
- `transportType` can be either UDP or TCP.



Step 7: Initialize TLS

To initialize TLS, call the following function:

```
void rvIppSipTlsInitConfig(RvIppSipPhoneCfg* cfg)
```

This function is defined in `rvSipTlsApi.h`. It must be called before the Multimedia Terminal Framework is initialized (before calling `rvIppSipStackInitialize()`).

OUTGOING CALLS

The user application can decide on a call basis whether an outgoing call will be sent over TLS or not.

When implementing the callback `RvMdmTermMapDialStringToAddressCB()`, which is called for every outgoing call, see section Matching Digits to an Address, if the returned address (output parameter) includes a "sips:" scheme,

the call will be sent via TLS. If the scheme is a "sip:" scheme, the call will be sent over the configured transport type (UDP or TCP). If no scheme is included, the default is "sip:".

Note If the Transport type is configured as UDP, and the "address" parameter includes "sips:" scheme, the call will be sent over TLS (via TCP).

DISABLING TLS

When TLS is compiled in the Multimedia Terminal Framework, TLS can be disabled during initialization by setting the parameter numOfTlsAddresses = 0. In this case, the SIP Stack will be initialized successfully and the Multimedia Terminal Framework will run without TLS support.

SUPPORTED OPERATING SYSTEMS

This feature is supported in the following operating systems:

- Linux RedHat 9
- VxWorks 5.5
- Windows: 2003 Server, XP, 2000 Pro, NT
- Linux Monta Vista 3.1 (Embedded Linux)

OUT-OF-BAND DTMF

This feature enables the Terminal Framework to send and receive DTMF signals during a call. These signals are generated when the user presses a digit/DTMF key during a call. The DTMF signals can be relayed in band (as part of the RTP media stream, according to *RFC 2833*) or out of band, as a message in the specific signaling protocol.

IMPLEMENTATION

In SIP, DTMF OOB messages are sent in INFO messages containing Signal and Duration parameters in their message body, with the content type "application/dtmf-relay".

Sample SIP message

```
INFO sip:ip-229@172.20.53.229;transport=UDP SIP/2.0
From: "IPP-1164"<sip:ip-164@sip>;tag=104a4e0-7d0114ac-13c4-476d-
7a976b19-476d
To: "ip-229"<sip:ip-229@172.20.53.229>;tag=1121f28-e53514ac-13c4-
4f6757-2079de0c-4f6757
Call-ID: 104d440-7d0114ac-13c4-476d-29a1e790-476d@sip
CSeq: 2 INFO
Via: SIP/2.0/UDP 172.20.1.125:5060;rport;branch=z9hG4bK-4775-
11724bf-c6320ac
Max-Forwards: 70
Supported: replaces
Contact: <sip:ip-164@172.20.1.125:5060;transport=UDP>
Content-Type: application/dtmf-relay
Content-Length: 21

Signal=9
Duration=90
```

To support this feature:**Outgoing DTMF**

Configure the Multimedia Terminal Framework to enable DTMF OOB (set the flag outOfBandDtmf in the RvIppSipPhoneCfg structure to True). The DTMF digits will be sent with INFO messages.

Incoming DTMF

When a DTMF message is received, RvMdmTermStartSignalCB callback will be called with the following parameters:

pkg = "dg"
id = "d1" for digit 1, etc.
params = duration

CALL FORWARD

Call Forward (CFW) permits a user to have incoming calls addressed to the user redirected to another number. The user's ability to originate calls is unaffected by Call Forward.

Three types of Call Forwarding exist:

- Unconditional (CFU)—After CFW Unconditional has been activated, incoming calls are forwarded independently of the status of the endpoint.
- Busy (CFB)—After CFW Busy has been activated, incoming calls are forwarded only if the endpoint is busy, i.e., all lines are active.
- No Reply (CFNR)—After CFW No Reply has been activated, incoming calls are forwarded only if the endpoint does not answer before a pre-configured timeout.

ACTIVATION/DEACTIVATION

The user can activate or deactivate Call Forward for each type separately during runtime. Each type can be activated with different parameters (phone numbers). Once Call Forward is activated, incoming calls will be redirected according to the activated type.

Sequential Activations

Sequential activations for different types are accumulated. For example, if CFB is activated and an activation request for CNFR is received, CFW will be activated for both Busy and No Reply.

Sequential activations for the same type overrun the previous request. For example, if CFB is activated and a new activation request for CFB is received, the new request will overrun the previous request. This can be used to change activation parameters such as the phone number or timeout.

To support this feature:

1. To activate or deactivate CFW, send a CFW event each time the user wants to activate or deactivate CFW:

`pkg = "kf"`

`id = "ku" when the key is up and "kd" when the key is down.`

`Parameters:`

- `keyid = "cfwu" for CFU, "cfwb" for CFB, "cfnr" for CFNR`
- `activate = "on" for activation, "off" for deactivation.`
- `timeout = time in seconds, must be greater than zero.`

2. (Optional) Implement CFW callbacks.

The following callback is called when the CFW activation process has been completed:

```
void rvIppCfwActivateCompletedCB (
    RvIppTerminalHandle      term,
    RvIppCfwType             cfwType,
    char*                    cfwDestination,
    RvIppCfwReturnReasons   returnCode);
```

The following callback is called when the CFW deactivation process has been completed:

```
void rvIppCfwDeactivateCompletedCB (
    RvIppTerminalHandle      term,
    RvIppCfwType             cfwType,
    RvIppCfwReturnReasons   returnCode);
```

The `returnCode` parameter (of type `RvIppCfwReturnReasons`) may have one of the following values:

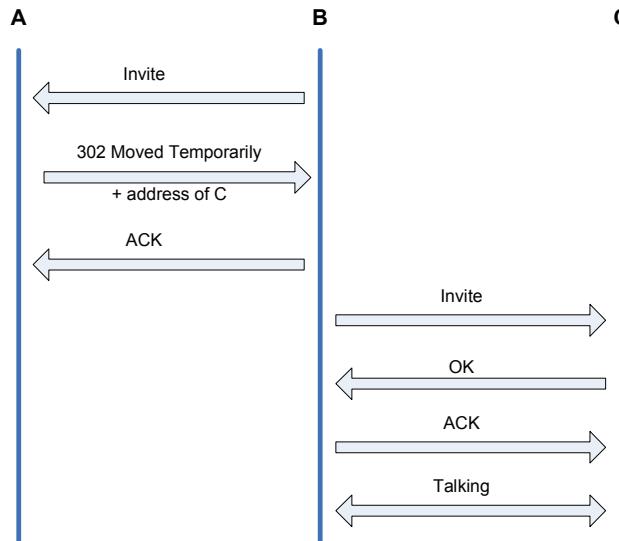
- `RV_IPP_CFW_SUCCESS`: The operation was completed successfully.
- `RV_IPP_CFW_INVALID_DEACTIVATION`: Deactivation request for a CFW type that was not activated.
- `RV_IPP_CFW_INVALID_PARAM`: One of the parameters is not valid.

- ❑ RV_IPP_CFW_ADDRESS_NOT_FOUND: The mapping of a phone number failed during activation.
- ❑ RV_IPP_CFW_NOT_ALLOWED: The operation is not allowed in this state.
- ❑ RV_IPP_CFW_CANCELLED_BY_USER: The operation was canceled by the user (for example, by accepting an incoming call during CFW activation).

SIP IMPLEMENTATION

When an Invite is received on the party on which CFW is activated, it will reply with message 302 Moved Temporarily with the new address (according to *RFC 3261* section 21.3.3).

When an Invite is sent with a reply of 302 Moved Temporarily, the endpoint will initiate a new Invite to the new destination received in the 302 message.



Activating CFW

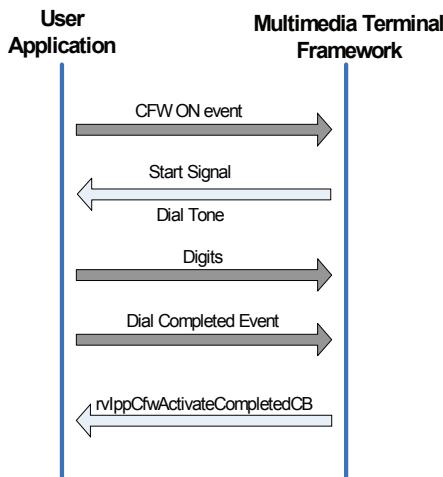
User Step	User Indication	User Application Indication
1. Press CFW (activate="on")	Hear dial tone	RvIppMdmPreProcessEventCB is called with terminal state (RvCCTerminalState) = RV_CCTERMINAL_CFW_ACTIVATING_STATE
2. Press digits	Dial tone stops	rvIppCfwActivateCompletedCB is called with reason code according to the process status

If the activation process succeeds, rvIppCfwActivateCompletedCB is called with the reason code RV_IPP_CFW_SUCCESS.

If the activation process fails:

- A warning tone is heard. The user should press Line or CFW to disconnect.
- rvIppCfwActivateCompletedCB is called with the appropriate reason code.

The following diagram describes the flow of the activation process between the user application and the Multimedia Terminal Framework.



Deactivating CFW

To deactivate the CFW type, press CFW (activate="off"). The callback rvIppCfwDeactivateCompletedCB will be called with the relevant reason code.

SESSION TIMER

The Session Timer extension enables determining the duration time of a call. A Session-Expires header included in the 2xx response of an initial INVITE determines the session duration. Periodic refresh enables extending the duration of the call and allows both User Agents (UAs) and proxies to determine if the SIP session is still active. Refreshing a SIP session in the Terminal Framework is done through a re-INVITE. The Session-Expires header in the 2xx response also indicates which side will be the refresher—in other words, which side will be responsible for generating the refresh request. The refresher can be the initiator or the receiver of the call. The refresher should generate a refresh request (using re-INVITE) before the session expires. If no refresh request is sent or received before the session expires, or if the refresh request is rejected, both parties should send BYE. The Session Timer feature defines two new headers (Session-Expires and Min-SE) and a new response code of 422. The Session-Expires header conveys the lifetime of the session, the Min-SE header conveys the minimum value for the Session Timer, and the 422 response code indicates that the Session Timer duration is too small. If the Min-SE header is missing, the minimum value for the Session Timer is 90 seconds by default, according to the Session Timer standard.

To support this feature:

Configure the following parameters:

- **sessionExpires**

The time at which an element will consider the call timed out, if no successful INVITE transaction occurs beforehand. This value is inserted into every INVITE in the Session-Expires header unless it was configured to 0. If the "timer" option tag is not part of the supported list, the sessionExpires value will be ignored.

Default: 1800 seconds

- **minSE**

The minimum value for the session interval that the application is willing to accept. If the application does not set this parameter, the minSE value is set to the default value of 90 seconds according to the Session Timer draft. Also, the Min-SE header will not be present in the sent requests (except for a request, following a 422 response). However, if the application set this parameter to 90 or any other value, the Min-SE header will appear in any sent request.

Default: 90

To disable this feature:

To disable the Session Timer, set the configuration parameter sessionExpires to 0.

TIMERS

When the SIP Stack is configured to support the timer extension, the Initial INVITE request, and every refresh will automatically include the Session-Expires header with the configured sessionExpires value. If a minSE value has been configured, a Min-SE header will also be included. After a 2xx response is received with the final Session-Expires value and the call is connected, the SIP Stack sets a timer for the session. A timer is set for each party of the call. On the refresher side, the timer is set to a default of the final sessionExpires/2. This timer alerts the call-leg to send a refresh request. You can change the alert time default value using the RvSipCallLegSessionTimerSetAlertTime() function. On the non-refresher side, a timer is set to sessionExpires-min(32,sessionExpires/3) according to the Session Timer standard. When this timer expires, BYE is sent.

SDP

When an INVITE is sent as a result of a Session Timer refresh the version value (in the "0" line) is NOT incremented, so that the receiving side knows that this Invite does not contain new SDP.

IMPLEMENTATION

The Session Timer feature supports RFC 4028. Refresh is implemented by Re-Invite messages.

PRACK

The Provisional Response Ack (PRACK) message ensures the reliability of provisional responses. To achieve reliability for provisional responses, these responses are retransmitted by the User Agent until a PRACK message is received. The PRACK request plays the same role as ACK, but for provisional responses. The reliable provisional response must contain a Require: 100rel header. PRACK also allows the media negotiation to be performed at an earlier stage, before the call is answered. This helps avoid media clipping as might happen with no PRACK. If a reply for media negotiation is received when the user answers the call (with OK), media may not be connected quickly enough before the user starts talking. Upon reception of a reliable provisional response the Multimedia Terminal Framework will reply with a PRACK message. The Multimedia Terminal Framework does not initiate reliable provisional responses.

Figure 8-16 shows the flow of the SIP message when Ringing with a request for PRACK (meaning, header "Required: 100rel" is included) is received:

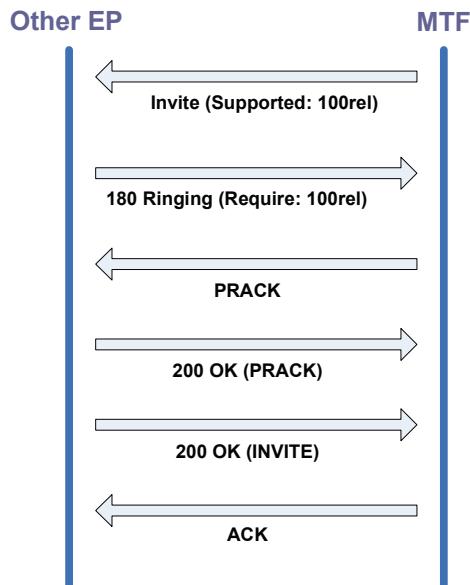


Figure 8-16 PRACK Call Flow

EARLY MEDIA

Early Media refers to media (e.g., audio and video) that is exchanged before a particular session is accepted by the called user. Typical examples of Early Media generated by the callee are ringing tone and announcements (e.g., queuing status). The Multimedia Terminal Framework implementation of Early Media enables a callee to connect media before it sends a final INVITE response. The callee may send a 183—Session Progress response to an INVITE with media description. Upon reception of this response the Multimedia Terminal Framework will connect the media immediately instead of waiting to a final INVITE response. The Multimedia Terminal Framework will accept media descriptions that are attached to other provisional responses and to OK (PRACK) messages, but in this case the media will be connected only after reception of a final INVITE response.

Figure 8-17 shows the call flow of Early Media. When a "session Progress 183" message is received with SDP, the media will be connected.

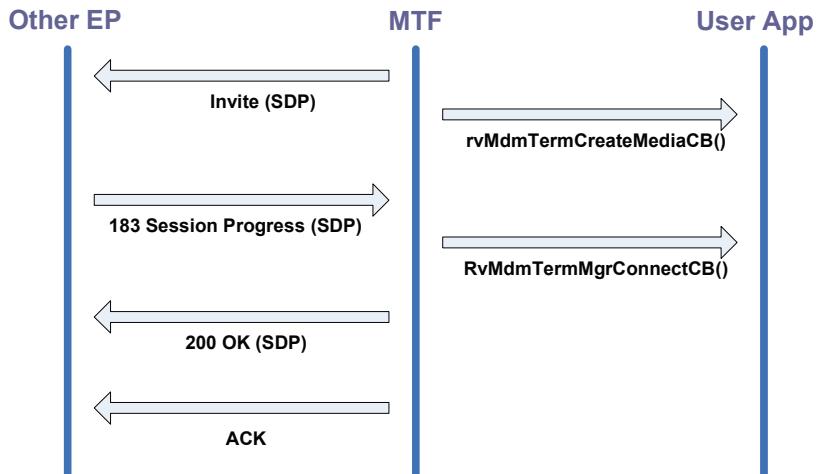


Figure 8-17 Early Media Call Flow

DISTINCTIVE RINGING

When the messages `Invite` and `180—Ringing` are received, the Multimedia Terminal Framework will apply on termination the ringing tone defined in the `Alert-info` header if it exists, instead of the default tone. The alert info in an `INVITE` message defines the callee ringing tone. The alert info in the `180` message defines the ring back tone played by the caller. The ringing tone defined in the `Alert-info` header is played locally. The value of `Alert-Info` header is passed to user application as is. It is the responsibility of the user endpoint to recognize the new parameter and play the ringing/ringback tone it specifies.

To support this feature:

When implementing the user callback `RvMdmTermStartSignalCB()`, check for an additional parameter for the Ringing signal (`pkg = "ind", id = "is"`) and the Ringback signal (`pkg="cg", id="rt"`):
`"distRing"`.

The value of this parameter is a string that contains the distinctive ringing as received in the `Alert-Info` header.

Figure 8-18 shows the flow of Distinctive Ringing.

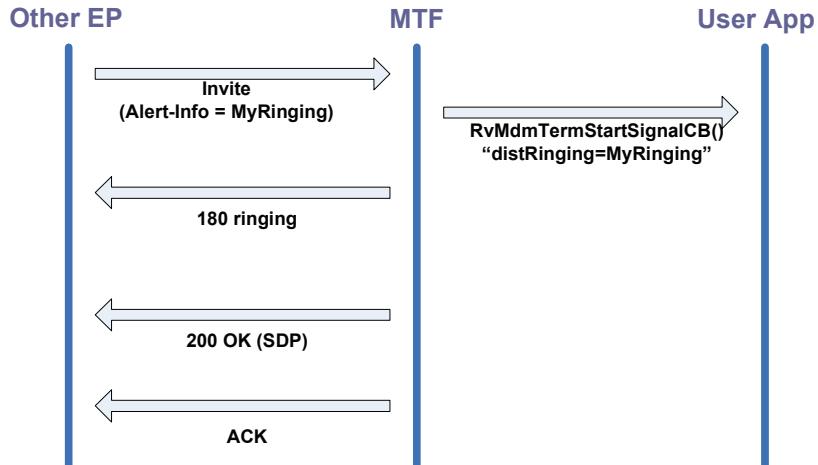


Figure 8-18 Distinctive Ringing Call Flow

ADVANCED DNS

The SIP Stack uses DNS procedures to allow the client to resolve a SIP Uniform Resource Identifier (URI) into the IP address, port, and transport protocol of the next hop to contact. By default, the SIP Stack is compiled with basic DNS functionality that includes A/AAAA queries to resolve IP addresses of hosts given as a domain name. The SIP Stack enables a second mode of operation, an enhanced DNS mode. When using the enhanced DNS mode, the behavior of the SIP Stack will comply with the client DNS procedures defined in RFC 3263 and RFC 3824. This includes usage of NAPTR and SRV DNS queries, and the ability to maintain a list of resolved addresses that the application will be able to try one after the other, in case of a send failure.

The Multimedia Terminal Framework enables the user application to use the Advanced DNS supported by the SIP Stack by performing the steps described below. The Multimedia Terminal Framework supports Advanced DNS for Invite and Register messages only. The most common use of Advanced DNS in the Multimedia Terminal Framework is for locating SIP Servers (RFC 3263).

IMPLEMENTATION

Before sending a request, SIP Stack obtains the destination host name from the request message. If the SIP Stack is compiled without Enhanced DNS features, it will try to resolve the host using a simple A/AAAA DNS query. If the SIP

Stack is compiled with Enhanced DNS support, the SIP Stack will try to obtain the transport, port and IP of the host name as specified in RFC 3263 and RFC 3824. The replies from the DNS Server are kept in a DNS List object. The DNS List holds the following sub-lists:

1. SRV list—the result of the NAPTR query.
2. Host list—the result of an SRV query applied on the first element of the SRV list. After the DNS SRV query is made, the first element in the SRV list is moved to the used SRV element member of the DNS list.
3. IP addresses list—the result of an A/AAAA query applied on the first element of the Host list. After the A/AAAA query is made, the first element in the Host list is moved to the used host element member of the DNS list.

When the DNS querying process has ended, every DNS sub-list (for example, hosts) will contain all the results retrieved from the DNS, excluding the result for which the next sub-list was constructed. Finally, the transaction will pop an IP from the IP sub-list and use it to send the request.

The transaction sends a request to the first IP address found in the IP list. If a failure occurs, the Multimedia Terminal Framework will notify the SIP Stack to send the request to the next element in the list. Once all IP addresses for a specific host are exhausted, the *transaction* will pop a host from the host sub-list, and the DNS will be queried for that host in the list.

The same procedure applies when all the hosts are exhausted, in which case the DNS will pop the next SRV record and query the DNS for hosts, and so on.

RFC 3263

RFC 3263 specifies the usage of three DNS record types:

1. NAPTR—used to determine the transport for a specific domain.
2. SRV—used to determine the port and transport of a specific SIP service.
3. A (or AAAA for IPv6)—used to determine the IPv4/IPv6 address of a specific host.

RFC 3824

RFC 3824 specifies the usage of ENUM NAPTR records for relating between E.164 numbers and URIs.

Note For more information on Advanced DNS implementation of the SIP Stack, see the SIP Stack Programmer Guide-> Working with DNS.

To support this feature:

Step 1: Compilation

Make sure the following libraries are compiled with the compilation flag `RV_DNS_ENHANCED_FEATURES_SUPPORT`: MTF, SIP, Ads, and Common. The Multimedia Terminal Framework is compiled with this flag by default.

Step 2: DNS Server Configuration

There are two ways to configure the DNS Server:

1. Operating System configuration

Note When working in Windows, the SIP Stack tries to fetch the DNS servers and DNS suffixes from the Windows configuration. To verify that your system is configured correctly, look at the output of the "ipconfig/all" command and make sure that the "DNS Suffix" item is configured both in the "Windows IP Configuration" section and in the "Adapter Connection Specific" section. The "DNS Servers" should also be configured.

2. SIP Stack configuration

pDnsServers

An array allocated by the application that holds DNS servers that the SIP Stack will use to resolve names. You **must** set the size of this array in the `numOfDnsServers` configuration parameter. If no DNS servers are listed, the

SIP Stack will use the DNS servers that the operating system is configured to use when the SIP Stack is initialized. If the port number inside any of the addresses is set to 0, the port will be converted to the default "well known" DNS port (53).

Default value: NULL

□ **numOfDnsServers**

The number of DNS servers in pDnsServers.

Default value: UNDEFINED

□ **pDnsDomains**

A list of domains for the DNS search. The SIP Stack provides Domain Suffix Search Order capability. The Domain Suffix Search Order specifies the DNS domain suffixes to be appended to the host names during name resolution. When attempting to resolve a fully qualified domain name (FQDN) from a host that includes a name only, the system will first append the local domain name to the host name and will query the DNS servers. If this is not successful, the system will use the Domain Suffix list to create additional FQDNs in the order listed and will query DNS servers for each. If you do not supply a domain list, the domain list will be set according to the computer configuration. The domain list is given as an array of string pointers. You **must** set the size of the list in the *numOfDnsDomains* configuration parameter.

Default value: NULL—the domain will be taken from the operating system.

□ **numOfDnsDomains**

The size of the array given in pDnsDomains.

Default value: UNDEFINED

□ **maxDnsDomains**

The maximum number of DNS domains the application will use concurrently. The application can change the list of DNS servers at runtime. It must specify the maximum number of DNS domain it is expected to use in the maxDnsDomains parameter.

Default value: UNDEFINED

□ **maxDnsServers**

The maximum number of DNS servers that the application will use concurrently. The application can change the list of DNS servers on runtime. It must specify the maximum number of DNS servers it is expected to use in the maxDnsServers parameter.

Default value: UNDEFINED

Step 3: Multimedia Terminal Framework parameters

Establishing a new call:

If the destination address is an NAPTR record, do the following:

1. In implementing the callback
rvMdmTermMgrMapDialStringToAddressCB():
 - Make sure the destination address does not include a scheme (for example, do not add "sip:" or "sips:" to the destination address)
 - Make sure the destination address does not include a port number (for example, do not add ":5070")
2. Set the configuration parameter "transportType = RVSIP_TRANSPORT_UNDEFINED"

If the destination address is an SRV record, do the following:

1. In implementing the callback
rvMdmTermMgrMapDialStringToAddressCB():
 - Add a scheme to the destination address, if needed (for example, add "sips:" to the destination address)
 - Make sure the destination address does not include a port number (for example, do not add ":5070")
2. Set the configuration parameter "transportType" to the required type (for example, RVSIP_TRANSPORT_TCP or RVSIP_TRANSPORT_UDP)

If the destination address is A/AAAA record, do the following:

1. In implementing the callback
rvMdmTermMgrMapDialStringToAddressCB():
 - ❑ Add a scheme to the destination address, if needed (for example, add "sips:" to the destination address)
 - ❑ Add a port number to the destination address, if needed (for example, add ":5070" to destination address)
2. Set the configuration parameter "transportType" to the required type (for example, RVSIP_TRANSPORT_TCP or RVSIP_TRANSPORT_UDP)

Sending a Register message to the Registrar:

If the domain name is an NAPTR record, do the following:

- Set the configuration parameter "RegistrarPort= RVSIP_TRANSPORT_UNDEFINED"
- Set the configuration parameter "transportType = RVSIP_TRANSPORT_UNDEFINED"

If the domain name is an SRV record, do the following:

- Set the configuration parameter "RegistrarPort= RVSIP_TRANSPORT_UNDEFINED"
- Set the configuration parameter "transportType" to the required type (for example RVSIP_TRANSPORT_TCP or RVSIP_TRANSPORT_UDP)

If the domain name is A/AAAA record, do the following:

- Set the configuration parameter "RegistrarPort" to the required port number
- Set the configuration parameter "transportType" to the required type (for example RVSIP_TRANSPORT_TCP or RVSIP_TRANSPORT_UDP)

Protocol-Specific Features

PART 4: MEDIA ADD-ON

9

MEDIA ADD-ON

9.1 WHAT'S IN THIS CHAPTER

The Media Add-on enables enhanced media features by seamlessly integrating with RADVISION's Media Framework and Advanced RTP Toolkit. This chapter includes the following:

- Overview
- Working with the Media Add-on
- Architecture
- Media Framework
- Supported Codecs and Features

9.2 OVERVIEW

The Media Add-on handles the actual media processing and offers a wide range of media features. This includes both codec integration and additional algorithms necessary for the creation of a rich media application. The Media Layer is built as an OS-independent Media Framework, enabling the building of media processing chains that handle media flows within multimedia terminals. These media processing chains may include codecs, pre-filtering algorithms, and post-filtering algorithms, using RADVISION-supplied components, pre-integrated third-party components, or application-specific components added independently by users of the Multimedia Terminal Framework. Media Add-on components may be either software or hardware components.

Note The Media Add-on saves application developers the effort of media integration as described in the section [Step 4: Integrating Media](#).

MEDIA PROCESSING ALGORITHMS

For the creation of a high-quality video telephony application, codecs alone are not sufficient; a set of media processing algorithms is also required. The Media Add-on includes integrated building blocks for the required media processing algorithms, such as adaptive jitter buffering, lip synchronization, and echo cancellation.

CODECS

Audio and video codecs must be integrated with the Multimedia Terminal Framework itself. Multimedia Terminal Framework supports two possible modes of operation:

- A thin layer, which enables easy integration with third-party codecs that meet VoIP requirements.
- Pre-integrated codecs supplied by RADVISION, which eliminate the extra work involved in integrating codecs into a multimedia terminal product.

9.3 WORKING WITH THE MEDIA ADD-ON

Perform the following steps to work with the Media Add-on in the Multimedia Terminal Framework:

CONFIGURATION PARAMETERS

DefaultDevice

This parameter is currently not in use.

RTPPortBase

This parameter indicates the number to be used as a basis for the port number, when opening the RTP port. The first port that is opened will have this number, while subsequent ports will be increased by two (for example, if this parameter is set to 7000, then the first port will be 7000, the second 7002, etc.). Odd numbers are reserved for RTCP ports respectively (7001, 7003, etc.).

Default value: 7000

RTPPortRange

Maximum number of RTP ports that can be opened (for all calls). This means that last port number will be RTPPortBase+ RTPPortRange.

Default value: 8

VideoFrameSize

Video frame resolution, the following resolutions are currently supported:

MF_RESOLUTION_QCIF and MF_RESOLUTION_CIF.

Default value: MF_RESOLUTION_CIF

VideoFrameRate

Number of Video frames per second.

Default value: 30

RecvAudioPolling_ms

Polling interval for RTP socket receiving audio stream (milliseconds).

Default value: 20

PlayAudioPolling_ms

Polling interval for RTP socket sending audio stream (milliseconds). Also used as polling interval for mixing mechanism (in Conference). This is relevant only when working with RADVISION G.711 codec.

Default value: 20

RecvVideoPolling_ms

Polling interval for RTP socket receiving video stream (milliseconds).

Default value: 20

PlayVideoPolling_ms

Polling interval for RTP socket sending video stream (milliseconds). Also used as polling interval for mixing mechanism (in Conference).

Default value: 30

MEDIA CALLBACK

There is one callback the user application should implement when working with the Media Add-on:

```
RvBool (*RvMtfCreateMediaNegotiateCB) (
    INOUT RvSdpMsg*      msgLocal,
    INOUT RvSdpMsg*      msgRemote,
    IN     RvSdpMsg*      msgAvailable,
```

```
IN     void*      userData);
```

The purpose of this callback is to enable the user application to choose the preferred media capabilities during media negotiation. This callback is called when:

- A new call is established and a new media stream should be created
- A new outgoing call is established, and a response (200 OK) was received from the remote party. In this case we already sent an Invite and created new one-directional media stream/s. When a response of 200 OK was received from the remote party, this callback is called to indicate the stream/s that need to be modified in line with the remote party's capabilities.
- During a connected call, when one of the parties wants to modify the existing media stream/s:
 - If initiated by the local party: This callback will be called before Re-Invite is sent.
 - If initiated by remote party: This callback will be called after Re-Invite was received.

Implementation of this callback should be done as follows:

1. Go through the list of msgLocal and msgAvailable and choose the local capabilities that are available and supported.
2. Go through the list of msgRemote and msgAvailable and choose common capabilities.
3. Set the chosen media capabilities in msgLocal and msgRemote.

msgLocal—This parameter contains all media capabilities of the local party as they were configured by the user application during initialization of the Multimedia Terminal Framework. The user application should modify this object so that it will contain only the media capabilities that were chosen (this means removing capabilities that were not chosen and leaving only capabilities that were chosen).

msgRemote—This parameter contains media capabilities of remote party, as received in the signaling message (Invite or Re-Invite). During establishment of the outgoing call it will be NULL.

msgAvailable—The media capabilities currently available for the local party. If there are limited resources and during runtime, some of the configured capabilities may not be available all the time. These capabilities will be obtained by the Multimedia Terminal Framework by calling the callback RvMtfGetMediaAvailableCB() before this callback is called.

UserData—User data previously passed to the Multimedia Terminal Framework by calling rvMdmTermSetUserData().

9.4 ARCHITECTURE

Figure 9-1 depicts the Multimedia Terminal Framework application without the Media Add-on, in which all media processing is done by the user application. The user application also needs to implement Multimedia Terminal Framework media callbacks. The Multimedia Terminal Framework will call these callbacks whenever the user application is required to perform a media operation (for example, opening an RTP stream, connecting an RTP stream to a Handset, etc.).

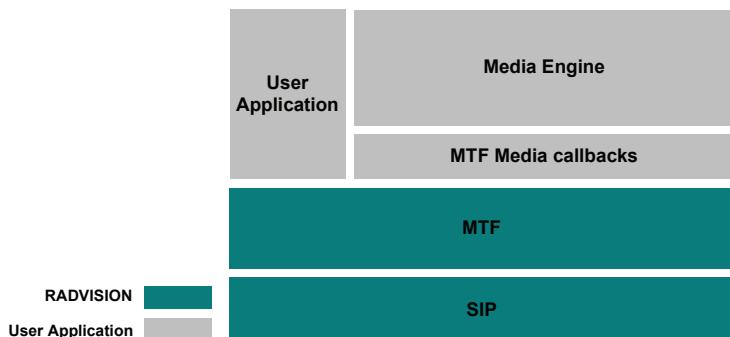


Figure 9-1 Multimedia Terminal Framework without Media Add-on

Figure 9-2 illustrates how the Media Add-on is integrated with the Multimedia Terminal Framework:

- **Advanced RTP (ARTP)**

RADVISION Advanced RTP Stack responsible for sending and receiving realtime media over IP.

- **Media Framework (MF)**

This library is the media engine responsible for processing media buffers received from the network as RTP packets and from audio/video devices (e.g., a microphone or a camera).

The buffers are processed as linear data paths (see details below). The Media Framework library includes integrations for third-party codecs vendors* and audio/video devices, as well as integration with the RADVISION Advanced RTP Stack.

- **Third-Party Codecs**

The Media Framework library can be integrated with different codecs vendors. Currently, the Media Framework is integrated with TeamSpirit™ SDK* of SPIRIT DSP.

- **Multimedia Terminal Framework Media callbacks implementation**

This component includes implementation for Multimedia Terminal Framework media callbacks, responsible for initializing and operating the Media Framework library whenever the user application is required to perform media processing.

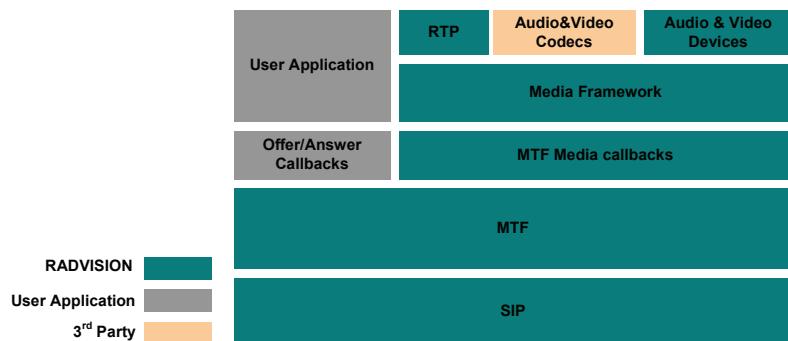


Figure 9-2 *Multimedia Terminal Framework with Media Add-on*

9.5 MEDIA FRAMEWORK

The RADVISION Media Framework is a set of software development tools intended for the development of media-capable software and hardware. The RADVISION Media Framework provides the ability to create media paths that are built from a set of algorithms and components.

The Media Framework library includes the following components:

MEDIA FRAMEWORK NODES

Nodes are responsible for buffer processing, filtering, analyzing, etc. Each node is responsible for a specific process. This means that each component that requires modification in the received buffer is a node.

The Media Framework library includes the following set of nodes:

- Audio/video codecs—Encoders and decoders are nodes. This requires integration with third-party vendors.*
- Audio/video devices (Handset or Camera)—requires integration with device drivers
- Media algorithms (for example, echo cancellation)—requires integration with third-party vendors*
- Advanced RTP Stack for sending and receiving RTP packets—includes integration with the RADVISION RTP Stack.

Each node implements the following operations:

- Insert()—This operation is done on incoming buffers.
- Extract()—This operation is performed on the processed buffer.

MEDIA FRAMEWORK CHAINS

Media processing chains are based on the set of nodes described above. Processing of media buffers is done as a linear data path where buffers are passed from one node to another. In each chain, one or more nodes are asynchronous while the remainder are synchronous so that each node performs its processing and passes the buffer to the next node in the chain.

EXAMPLE: BASIC CALL

Figure 9-3 illustrates a basic call as two linear data paths:

- Incoming call (upper diagram)—the chain starts in an RTP socket wrapped by an RTP Receiver node. It passes through the chain that contains a jitter buffer and decoder node, and ends in a speaker device wrapped by a Speaker Proxy node.

- Outgoing call (lower diagram)—the chain starts in a Microphone device wrapped with a Microphone Receiver node, passes through the chain and the encoder node, and ends in an RTP socket wrapped by an RTP Proxy node.

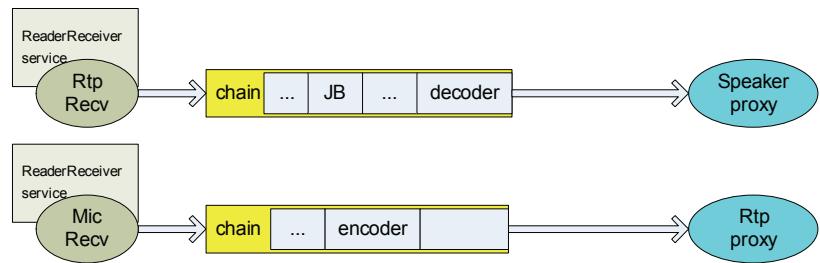


Figure 9-3 *Media Chains*

*) Please contact a RADVISION representative for more information on codecs vendors.

9.6 SUPPORTED CODECS AND FEATURES

The following audio codecs are supported with TeamSpiritTM SDK*:

- G.711 alaw
- G.711 m-law
- G.723.1 (with and without Silence Suppression)
- G.729a
- G.729ab

The following video codecs are supported with TeamSpiritTM SDK:

- H.263
- H.264
- MPEG4

The following media algorithms are supported:

- Jitter Buffer (JB)—for audio, it is integrated with the decoder of TeamSpiritTM SDK. For video, it uses RADVISION's.
- Echo Cancellation—integrated with TeamSpiritTM SDK.
- Packet Lost Concealment (PLC)—Audio only, integrated with the decoder of TeamSpiritTM SDK.

The following signaling features are also supported in the Media Add-on:

- Basic Audio Call
- Basic Video Call
- Attended Transfer
- Blind Transfer
- Call Forward
- Video Conference
- Audio Conference
- Mute
- Hold
- Dynamic Media Change

Supported Codecs and Features

10

OPTIMIZATIONS

10.1 WHAT'S IN THIS CHAPTER

This chapter explains how the user application can optimize the amount of allocated memory used by the Multimedia Terminal Framework and its performance:

- Introduction
- Reducing Footprint
- Minimizing Memory
- Optimizing Performance

10.2 INTRODUCTION

The Multimedia Terminal Framework enables customers to develop client applications of the following types:

- Terminal applications, such as IP Phones
- Terminal manager applications, such as Residential Gateways or IAD applications, which handle large numbers of terminals

IP Phone applications are usually expected to run on embedded operating systems with limited memory resources. Therefore, they will want to reduce the amount of memory allocated by the Multimedia Terminal Framework.

Residential Gateways or IAD applications may prefer better performance over memory reduction.

This chapter describes three types of optimizations:

- Reducing Footprint
- Minimizing Memory
- Optimizing Performance

10.3 REDUCING FOOTPRINT

For a minimal footprint, compile the Multimedia Terminal Framework in Release mode with a minimal number of logs or no logs at all.

Release Mode Compilation

During development, you generally build and test with a debug build of the Multimedia Terminal Framework. However, a debug build adds substantial debug information into the compilation results and hence increases the project footprint. To reduce footprint, it is important to compile the Multimedia Terminal Framework in Release mode. To compile in Release mode, add “release=on” to your make command, or choose a release configuration on Windows platforms. For more information, see the Readme file of your operating system.

Reduced Log Compilation

The Multimedia Terminal Framework contains a considerable amount of logging information that informs the application about the activities of the Toolkit. Although logging information is important, each log line in the code increases the code footprint. Therefore, to reduce footprint, it is important to remove as many log lines as possible during compilation. Removing all log prints will, of course, have the best impact on the footprint. To remove all logging information, add “nolog=on” to your make command.

The application can also control the compiled log messages by setting the “RV_LOGMASK” compilation flag found in the *rvusrconfig.h* file to the specific log messages required.

For example:

- `#define RV_LOGMASK RV_LOGLEVEL_NONE—`
removes all logging
- `#define`
`RV_LOGMASK RV_LOGLEVEL_EXCEP+RV_LOGLEV`
`EL_ERROR—enables only the exception and error logs`

For more information on logging, see the [Logging](#) chapter.

10.4 MINIMIZING MEMORY

Multimedia Terminal Framework

By default, the Multimedia Terminal Framework allocates its memory dynamically: During runtime, whenever the Multimedia Terminal Framework requires memory, it asks the operating system to allocate the exact amount of memory. Thus, it consumes a minimal amount of memory required for its

operations without allocating additional memory for memory management. After startup, the Multimedia Terminal Framework allocates additional memory for each termination registration and for each call.

SIP Stack

All memory is allocated statically during the initialization process and is managed by the SIP Stack. The SIP Stack does not allocate memory dynamically. The amount of memory allocated is specified in the user configuration, as explained below.

The Multimedia Terminal Framework sets the following SIP Stack parameters to their default values during startup. The SIP Stack parameters described here affect the amount of memory allocated by SIP Stack:

- maxTransactions = maxCallLegs * 5
- maxConnections = 25
- messagePoolNumofPages = 100
- messagePoolPageSize = 2048
- generalPoolNumofPages = 100
- generalPoolPageSize = 2048
- sendReceiveBufferSize = 4096
- maxSubscriptions = 5

You can decrease the amount of allocated memory by adjusting the SIP Stack parameters to the needs of your application. This is done by implementing the callback *RvIppSipStackConfigCB* (for more information, see the chapter [Extensibility](#), section [SIP Control](#)). For more information about parameters and memory management of the SIP Stack, see the *SIP Protocol Toolkit Programmer Guide*.

SDP

The SDP Stack allocates its memory dynamically. It is using the Multimedia Terminal Framework allocator (either the default allocator or the Cache Allocator, as configured in the Multimedia Terminal Framework).

10.5 OPTIMIZING PERFORMANCE

To optimize performance, it is recommended to do the following:

1. Compile all projects in Release mode, without logs (see the Readme file of your operating system for instructions).
2. Compile the Multimedia Terminal Framework project with the Cache Allocator.

Compiling the Multimedia Terminal Framework with the Cache Allocator

By default, the Multimedia Terminal Framework allocates its memory dynamically. Thus, it consumes a minimal amount of memory; whenever memory is needed, the Multimedia Terminal Framework allocates the exact amount without allocating additional memory for memory management. However, dynamic allocations use a large number of system calls, which affects performance.

If you want to improve the performance of the Multimedia Terminal Framework, or decrease the number of times it uses system calls to allocate memory, you can change the allocation mode to semi-static allocation using the Cache Allocator module.

The Cache Allocator allocates memory blocks during startup. These blocks are used and reused (cached) during runtime. In addition, blocks are released and new ones allocated depending on Multimedia Terminal Framework usage, thereby significantly reducing the number of system calls.

For more information about the Cache Allocator, see the appendix [Cache Allocator](#).

To use the Cache Allocator in the Multimedia Terminal Framework, you should compile the Multimedia Terminal Framework project with the compilation flag `RV_MTF_USE_CACHE_ALLOCATOR`.

In addition, for better performance when working with the Cache Allocator, you can compile with the compilation flag `RV_MTF_PERFORMANCE_ON`. When this flag is compiled, the Cache Allocator will not check for memory corruption.

Note Compiling the Multimedia Terminal Framework with the Cache Allocator increases memory consumption by ~150K, and improves performance by ~20%.

APPENDIX A

CACHE ALLOCATOR

INTRODUCTION

Multimedia Terminal Framework version 2.5 features a new memory allocator, called Cache Allocator. The new allocator caches (reuses) freed memory blocks so as to significantly reduce the number of calls made to the memory allocator of the operating system. The Cache Allocator keeps an adaptive, relatively small, memory footprint that automatically adjusts itself according to the needs of the Multimedia Terminal Framework, with the ability to handle different loads at runtime.

The Cache Allocator holds a cache table that contains 16 entries, each of which is responsible for memory blocks of a certain size range. Each table entry has a queue of cached memory blocks:

Table Entry 0: 0 - 127 bytes

Table Entry 1: 128 - 255 bytes

Table Entry 2: 256 - 383 bytes

...

Table entry 15: 1920..2047 bytes

Using the configuration shown above, the Cache Allocator is good for all block allocations under 2KB. The allocation of blocks larger than this limit will be performed by the default allocator. Hence, the allocation of large blocks through the Cache Allocator is not recommended (a pool should be used instead).

Cache Allocator behavior can be adjusted to the usage of your application. During initialization, the Cache Allocator allocates memory blocks for its cache. During runtime, it allocates new blocks or releases blocks according to memory consumption. This behavior is determined by the value of the following configuration parameters:

- **Initial limit (il)**—The initial size limit of the cache. This value should be anywhere between the lower and the upper limit.
- **Upper Limit (ul)**—The maximum size to keep in the cache. This value may change during runtime to adjust the cache size to memory consumption by the application.
- **Lower Limit (ll)**—Indicates the minimum size to keep in the cache. This value may change during runtime to adjust the cache size to memory consumption by the application.
- **Initial value (iv)**—Indicates how many memory blocks should be allocated during initialization.
- **Increase Trigger Counter (itc) and Increase by (ib)**—These parameters indicate when more blocks should be allocated, and howmany. Whenever the cache is empty and new allocations are made, new memory blocks are allocated using the default (slow) allocator. Each time this event happens, it is counted by an internal counter. When the counter reaches the value of "itc" the cache allocator increases the cache size of this block. The next time memory allocation is required, it will be allocated from the cache. This value is increased by a value of "ib". You can set these values to zero to cancel the limit adaptation by the Cache Manager.

Note The "itc" parameter will be added in the next version.

- **Decrease Trigger Counter (dtc) and Decrease by (db)**—These parameters indicate when to release memory blocks, and howmany should be released. Each time a memory block is freed into the cache, a counter is increased. When this counter reaches the value of "dtc", the Cache Allocator decreases the cache size of this block by the value of "db". You set these values to zero to cancel the limit adaptation of the cache manager.

COMPILE-TIME CONFIGURATION FLAGS

The compile-time configuration flags are listed below.

`RV_MTF_PERFORMANCE_ON`

When this flag is **not** defined, the Multimedia Terminal Framework will perform run-time boundary checks that validate the integrity of the memory blocks that are returned into the cache. Stack canaries will be placed at both ends of the buffer to detect memory under/overruns, and the entire buffer space is initialized to have an arbitrary value. Remove this flag to detect memory corruptions. Note that when this flag is compiled, it will increase performance.

Introduction

APPENDIX B

WHAT'S NEW IN VERSION 1.4.2

This appendix describes additions and modifications in version 1.4.2 as compared with previous versions:

- [New Features](#)
- [Interface Changes](#)

B.1 NEW FEATURES

TLS SUPPORT

TLS is a security mechanism that operates on the Transport layer, on top of TCP Transport. By using TLS as connection transport, a SIP entity can send and receive data in a secure and authenticated manner. The feature is compliant with *RFC 3261*.

Note This feature is relevant for SIP Phones only.

B.2 INTERFACE CHANGES

The following interface changes were made:

- The name of the function:
rvIppMdmGetHoldConn()
was changed to:
rvIppMdmTerminalGetHeldConn()
- In the callback:
rvIppStunAddressResolvedStartCB()
the parameter "bAsynchronousResolvePermitted" was removed.

Interface Changes

APPENDIX C

WHAT'S NEW IN VERSION 2.0

This appendix describes additions and modifications in version 2.0 of the Multimedia Terminal Framework, as compared with previous versions of the IP Phone Toolkit:

- New Features
- New API and Types
- Interface Changes
- Modifications

C.1 NEW FEATURES

CALL FORWARD

This section describes additional features in the current version of the Multimedia Terminal Framework.

Multimedia Terminal Framework version 2.0 supports the Call Forward feature in three different modes:

- **Unconditional**—all incoming calls are forwarded.
- **Busy**—incoming calls are forwarded only when the terminal is busy with another call.
- **No Reply**—incoming calls are forwarded only when they are not answered.

OUT-OF-BAND DTMF

In Multimedia Terminal Framework version 2.0 a new configuration parameter was added, enabling DTMF relay to be sent with a signaling message carried by an INFO message.

Note This feature is relevant for SIP Phones only.

SESSION TIMER

The Session Timer feature defines a keepalive mechanism for SIP. As such, it helps user agents and Proxies determine whether a call is still active. The duration of a call is determined by Session Timer headers and can be extended by refresh messages. This feature is compliant with *RFC 4028* (Session Timers in the Session Initiation Protocol), and is implemented by a Re-Invite message as a refresh message.

Note This feature is relevant for SIP Phones only.

C.2 NEW API AND TYPES

NEW API

This section describes new APIs and directory types that were added to Multimedia Terminal Framework version 2.0.

The following APIs were added in Multimedia Terminal Framework v2.0:

[rvMdmControlApi.h](#)

New APIs were added:

- rvIppMdmTerminalSetState
- rvIppMdmTerminalGetState
- rvIppMdmTerminalGetMdmTerm()
- rvIppMdmConnGetCallState()

[rvSipControlApi.h](#)

A new function was added:

- RvIppSipControlGetMdmTerminal()

The following callback was removed:

- RvSipPostStateChangedCB()

[rvIppCfwApi.h](#)

Call forward functions and callbacks were added:

- **Functions**
rvCCCfwGetTypeNumber()
- **Callbacks**
rvIppCfwActivateCompletedCB()
rvIppCfwDeactivateCompletedCB()

rvmdm.h

A new function was added:

- `rvMdmTermMgrUnregisterTermFromNetwork_()`

NEW TYPES

The following types were added:

- **Terminal State**

States were added to the Terminal object:

`RV_CCTERMINAL_IDLE_STATE`

`RV_CCTERMINAL_CFW_ACTIVATING_STATE`

- **RvCCTermConnState**

New state was added:

`RV_CCTERMCONSTATE_REMOTE_AND_LOCAL_HOLD`

- **RvCCTerminalEvent**

New event was added:

`RV_CCTERMEVENT_CFW`

Note For more information, see the Multimedia Terminal Framework *Reference Guide*.

C.3 INTERFACE CHANGES

The name of the function:

rvIppMdmGetHoldConn()

was changed to:

rvIppMdmTerminalGetHeldConn()

C.4 MODIFICATION S

HOLD FEATURE

This section describes modifications in the current version of the Multimedia Terminal Framework, as compared with previous versions of the Toolkit.

The Hold feature was modified to enable users to implement Music On Hold. In the current version, different media callbacks are called when a call is put on Hold and when it is Unheld. The modifications are described in the table below.

	In previous versions	In version 2.0
Call on Hold	The media stream was disconnected by calling the callback <i>RvMdmTermMgrDisconnectCB</i> .	The media stream is made unidirectional by calling the callback <i>RvMdmTermModifyMediaCB</i> .
Call is Unheld	The media stream was re-connected by calling the callback <i>RvMdmTermMgrConnectCB</i> .	The media stream is made bi-directional by calling the callback <i>RvMdmTermModifyMediaCB</i> .



Existing customers who have already integrated a previous version of the **Multimedia Terminal Framework** into their application and want to integrate version 2.0, should modify the implementation of *RvMdmTermModifyMediaCB* as described in the chapter [Building Your Application](#), section [Step 4: Integrating Media](#).

TLS SUPPORT

The TLS feature was modified as follows:

- The configuration parameter **NumOfTlsAddresses** was added.
- The ability to disable TLS was added:
 - In the previous version of the Multimedia Terminal Framework, applications compiled with TLS flags could not disable TLS unless the application was re-compiled without TLS flags. In other words, if the application was compiled with TLS flags, the application could choose to initiate outgoing calls without TLS, but all incoming TLS calls would still be supported.
 - In the current version of the Multimedia Terminal Framework, applications compiled with TLS can disable TLS simply by changing the configuration. If the application was compiled with TLS flags, the application can now set the new configuration parameter **NumOfTlsAddresses** to the value of zero

and initialize the Multimedia Terminal Framework and the Stack without TLS support, as if it had been compiled without TLS. (The application will not be able to initiate TLS calls or support incoming TLS calls.)

MATCH DIAL STRING TO ADDRESS (1)

If the Multimedia Terminal Framework wants the user application to match a dial string to a Destination Address, it calls the callback

RvMdmTermMapDialStringToAddressCB(). The user application may return the address with or without a scheme, in the following format:

<scheme> : <username> @ <domain name/ipp address> : <port>

For example:

sips:User-057@172.20.69.57:5080

If a scheme is not specified, it is set by default to: "sip:".

For example:

User-057@172.20.69.57:5080

Note For more information, see chapter [Building Your Application](#), section [Matching Digits to an Address](#).

MATCH DIAL STRING TO ADDRESS (2)

If no local match for a dial string was found (i.e., the callback *RvMdmTermMapDialStringToAddressCB()* returns False), the Destination Address will be set to Server Address along with the dialed digits, as described below.

In previous versions:

The Destination Address was set to the Outbound Proxy address:

<dial string> @ <outbound proxy address>

In version 2.0:

The Destination Address is set to the Registrar Address:

<dial string> @ <registrar address>

SAMPLE APPLICATION

The sample application was changed: Both the GUI application and the sample application were modified to support video.

Modifications

APPENDIX D

WHAT'S NEW IN VERSION 2.1

This appendix describes additions and modifications in version 2.1 of the Multimedia Terminal Framework:

- [New Features](#)

D.1 NEW FEATURES

EARLY MEDIA

The Multimedia Terminal Framework enables an external callee to connect media before a final INVITE response is sent. This is done when receiving 183—Call In Progress with SDP body.

Note This feature is relevant for SIP Phones only.

DISTINCTIVE RINGING

Allows the sender of an INVITE/180 (Ringing) message to instruct the receiver of the message to play a non-default Ringing/ringback tone according to the content of the Alert-Info header in the message.

Note This feature is relevant for SIP Phones only.

PRACK

The Multimedia Terminal Framework supports the Reliable Provisional Request mechanism by sending a PRACK response to provisional requests.

Note This feature is relevant for SIP Phones only.

New Features

APPENDIX E

WHAT'S NEW IN VERSION 2.5

This appendix describes additions and modifications in version 2.5 of the Multimedia Terminal Framework:

- [Interface Changes](#)
- [New Features](#)

E.1 INTERFACE CHANGES

[Dynamic Modify Media](#)

In Multimedia Terminal Framework version 2.5 user application does not need to open a new media stream before calling `rvMdmTermModifyMedia()`. Instead, the Multimedia Terminal Framework will call the `RvMdmTermModifyMediaCB()` callback to indicate to the user application that media should be modified. (See [Dynamic Media Change](#), [Figure 8-14](#).)

[Unnecessary APIs removed](#)

The following APIs were removed from the Multimedia Terminal Framework and should be deleted from the user application code. These functions did not have any effect on Multimedia Terminal Framework functionality and should not affect functionality of the user application:

- `rvMdmTermClassSetUserData`
- `rvMdmMediaStreamSetUserData2`
- `rvMdmPackageRegisterEventParametersCB`
- `rvMdmTermClassRegisterSetStateCB`

Extension callback RvIppSipPreRegClientStateChangedCB()

A new parameter was added to this callback to enable the user application to get a handle to the MDM terminal when the Register state changes:

The callback was changed from:

```
typedef RvBool (*RvIppSipPreRegClientStateChangedCB) (
    IN RvIppSipControlHandle          sipMgrHndl,
    IN RvSipRegClientHandle           hRegClient,
    IN RvSipAppRegClientHandle       hAppRegClient,
    IN RvSipRegClientState           eState,
    IN RvSipRegClientStateChangeReason eReason,
    IN void*                          userData);
```

To:

```
typedef RvBool (*RvIppSipPreRegClientStateChangedCB) (
    IN RvIppSipControlHandle          sipMgrHndl,
    IN RvSipRegClientHandle           hRegClient,
    IN RvIppTerminalHandle           mdmTerminalHndl,
    IN RvSipRegClientState           eState,
    IN RvSipRegClientStateChangeReason eReason,
    IN void*                          userData);
```

Call Forward callback rvIppCfwActivateCompletedCB()

A new parameter, cfwDestination, was added to this callback to enable the user application to match the callback to the Call Forward destination.

The callback was changed from:

```
typedef void (*rvIppCfwActivateCompletedCB) (
    IN RvIppTerminalHandle          term,
    IN RvIppCfwType                 cfwType,
    IN RvIppCfwReturnReasons       returnCode);
```

To:

```
typedef void (*rvIppCfwActivateCompletedCB) (
    IN RvIppTerminalHandle          term,
    IN RvIppCfwType                 cfwType,
    IN RvChar*                      cfwDestination,
    IN RvIppCfwReturnReasons       returnCode);
```

STUN callback RvIppAddressResolveReplyCB()

A new parameter, addr, was added to this callback.

Callback changed from:

```
typedef RvStatus (RVCALLCONV *RvIppAddressResolveReplyCB) (
    IN RvAddress*     addr,
    IN RvChar*        buf,
    IN RvSize_t       size,
    OUT RvBool*       bDiscardMsg);
```

To:

```
typedef RvStatus (RVCALLCONV *RvIppAddressResolveReplyCB) (
    IN RvChar*        buf,
    IN RvSize_t       size,
    OUT RvBool*       bDiscardMsg);
```

Extension callback RvIppSipPreMsgReceivedCB

The return value of this callback was changed from RvBool to RvMtfMsgProcessType, so the callback was changed from:

```
typedef RvMtfMsgProcessType (*RvIppSipPreMsgReceivedCB) (
    IN RvIppSipControlHandle    sipMgrHndl,
    IN RvSipCallLegHandle      hCallLeg,
    IN RvSipAppCallLegHandle   hAppCallLeg,
    IN RvSipMsgHandle          hMsg,
    IN void*                   userData);
```

To:

```
typedef RvBool (*RvIppSipPreMsgReceivedCB) (
    IN RvIppSipControlHandle    sipMgrHndl,
    IN RvSipCallLegHandle      hCallLeg,
    IN RvSipAppCallLegHandle   hAppCallLeg,
    IN RvSipMsgHandle          hMsg,
    IN void*                   userData);
```

RvMtfMsgProcessType can be set the one of the following values:

RV_MTF_IGNORE_BY_STACK—This value indicates to both SIP Stack and Multimedia Terminal Framework to ignore the message. When this value is returned, callback **RvIppSipPreCallLegCreatedIncomingCB()** should return **False** as well, otherwise a memory leak will occur (when RvIppSipPreCallLegCreatedIncomingCB() is called resources (of SIP connection) will be allocated but not be released).

RV_MTF_IGNORE_BY_MTF—This value indicates to the Multimedia Terminal Framework to ignore the message, but message will still be processed by the SIP Stack.

RV_MTF_DONT_IGNORE—This value indicates both SIP Stack and Multimedia Terminal Framework to process the message.

E.2 NEW FEATURES

CACHE ALLOCATOR

In previous versions of the Multimedia Terminal Framework, memory was allocated purely dynamically, meaning that only the necessary amount of memory was allocated when it was required. Besides resulting in large numbers of system calls, dynamic allocation could also cause memory fragmentation and affect performance.

Multimedia Terminal Framework version 2.5 changes the behavior of memory allocation through the new Cache Allocator module. This module allocates an initial amount of memory during startup, which it uses and reuses during runtime. In addition, adjustments to application behavior can be made during runtime by allocating more blocks when memory consumption increases and releasing them when memory consumption decreases.

By using the Cache Allocator for memory allocation, the Multimedia Terminal Framework avoids memory fragmentation as much as possible, while optimizing performance. The Cache Allocator can be easily configured to match usage by the application. For more information on the Cache Allocator, see the chapter [Optimizations](#) and the Appendix [Cache Allocator](#).

ADVANCED DNS

The Multimedia Terminal Framework enables the user application to use the Advanced DNS supported in the SIP Stack, which supports the client DNS procedures defined in RFC 3263 and RFC 3824. This includes usage of NAPTR and SRV DNS queries and the ability to maintain a list of resolved addresses that the application can try one after the other in case of a send failure.

The Multimedia Terminal Framework supports Advanced DNS for Invite and Register messages only. The most common use of Advanced DNS in the Multimedia Terminal Framework is for locating SIP Servers (RFC 3263).

NEW DIALING API

In previous versions, Multimedia Terminal Framework enabled the user to establish a call by collecting one digit at a time. The new dialing API enables the user to make a call with a destination address (any valid SIP address). Calling this API will result in sending out Invite message to the destination specified in "address" parameter:

```
RvStatus rvMtfTerminalMakeCall(  
    IN RvIppTerminalHandle    terminalHndl,  
    IN const RvChar*          address);
```

Before calling this API, the user application should send an Off Hook event. The call should be terminated by sending an On Hook event.

MEDIA ADD-ON

The Media Add-on enables enhanced media features by seamlessly integrating with RADVISION's Media Framework and Advanced RTP Toolkit. The Media Add-on saves application developers the effort of media integration as described in the section [Step 4: Integrating Media](#).

New Features

APPENDIX F

WHAT'S NEW IN VERSION 2.5.1.47

This appendix describes additions and modifications in version 2.5.1.47 of the Multimedia Terminal Framework:

- [New Features](#)
- [Interface Changes](#)
- [Memory Optimization](#)

F.1 NEW FEATURES

Version 2.5.1.47 of the Multimedia Terminal Framework includes the following new features:

- Authentication for REFER is supported—when the MTF is challenged with regard to authentication of a REFER message, it sends another REFER message containing the authentication credentials.
- New configuration parameters were added:
 - **sdpStackCfg**—This parameter is an SDP structure that enables the user application to configure the SDP Stack. MTF will use this parameter to initialize the SDP Stack as part of MTF initialization. This parameter allows the user application to configure log filters.
 - **connectOn180**—When this parameter is set to True, media will be connected when 180 Ringing; 183 Session In Progress and 200 OK are received. When set to False, media will be connected in 183 Session In Progress and only 200 OK is received.

F.2 INTERFACE CHANGES

Version 2.5.1.47 of the Multimedia Terminal Framework includes the following interface changes:

- Interface of rvMtfAllocatorDealloc() changed from:

```
RvBool rvMtfAllocatorDealloc(  
    IN void *ptr);
```

To:

```
RvBool rvMtfAllocatorDealloc(  
    IN void *ptr,  
    RvSize_t size);
```

F.3 MEMORY OPTIMIZATION

Version 2.5.1.47 of the Multimedia Terminal Framework includes the following changes:

- The Cache Allocator table (rvDefAllocCacheMemoryConfig) is configured to the usage of 1-2 simultaneous calls, thus reducing the initial memory allocated by the MTF.
See the [Optimizations](#) chapter and the [Cache Allocator](#) appendix for more information about configuring the Cache Allocator.

APPENDIX G

WHAT'S NEW IN VERSION 2.5.1.54

This appendix describes additions and modifications in the current version of the Multimedia Terminal Framework:

- [Memory Reduction](#)
- [Interface Changes](#)
- [Other Changes](#)

G.1 MEMORY REDUCTION

Memory consumed by Multimedia Terminal Framework version 2.5 was reduced. The following modifications were made:

- By default, Multimedia Terminal Framework version 2.5.1.54 allocates its memory dynamically, without using the Cache Allocator. Thus, it allocates the minimum amount of memory required. If the user application wants to reduce the number of dynamic allocations and improve performance, it can add the Cache Allocator with a compilation flag. For more information, see the [Optimizations](#) chapter.
- When the Multimedia Terminal Framework is compiled with the Cache Allocator, memory consumption of the Cache Allocator was reduced to ~150K. It is recommended not to change the default configuration of the Cache Allocator.
- The linking to C++ library was removed. Changed compilation mode of g++ to gcc (file tool_gnu.mak).

G.2 INTERFACE CHANGES

A new callback, *RvIppSipPostStateChangedCB*, was added. This event callback is invoked each time the SIP call-leg state is changed, and after the Multimedia Terminal Framework has completed processing that event. For more information, see the [Extensibility](#) chapter.

Note

To avoid incompatibility when new callbacks are added, it is recommended to change the registrations of Multimedia Terminal Framework callbacks in your application from an assignment of the full structure, as in the following:

```
userSipClbks =
{
    userSipStackConfig,
    userSipRegisterStackEvents,
    userSipPreCallLegCreatedIncoming,
    userSipPostCallLegCreatedIncoming,
    userSipPreCallLegCreatedOutgoing,
    userSipPostCallLegCreatedOutgoing,
    userSipPreStateChanged,
    userSipPostStateChanged,
    userSipPreMsgToSend,
    userSipPostMsgToSend,
    userSipPreMsgReceived,
    userSipPostMsgReceived,
    userSipPreRegClientStateChanged,
    &userInfo
}
```

To an assignment of each individual callback, as in the following:

```
userSipClbks.stackConfigF = userSipStackConfig;
userSipClbks.registerStackEventsF =
    userSipRegisterStackEvents;
userSipClbks.preCallLegCreatedIncomingF =
    userSipPreCallLegCreatedIncoming;
userSipClbks.postCallLegCreatedIncomingF =
    userSipPostCallLegCreatedIncoming;
userSipClbks.preCallLegCreatedOutgoingF =
    userSipPreCallLegCreatedOutgoing;
userSipClbks.postCallLegCreatedOutgoingF =
    userSipPostCallLegCreatedOutgoing;
```

```
userSipClbks.preStateChangedF = userSipPreStateChanged;
userSipClbks.postStateChangedF = userSipPostStateChanged;
userSipClbks.preMsgToSendF = userSipPreMsgToSend;
userSipClbks.postMsgToSendF = userSipPostMsgToSend;
userSipClbks.preMsgReceivedF = userSipPreMsgReceived;
userSipClbks.postMsgReceivedF = userSipPostMsgReceived;
userSipClbks.PreRegClientStateChangedF =
    userSipPreRegClientStateChanged;
userSipClbks.userData = &userInfo;
```

G.3 OTHER CHANGES

The compilation flag name was changed from RV_MTF_TRACE_LEAKS to RV_MTF_USE_CACHE_ALLOCATOR.

Other Changes

INDEX

A

advanced DNS 346
application
 building 45
 sample 157
architecture
 media add-on 319
 MTF 7

B

building an application 45

C

cache allocator 329, 346
 compile-time configuration flags 331
call control 20
call flow 109
 sample 24, 25, 197
call forward 298, 335
call object 15
call transfer 240
 attended 240
 blind 250
 semi-attended 240
call waiting 237
caller ID 233
camera 84
capability guidelines 9
client authentication 267
code flow 118
code sample 190
common core 23

common features 233
compile-time configuration flags 331
components 7, 20
configuration 52, 159
connection object 14

D

dial string 339
 match 339
digits
 collecting 75
digits, matching to an address 82
distinctive ringing 341
DNS, advanced 346
DSP services 22, 84
DTMF out-of-band 296, 335
dynamic media change 282

E

early media 341
echo cancellation 84
ephemeral terminations 12
EPP 195
 events table 198
 integrating MTF without 204
 objects 195
 signals table 200
 sockets 196
events 15
 reporting 69
 signals and packages 15
extensibility 111

F

fast update 84
features 233

G

GUI application 165

H

handset 84
handsfree 257
headset 257
headset/handsfree 257
hold 238, 338

I

initialization 46
 steps 46
initializing the log 213
integrating media 84
integrating MTF without EPP 204
interface changes 333
IPv4 261
IPv6 207, 261
 sample application 207

J

jitter buffer 84

L

lip synch 84
list of events 70
 for analog terminations 73
list of signals 66
 for analog terminations 68
log 213, 218
 closing 218
 SIP stack 219
logging 213

M

matching digits to an address 82
MDM 22
 adaptor 21
 API 22
 control 112
 extension APIs 117
 extension callbacks 113
media add-on 315, 347
 codecs 323
 features 323
 working with 316
media callbacks 85
media framework 321
media stream 85
media, integrating 84
memory optimization 350, 353
message waiting indication 208
mixing 84
model design 11
mute 259
MWI sample application 208

O

optimization 325
outbound proxy 267
out-of-band DTMF 335

P

performance, optimizing 327
physical termination 11
playing signals 64
PRACK 341
protocol stack 23
protocol-specific features 263

R

registering terminations 58
registration 263

registration refresh 265
reporting events 69
 API 69
residential gateway, sample 14
ringing, distinctive 341

S

sample application 157
 running 168
SDP stack 23
session timer 302, 336
shutting down 227
signal callbacks 64
signals 15
 playing 64
SIP
 adaptor 22
 call flow 25
 control 133
 extension callbacks 135
 features 263
 stack 23
 stack log 219
softphone sample application 165
stack log 219
starting the system 62
state machine 26
structure 20
structure and objects 185
STUN 268

T

telephony services 22, 84
terminations 11
 ephemeral 12
 physical 11
registering 58
thread model 41
thread relations 42
three-way conference 254
TLS 288, 333, 338

U

unregistration 266
user interface files 119

V

voice compression 84
voice decompression 84

W

warm restart 260