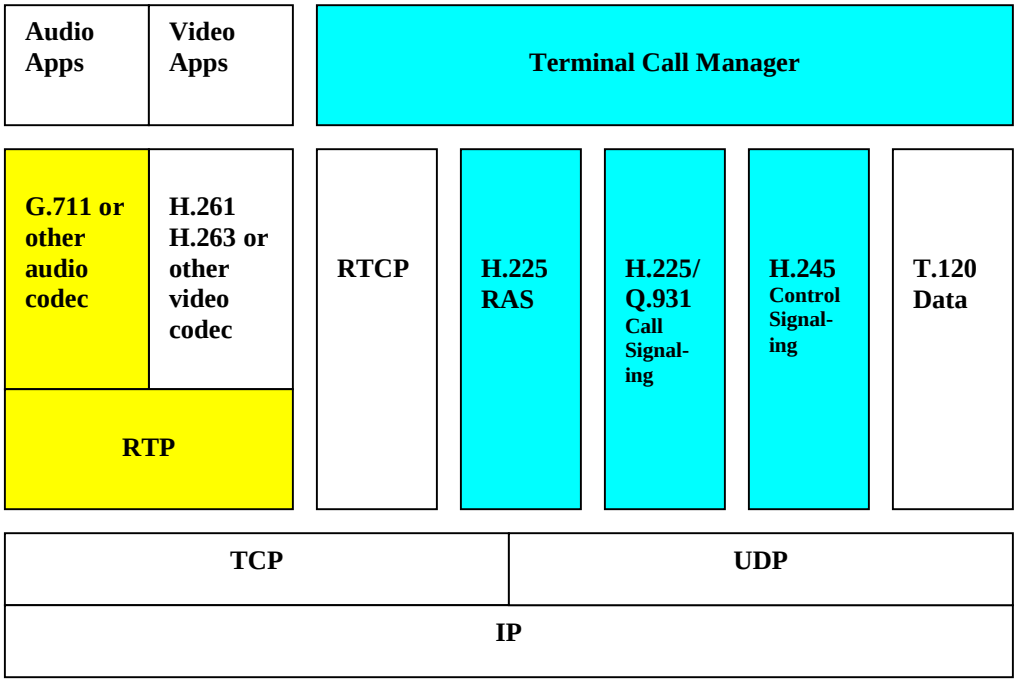
Objective Open
H.323 for C
User's Guide

Objective Systems, Inc. March 2005

1. Introduction

H.323 is an ITU-T Recommendation describing the protocols involved in establishing channels for multimedia communications over a packet-based network.

The Objective Open H.323 for C (OOH323C) protocol stack is an open source applications program interface (API) for building H.323 based applications. The stack implements Q.931/H.225 call signaling procedures, H.245 logical channel operations, and Registration, Admission, and Status (RAS) messaging for Gatekeeper communications. The supported parts of a standard H.323 stack are shown in the diagram below:



Key:

Supported by ooH323c

Supported by 3rd-party software

In this diagram, parts of the H.323 stack supported by OOH323C are shown in the light blue color. In addition, sample programs are included that contain third-party open-source media code to demonstrate a complete audio application that can play and receive a recorded audio file. The additional parts of the stack supported by this code are shown in yellow.

2. Architecture

The Objective Open H.323 for C stack is implemented as a simple, single-threaded application that allows a single H.323 endpoint to be set up that can create and teardown H.323 signaling and media channels. I/O multiplexing is done by means of a UNIX *select* command to monitor all active I/O channels. TCP/IP and UDP communications are supported.

The main structure off which everything in the stack operates is the H.323 Endpoint – *OOH323EndPoint*. There is one global endpoint that is shared by all of the modules within the stack. This holds all of the standard configuration data (port numbers, terminal types, timeout values, etc). It also holds the active call list.

The active call list is a linked list of *OOH323CallData* records. Each time a call is initiated either from the stack or from a remote endpoint, a record is added to this list. This record is maintained throughout the life of the call. It contains details on negotiated capabilities and the different channels that have been set up to handle the call.

User application program interaction with the stack is accomplished by means of stack commands and callback functions.

2.1 Stack Commands

Stack command API functions are called to initiate a processing sequence within the stack. The typical initiation function that is called is *ooMakeCall* to initiate a call. Once this is done, callback functions are triggered as different H.323 messages are exchanged to set up the channels.

The following are the top-level stack commands:

Stack Command	Description
ooMakeCall	Initiate a call.
ooAnswerCall	Manually answer a call (this is only required when auto-answer is disabled).
ooHangCall	Terminate a call.
ooForwardCall	Forward call.
ooManualRingback	When manual ring back is enabled, this command is used to send ring-back message(alerting).
ooSendDTMFDigit	Used to send dtmf sequence to remote endpoint.
ooStopMonitor	This command is used to terminate stack thread.

ooMakeCall
This stack command is used by an H.323 application to place a call. The complete signature of this command function is as follows:

```
OOStkCmdStat ooMakeCall(const char *dest, char *callToken, size_t bufsiz, OOCallOptions *opts);
```

This command instructs stack to place an outgoing call on applications behalf. The first parameter *dest* specifies the destination for the call. Destination can be ip address, ip:port combination or alias name, if the gatekeeper is used. The second parameter, *callToken,* is a buffer in which the stack will return a token for the newly created call. Applications will use this token for future communication with stack for the newly created call. The third parameter, *bufsiz,* indicates the size of the call token buffer. The last argument, *opts*, is a pointer to OOCallOptions structure which allows call specific configuration. It allows application to enable/disable faststart, tunneling per call basis. It also allows application to disable use of gatekeeper for a particular call. Though this is not standard practice, it is useful for applications such as PBXs, which want to use gatekeeper for outside calls and use dialplan/database/configuration to route internal calls. In addition, call options allow application to override default call mode set for the global endpoint. The return value is OOStkCmdStat enumeration which returns the status of the command. The text description for the returned status value can be obtained by calling function *const char* ooGetStkCmdStatusCodeTxt(OOStkCmdStat stat).*

ooAnswerCall
This stack command is used by an H.323 application to answer an incoming call.

```
OOStkCmdStat ooAnswerCall(char* callToken);
```

It takes *callToken*, which uniquely identifies the call, as parameter. Applications can configure an H.323 endpoint to automatically answer an incoming call, in which case this command will not be used. The return value is same as described for *ooMakeCall.*

ooHangCall
This stack command is used to terminate a call. It's signature is as follows:

```
OOStkCmdStat ooHangCall(char *callToken, OOCallClearReason reason);
```

It takes *callToken*, an unique identifier for the call and *reason*, which indicates the reason for call termination, as parameters. The call clear reason is of type OOCallClearReason and is defined in ootypes.h file. The return value is same as described for *ooMakeCall.*

ooForwardCall
This stack command is used to forward an existing call to third party. The command signature is as follows:

```
OOStkCmdStat ooForwardCall(char *callToken, char *dest);
```

The first parameter, *callToken*, uniquely identifies the call. The second parameter, *dest*, is the new destination for the call. Destination can be ip address, ip:port combination or alias name. The return value is same as described for *ooMakeCall.*

ooManualRingback
This stack command is used to send an alerting message after the user has been alerted by application. The command signature is as follows:

```
OOStkCmdStat ooManualRingback (const char* callToken);
```

The callToken argument is a unique identifier for the call. This function is effective only when manual ring-back is enabled for the endpoint. By default, it is disabled and the ooh323c stack sends alerting message automatically. The return value is same as described for *ooMakeCall.*

ooSendDTMFDigit
This stack command is used to send a dtmf sequence to remote endpoint. The command signature is as shown below:

```
OOStkCmdStat ooSendDTMFDigit(const char* callToken,
                             const char* alpha);
```

Note that this command sends dtmf digits by using either H.245 UserInput messages or by using Q.931 keypad IE. The return value is same as described for *ooMakeCall.*

ooStopMonitor
This function is used to terminate stack thread. The command signature is shown below:

```
OOStkcmdStat ooStopmonitor ();
```

The return value is same as described for *ooMakeCall.*

2.2 Callback Functions

The stack is event-driven and will react to I/O or timer events by calling registered callback functions. There are three types of callbacks, viz., H.323 callbacks (call level callbacks), H.225 callbacks (H.225 message level callbacks), and channel callbacks (callbacks to start/stop transmit/receive channels).

2.2.1 H.323 or Call Level Callbacks

Callback	Description
onNewCallCreated	This is triggered when a new call structure is created inside the stack for an incoming or outgoing call.
onIncomingCall	This is triggered when there is an incoming call. In case where a gatekeeper is being used, the new call must first be admitted by the gatekeeper.
onOutgoingCall	This callback is invoked after a Q.931 setup message is sent for an outgoing call.
onAlerting	This is triggered when a Q.931 alerting message is received for an outgoing call or when a Q.931 alerting message is sent for an incoming call.
onCallEstablished	This is triggered when a Q.931 connect message is sent in case of incoming call. In case of outgoing call, this is invoked when a Q.931 connect message is received. It is not invoked until after fast start and H.245 tunneling messages within the connect message are processed.
onCallForwarded	This is triggered when remote destination has forwarded our call to a new destination.
onCallCleared	This is triggered when a call is cleared.

openLogicalChannels	This is called when Master-Slave determination and Capabilities Negotiation procedures are successfully completed for a call. If not provided, the stack will use call mode to decide which channels to create.(Refer section 6.2)
onReceivedDTMF	This is called when DTMF digit is received from remote endpoint either as Q.931 keypad IE or as H.245 UserInput indication message.

2.2.2 H.225 or Message Level Callbacks

Callback	**Description**
onReceivedSetup	This is triggered when a H.225 SETUP message is received.
onReceivedConnect	This is triggered when a H.225 CONNECT message is received.
onBuiltSetup	This is invoked after a H.225 SETUP message is built and before it is sent out.
onBuiltConnect	This is invoked after H.225 CONNECT message is built and before it is sent out.

2.2.3 Channel Callbacks

These callbacks are used to start and stop transmit and receive media channels. User applications can register the same callbacks for various capabilities or can register different callbacks for different capabilities.

Callback	**Description**
startReceiveChannel	This is triggered either during a FastStart operation or in response to a received *OpenLogicalChannel* message from the remote endpoint.
startTransmitChannel	This is triggered either during a FastStart operation or in response to a received *OpenLogicalChannelAck* message from the remote endpoint.
stopReceiveChannel	This is triggered when a call is being cleared or in response to a *CloseLogicalChannel* request received from the remote endpoint.
stopTransmitChannel	This is triggered when a call is being cleared or in response to a *CloseLogicalChannel* request received from the remote endpoint.

3. Contents of the Package

The following diagram shows the directory tree structure that comprises the H.323 stack package:

```
ooh323c
 |
 +- doc
 |
 |
 +- src
 |  |
 |  +- h323
 |
 +- specs
 |
 +- tests
 |   |
 |   +- chansetup
 |   |
 |   +- simple
 |   |
 |   +- player
 |   |
 |   +- receiver
 |
 +- media
```

The purpose and contents of the various subdirectories are as follows:

- doc – this directory contains this document as well as the *H.323 Introduction*
- src – this directory contains all of the C source files
    - The top level src directory contains the C source files for the stack
    - The h323 subdirectory contains compiled ASN.1 code
- specs – this directory contains the H.323 ASN.1 specifications
- tests – this directory contains sample H.323 application programs.
    - The H.323 player application transmits audio data from a audio file (16 bit, 8000 samples/sec WAV file on windows, 16 bit raw data audio file on Linux)
    - The receiver application receives the RTP audio data stream and plays it on the speaker
    - The simple H.323 phone application demonstrates the ability to set up voice calls, negotiate capabilities and start voice channels.
    - The chansetup program contains basic skeleton code for setting up and tearing down an H.323 channel.
- media – this directory contains the sample media library implementation. It supports crude RTP protocol support and ulaw/alaw codec support. It also supports wave file transmission on windows platform. The G711 codecs implementation is the one provided by Sun Microsystems to open source community. The H.323 application developers can use their own RTP/RTCP and codec implementations with ooH323c stack.

4. Installation and Build Procedures

The package is delivered as a .tar.gz or .zip archive file that may be unpacked into any root directory. After unpacking the package, change directory to the package root directory.

Building the Package on Linux

Generate Makefiles for the package

./configure --prefix=<install-path>

The default <install-path> is /usr/local. This is where the final compiled oohh323c stack library is installed upon completion of the build and install procedure. Users must specify <install-path> to have the stack in their local user directory. For example, the following will cause the library to be installed in the user's ooh323c/lib subdirectory after 'make install' is executed:

./configure –prefix=$HOME/ooh323c

Build and Install the Package

To build the complete package including test applications:
./make

To build an optimized version (this is the default):
./make opt

To build the debug version:
./make debug

To install package,
./make install

This will install the libraries in the <install-path> specified while running 'configure' script.

Building the Package on Windows

The package includes Visual Studio 6 based workspace and project files.

1. Open the package root directory ooh323c-x.y, where x.y indicate the release number.

2. Open the ooh323c.dsw workspace, which includes all the projects within the package.

3. You can now do a batch build to build the complete package.

4. To build individual projects, dependencies are as follows:
   oostk.dsp – none
   oomedia.dsp – none
   h323ep.dsp - oostk.dsp, oomedia.dsp
   ooPlayer.dsp - oostk.dsp, oomedia.dsp
   ooReceiver.dsp - oostk.dsp, oomedia.dsp

After a successful build the libraries will be installed in the ooh323c-x.y\lib\release and ooh323c-x.y\lib\debug directories.

5. Running the Sample Programs

Running the Programs on Linux

Running the Receiver and Player Applications (player and receiver)

Make sure that the path to the *liboomedia.so* shared object file is in your LD_LIBRARY_PATH path. This library is located in the installed *lib* subdirectory. If the library is in your local *ooh323c* directory tree, the following commands can be used to set the library path:

```
LD_LIBRARY_PATH=$HOME/ooh323c/lib
export LD_LIBRARY_PATH
```

note that in the command above, *ooh323c* may have additional version information appended.

Next, the 'receiver' application can be run as follows:

```
cd tests/receiver
./ooReceiver [--use-ip <ip>] [--use-port <port>]
```

where, `[--use-ip <ip>] and [--use-port <port>]` options are used to specify local ip address and port. By default, the receiver application tries to determine the ip address and uses H323 default port 1720.
A log file will be created in the current directory (ooReceiver.log). Also, a log file for the media plug-in will be created in the same directory (media.log).

Now, run the player application from a new console window as follows:

```
cd tests/player
./ooPlayer –-audio-file space.raw [--use-ip <ip>]
```

where, `--audio-file is used to specify the audio file to play and [--use-ip <ip>] is` used to specify local ip address. By default, the player application tries to determine the ip address on it's own.
A log file will be created in the current directory (ooPlayer.log). Also, a log file for the media plug-in will be created in the same directory (media.log).

The result should be the recorded sounds in the *space.raw* file being played on your computer's speakers.

Running the Simple Phone Application (simple)

Make sure all the path to the *liboomedia.so* shared object file is in LD_LIBRARY_PATH as specified in the previous section.

Next, change directory to the simple test directory:

```
cd tests/simple
```

To see the usage information including various options:

./simple OR ./simple --help

To make a call:

./simple [options] <remote>

where,

<remote> - is the dotted representation of the destinations IP address. In case of gatekeeper, aliases can also be used.

To receive a call:

./simple [options] --listen

You will find simple.log and media.log in the current directory.

Running the Programs on Windows

To run the sample programs on Windows, make sure that the media plug-in library *oomedia.dll* is in your PATH. The libraries are located in the ooh323c-x.y\lib\release and ooh323c-x.y\lib\debug directories.

First, run the receiver from the command prompt as follows (note: you must be in the package root directory):

```
cd tests\receiver\Release
ooReceiver.exe [--use-ip <ip>] [--use-port <port>]
```

where, `[--use-ip <ip>] and [--use-port <port>]` options are used to specify local ip address and port. By default, the receiver application tries to determine the ip address and uses H323 default port 1720.

A log file will be created in the current directory (ooReceiver.log). Also, a log file for the media plug-in will be created in the same directory (media.log).

Now run the player from the command prompt:

```
cd tests\player\Release
ooPlayer.exe –-audio-file states.wav [--use-ip <ip>]
```

where, `--audio-file is used to specify the audio file to play and [--use-ip <ip>] is` used to specify local ip address. By default, the player application tries to determine the ip address on it's own.

A log file will be created in the current directory (ooPlayer.log). Also, a log file for the media plug-in will be created in the same directory (media.log).

To run the sample telephony endpoint application, again ensure that the media plug-in library (*oomedia.dll*) is in your PATH. The library is located in ooh323c-x.y\lib\release and ooh323c-x.y\lib\debug directories.

To run the telephony application:

```
cd tests\simple\Release
```

To see the usage information including various options:
simple OR simple --help

To make a call:

simple [options] <remote>

where,
<remote> - is the dotted representation of the destinations IP address. In case of gatekeeper, aliases can also be used.

To receive a call:

simple [options] --listen

You will find simple.log and media.log in the current directory.

6. Developing H.323 Application

This section explains how users can develop H.323 enabled applications using OOH323C stack library. It first covers how to integrate third party RTP/RTCP stack and codecs and then covers application development process.

6.1 Using third party RTP/RTCP stack and Codecs

Though media library provided with OOH323C stack supports prototype implementation of RTP protocol and Sun Microsystems G711 codecs, most commercial application developers would want to use their own RTP/RTCP stack as well as codecs. OOH323C stack is designed keeping this requirement in mind. In an OOH323C based application, whenever media transmission/reception has to be started/stopped, the stack uses the channel callback functions described in section 2.2.3. The only thing application developers have to do, is to implement these callback functions so that they will use their own proprietary RTP/RTCP stack and codecs. Consider the callback function to start transmit channel:

```
int (*cb_startTransmitChannel)(struct OOH323CallData *call, struct OOLogicalChannel *pChannel);
```

This callback function is called with two parameters. The first parameter is the call specific data structure, which holds all the call related information. The second parameter holds information for media channel, specifically, remote media IP and Port address and codec to be used for transmission. Armed with this information, application developers can easily implement this callback function to use their own RTP/RTCP stack and codecs. Same applies for receive channel, where instead of remote media IP and Port, local media IP and Port address information is provided along with codec type of the media to be received. Typically, the callback function will create a new thread which will handle transmission/reception of the media and further processing and return back. The newly created transmission/reception thread will be terminated when corresponding stop callback function is called, for example, *stopTransmitChannel.*

6.2 Creation of Logical Channels

H.323 applications might want to participate in various types of media channels such as audio call, audio receive only, audio transmit only, video call etc. OOH323C stack provides two ways to handle opening of various logical channels. In case of faststart, or in case where H.323 application does not provide *openLogicalChannels* callback, OOH323C stack uses call mode to determine what types of media channels(logical channels) have to be established for a particular call. An H.323 application can provide default call mode for all calls during initialization of global H.323 endpoint structure. In addition, an H.323 application can override global call mode setting on per call basis by providing different call mode in OOCallOptions structure passed as parameter to *ooMakeCall* command. Call mode can be *OO_CALLMODE_AUDIOCALL, OO_CALLMODE_AUDIORX, OO_CALLMODE_AUDIOTX* or *OO_CALLMODE_VIDEOCALL* Another way to handle opening of logical channels is providing a openLogicalChannels callback. Inside this callback functions applications can call stack API function *ooOpenLogicalChannel* which takes type of channel to be opened as parameter, to open audio, video channels as desired.

6.3 Stack Thread

The ooh323c stack always runs in a single thread. The applications have to start the thread with *ooMonitorChannels*() function. This function continuously monitors all the open channels. If application has registered callbacks, the callbacks are also executed by this thread as events occur. The stack thread also monitors command channel which is used to receive commands from application. The application should create a command listener before starting the stack thread. When application calls functions defined in stack commands API(ooStackCmds.h file) to issue a command to stack, the stack command API internally sends that command to stack thread over command channel. The stack command API creates the

command channel when first command function is called by application and uses the same channel for subsequent calls. If application is going to have multiple threads issuing stack commands, then application developer should protect these stack command API calls with mutexes.

6.4 Basic H.323 Application Design Pattern

The general flow of the application will be to use stack API and create an H.323 endpoint, configure the endpoint with various options such as fast-start, tunneling and capabilities supported. Specify whether gatekeeper has to be used and if it has to be discovered. After this start the stack thread and start placing and receiving calls.

Initializing the Endpoint

First thing an H.323 application using OOH323C stack has to do is to initialize the global H.323 endpoint structure. This is accomplished by calling the following API function:

```
ooH323EpInitialize (enum OOCallMode callMode, const char *tracefile);
```

The first argument specifies call mode which is described in section 6.2 above. The second argument is the name of a trace file where logging information should be written.

Other properties can then be set through a series of "ooH323EpSet" calls. These set properties within the global endpoint object.

Enabling debug tracing

By default the OOH323C stack will log trace messages which fall in *INFO*, *ERROR* and *WARNING* categories. For more detailed debug tracing additional three levels are supported. To enable additional tracing following API function is used:

```
int ooH323EpSetTraceLevel(int traceLevel);
```

The three debug trace levels are *OOTRCLVLDBGA*, *OOTRCLVLDBGB*, *OOTRCLVLDBGC* in increasing level of trace information.

Setting Local IP and Port

The most important thing is to set the local ip address and port number to use for listening for incoming calls. For this following API function is used.

```
int ooH323EpSetLocalAddress(char *localip, int listenport);
```

Here, application developers can specify IP address and port number they want to use. If application developer wants H.323 application to listen on all network interfaces, they should specify IP address as "0.0.0.0". The *listenport* can be left 0 to use H.323 default port number 1720.

Enabling/Disabling faststart/tunnelling

The application developers can enable/disable faststart/tunneling at global endpoint level using following functions:

```
int ooH323EpEnableFastStart();
int ooH323EpDisableFastStart();
int ooH323EpEnableH245Tunneling();
int ooH323EpDisableH245Tunneling();
```

Note: This global faststart/tunneling setting can be over-ridden on per call basis for outgoing calls by using appropriate options in *ooMakeCall* stack command as described in section 2.1

Setting Aliases

For adding aliases for your H.323 endpoint following functions can be used:

```
int ooH323EpAddAliasH323ID(char *h323id);
int ooH323EpAddAliasDialedDigits(char *dialedDigits);
int ooH323EpAddAliasURLID(char *url);
int ooH323EpAddAliasEmailID(char *email);
int ooH323EpAddAliasTransportID(char *ipaddress);
```

Setting Caller ID

There are two components to this. One is display name and second is party number, which will be used as calling party number or called party number to represent your endpoint in a call. For setting these values following API functions can be used:

```
int ooH323EpSetCallerID(const char *callerID);  //This sets the display name
int ooH323EpSetCallingPartyNumber(const char *number);  //This sets the party
                                                        //number
```

Auto-Answer Mode

Some times H.323 applications want an incoming H.323 call to be answered directly. Typically non-endpoint applications, such as gateways, gatekeepers, PBXs need this facility. The auto-answer mode can be toggled using following API functions:

```
int ooH323EpEnableAutoAnswer();
int ooH323EpDisableAutoAnswer();
```

Registering callbacks

There are two types of callbacks as described in section 2.2. Call Level callbacks can be registered using API function shown below:

```
int ooH323EpSetH323Callbacks(OOH323CALLBACKS h323Callbacks);
```

Here *h323Callbacks* is a structure of type *OOH323CALLBACKS*. All members of this structure should be initialized to either a valid callback value or NULL. For example, if an H.323 application defines *onNewCallCreated* callback function as *osEpOnNewCallCreated* and *onAlerting* callback function as *osEpOnAlerting* then the code to register them will be:

```
OOH323CALLBACKS h323Callbacks;
h323Callbacks.onNewCallCreated = osEpOnNewCallCreated;
h323Callbacks.onAlerting = osEpOnAlerting;
h323Callbacks .onIncomingCall = NULL;
h323Callbacks.onOutgoingCall = NULL;    //Don't forget the NULLs
h323Callbacks.onCallEstablished = NULL;
h323Callbacks.onCallForwarded = NULL;
```

```
h323Callbacks.onCallCleared = NULL;
h323Callbacks.openLogicalChannels = NULL;
h323Callbacks.onReceivedDTMF = NULL;

ooH323EpSetH323Callbacks(h323Callbacks);
```

For registering message level callbacks following API function is used:

```
int ooH323EpSetH225MsgCallbacks(OOH225MsgCallbacks h225Callbacks);
```

Here *h225Callbacks* is a structure of type OOH225MsgCallbacks which contains pointers to callback functions for handling incoming and outgoing messages. Sometimes H.323 applications exchange non-standard data through extension elements and this non-standard data is not handled by OOH323C stack. In such cases, H.323 applications need an ability to process the received non-standard data and to be able to send non-standard data. For this, H.323 applications can provide callback functions to OOH323C stack. Whenever a new message is received, the applications callback function will be called after the stack finishes processing of standard data in the message. The complete message is passed to the application for further processing. Similarly, before sending out a message on behalf of the application, callback function will be called to allow application to add additional non-standard data, if it desires so. Currently callback functions are supported only for SETUP and CONNECT messages. But support for other messages can be added easily. All members of the structure OOH225MsgCallbacks should be initialized to either a valid callback value or NULL. For example, if an H.323 application defines *onReceivedSetup* callback function as *osEpOnReceivedSetup* and *onBuiltConnect* callback function as *osEpOnBuiltConnect* then the code to register them will be:

```
OOH225MsgCallbacks h225Callbacks;
h225Callbacks.onReceivedSetup = osEpOnReceivedSetup;
h225Callbacks.onReceivedConnect = NULL;  //Don't forget the NULLs
h225Callbacks.onBuiltSetup = NULL;
h225Callbacks.onBuiltConnect = osEpOnBuiltConnect;
ooH323EpSetH225MsgCallbacks(h225Callbacks);
```

Adding Capabilities

An H.323 application must specify to its peers what it is capable of doing. A capability negotiation will then take place within the H.245 message processing or fast-start to arrive at a mutually agreed upon set of capabilities.

The application should add capabilities to the endpoint based on what it can support. The capabilities supported by OOH323C stack are G711 ulaw/alaw, GSM, G723.1, G729 and H.263. For example, to add G.711-ulaw-64K capability with transmission of 30 frames per packet and receive capacity set to 240 frames per packet to the H.323 endpoint:

```
ooH323EpAddG711Capability (OO_G711ULAW64K, 30, 240, OORXANDTX,
                           &startReceiveChannel, &startTransmitChannel,
                           &stopReceiveChannel, &stopTransmitChannel);
```

The different types of capabilities are defined in *ooCapability.h*. These are constants that define known capability types. Users can extend this list if they plan to support additional capabilities not currently in this list. The callback functions *startReceiveChannel, startTransmitChannel* etc. are to be implemented by the H.323 application developer as described in section 6.1. If application developer wants to use the sample media library provided with OOH323C stack, then they should refer to simple.c in simple telephony application for reference on implementing these callbacks. Note that supported capability means that the

stack understands what that capability is and how to negotiate it. It does not mean that the GSM codec or G.729 codec is provided with the stack.

Enable/Disable DTMF Support

The stack currently supports H.245 based alphanumeric as well as signal DTMF and also supports Q.931 based DTMF using keypad information element. If application developers are using their own RTP stack and have capability of rfc2833 based DTMF, then they can enable rfc2833 based DTMF capability support in the stack. Note the media library provided with the stack does not support rfc2833. Enabling rfc2833 based DTMF simply means adding that capability in the outgoing TerminalCapabilitySet message for the endpoint. The functions used to enable/disable DTMF support are:

```
int ooH323EpEnableDTMFH245Alphanumeric();
int ooH323EpDisableDTMFH245Alphanumeric();

int ooH323EpEnableDTMFH245Signal();
int ooH323EpDisableDTMFH245Signal();

int ooH323EpEnableDTMFRFC2833(int dynamicRTPPayloadType);
int ooH323EpDisableDTMFRFC2833();

int ooH323EpEnableDTMFQ931Keypad();
int ooH323EpDisableDTMFQ931Keypad();
```

Gatekeeper Support

If an H.323 application wants to use services of a gatekeeper, the gatekeeper client should be initialized, which handles communication with the gatekeeper on behalf of the H.323 application. The API function to initialize gatekeeper client is shown below.

```
int ooGkClientInit(enum RasGatekeeperMode eGkMode, char *szGkAddr,
                   int iGkPort);
```

The first parameter is *eGkMode*, which specifies whether gatekeeper has to be discovered (RasDiscoverGatekeeper) or a specific gatekeeper has to be used (*RasUseSpecificGatekeeper*). The second and third parameters are used to specify the IP address and port number of the gatekeeper. These parameters are valid only if the gatekeeper mode is set to specific gatekeeper. Otherwise, they are ignored by the stack.

Creating H.323 Listener

Next step is to create an H.323 listener to accept incoming connection requests. All that is required to start the listener is a call to the following function:

```
int ooCreateH323Listener ()
```

As you can see, this function takes no arguments; it just starts the listener service. On success, it returns OO_OK and on failure to create listener, it returns OO_FAILED.

Creating Command Listener

The application interacts with stack thread in two ways, through callbacks and by issuing commands. For issuing commands (ex. MakeCall, AnswerCall etc.) to stack thread, application uses stack command API defined in ooStackCmds.h file. This API communicates stack commands to stack thread over a TCP connection. Hence, before issuing any stack command it is necessary that stack thread is listening for

incoming TCP connection for command channel on a specific port. By default port number 7575 is used, but applications can specify another value. The IP address for TCP connection is same as signaling IP address. Hence it is required that signaling IP should be set before creating command listener. To create a command listener following function is used:

```
/* Pass 0 to use default port number 7575 */
int ooH323EpCreateCmdListener(int cmdPort);
```

Starting Stack Thread

The OOH323C stack is designed to run in a single thread. This thread uses *select* socket call to monitor all the channel sockets and handle socket events as they occur. To run OOH323C stack, an H.323 application will typically create a separate thread and call following API function:

```
int ooMonitorChannels ( )
```

This function runs in an infinite loop, handling all the active calls and calling application registered callback functions as and when required. The applications main thread communicates with the stack by means of stack commands described in section 2.1. These commands are queued into a mutex protected queue, from where the stack thread retrieves them and processes.

Initiating a Call

"Initiating a call" refers to the procedure to open channels for any type of media communications – not just audio. A call can be initiated to send video or data as well. The *ooMakeCall* function is used for this purpose. Its calling sequence is as follows:

```
ooMakeCall (dest, callToken, bufsiz, opts);
```

The following arguments are passed to this function:

dest – An identifier of the destination endpoint to be called. For example, an IP address and port.
callToken – A unique token identifier returned to identify the call
bufsiz – The size of the callToken buffer
opts – A reference to ooCalloptions structure which is defined in ootypes.h. This structure is used to override endpoint settings for a particular call. If non-NULL value is passed, the options specified in the structure will be applicable for the new call.

Closing a Call

Either side of an H.323 connection can terminate a call. The function used to do this is *ooHangCall*. Its calling sequence is as follows:

```
ooHangCall (callToken);
```

The following arguments are passed to this function:

callToken – The unique token identifier returned to identify the call. This was set in the call to ooMakeCall.

Shutting Down the Stack

The application can call the "ooStopMonitor" function to shut the stack down. This will cause the "ooMonitorChannels" function to exit. Once this happens, the application should call to the "ooH323EpDestroy" function at the end of their program to do a necessary cleanup associated with the endpoint.

7. Application-Stack Interaction

This section explains how the interaction between an application and stack takes place though commands and callbacks. For this we consider an example flow of an outgoing call with faststart disabled below.

Sample Outgoing Call flow without faststart

1. The Application after initializing H.323 endpoint structure and starting stack thread, places an out going call by using stack command *ooMakeCall*. When this API funciton returns, the application gets callToken, which uniquely identifies the call. Refer section 2.1 for details
2. The stack processes the application request, allocates a new call structure and calls callback function *onNewCallCreated*, if the application has registered this callback function with the stack.
3. In the *onNewCallCreated* callback function, application can override aliases set for the endpoint, add new capabilities to be used only for this new call, instead of using endpoint capabilities. These things can be done with the help of CallManagement functions of the API described in reference manual.
4. The stack builds SETUP message to be sent and if application has registered *onBuiltSetup* message callback function, calls it.
5. Application can modify the outgoing SETUP message in *onBuiltSetup* callback if it so desires.
6. The stack sends SETUP message and calls *onOutgoingCall* callback function, if application has registered such a function.
7. Application can do what ever it wants in *onOutgoingCall* callback function. For example, it can print a display message indicating that it's calling the remote phone.
8. When the remote endpoint sends Alerting message, the stack calls *onAlerting* callback, if it is registered.
9. Application can display remote ringing message in *onAlerting* callback or do what ever it wishes.
10. When the remote endpoint sends CONNECT, stack processes it and calls *onReceivedConnect* callback function of the application.
11. Applications can process the received connect message in the *onReceivedConnect* callback, if it so desires. The applications would typically use this callback to process non-standard part of the received Connect message, which is not handled by stack.
12. The stack will then call *onCallEstablished* callback function.
13. In *onCallEstablished* callback application can display call established to it's user.
14. At this point stack will start H.245 capability exchange and master-slave determination with remote endpoint.
15. After successful capability exchange and master-slave determination, if application has registered *openLogicalChannels* callback, it will be called. Otherwise, stack will start sending OpenLogicalChannel requests based on call mode(Refer section 6.2 for details).
16. The next step is to start media transmission/reception. For this the stack calls channel callback functions registered by the application for the corresponding capability.

    ===== Media is Established at this point ====

17. To hang-up the call normally, the H.323 application will use ooHangCall command with call end reason parameter set to OO_REASON_LOCAL_CLEARED.
18. The stack will send H.245 EndSession command to remote endpoint and call channel callback functions *stopTransmitChannel, stopReceiveChannel* as appropriate to stop media (Refer section 2.2.3).
19. The stack will then send ReleaseComplete message to remote endpoint and call the callback function *onCallCleared*.

20. The application can do post call processing, if any, in *onCallCleared* callback function.
21. The stack will then delete the call structure.