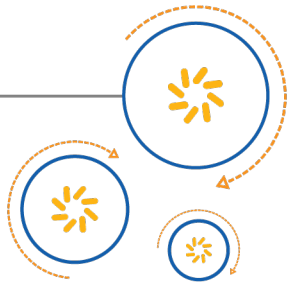
Qualcomm Technologies International, Ltd.



Introduction to the Audio Library

Application Note

80-CT404-1 Rev. AL

October 26, 2017

Revision history

Revision	Date	Description
1	OCT 2007	Original publication of this document. Alternative document number CS-00115065-AN.
2	NOV 2007	References to GAVDP library removed.
3	DEC 2007	Minor editorial correction
4	OCT 2008	Minor formatting corrections
5	MAR 2010	Updated for 2010 SDKs and to latest style guidelines.
6	JUL 2011	Descriptions of `ixia_locid="23">AudioSetRoute`, `AudioVoicePromptInit`, `ixia_locid="25">AudioSetPower` and `AudioMicSwitch` functions added. Updated to latest CSR™ style.
7	JAN 2012	Updated to latest CSR style.
8	MAY 2015	Updated to latest CSR style.
9	SEP 2016	Updated to conform to QTI standards; no technical content was changed in this document revision.
10	APR 2017	Updated to add new functions. Added to the Content Management System.
AL	OCT 2017	Document Reference Number updated to use Agile numbering. No change to technical content.

Contents

Revision history 2

1 Overview of the audio library 6

1.1 Audio library plug-ins 6

2 Application library architecture 8

2.1 Audio library functionality 8

3 Use of the audio library in VM applications 10

3.1 AudioConnect function 10

3.2 AudioDisconnect function 10

3.3 AudioDisconnectInstance function 10

3.4 AudioSetVolume function 10

3.5 AudioSetGroupVolume function 11

3.6 AudioSetMode function 11

3.7 AudioPlayTone function 11

3.8 AudioStopToneAndPrompt function 11

3.9 AudioPlayAudioPrompt function 11

3.10 AudioVoicePromptsInit function 11

3.11 AudioSetPower function 12

3.12 AudioMicSwitch function 12

3.13 AudioSetMusicProcessingEnhancements function 12

3.14 AudioConfigureSubWoofer function 12

3.15 AudioSetSoftMute function 12

3.16 AudioStartForwarding function 12

3.17 AudioStopForwarding function 12

3.18 AudioOutputSwitch function 13

3.19 AudioStartASR function 13

3.20 AudioSetInputAudioMute function 13

3.21 AudioSetMaximumConcurrentAudioConnections function 13

3.22 AudioSetUserEqParameter function 13

3.23 AudioApplyUserEqParameters function 13

3.24 AudioClearUserEqParameters function 13
3.25 AudioSetTwsChannelModes function 13
3.26 Typical message sequence chart for a simple audio request 14
4 Writing audio plug-ins 15
Document references 17
Terms and definitions 18

Figures

Figure 2-1: General architectural overview..... 8
Figure 2-2: Audio library architectural interfaces..... 9
Figure 3-1: Audio library message sequence chart..... 14

1 Overview of the audio library

The audio library simplifies audio connections in Virtual Machine (VM) applications. Previously audio was highly integrated and different in each application. This meant that audio handling code could not be shared in different applications. It also made it difficult for third parties to provide DSP audio algorithms that are application independent.

The audio library allows third party DSP algorithm vendors or system integrators to easily create DSP features that can be re-used in different applications.

The audio library:

- Simplifies audio connections for application integrators.
- Provides a uniform API.
- Contains simple functions that allow the application integrator to manage audio connections from the VM application.
- Allows third party vendors to develop audio plug-ins that engineers can readily incorporate into applications.
- Supports:
  - Synchronous connections, that is, SCO, eSCO
  - Transparent Synchronous connections, that is AuriStream, SBC for wide-band-speech
  - AV audio, for example SBC/MP3
  - Tone mixing
- Allows run-time inclusion of third party algorithms.
  The engineer can select between multiple audio plug-ins from within the VM application at the point of connection.
- Removes the audio connection code from the VM application.
  The connection and audio processing is now handled by the audio library and the relevant plug-in library. This simplifies the application coding requirements.

1.1 Audio library plug-ins

The audio library relies on the concept of audio plug-ins. A plug-in can be loaded at run time to allow a specific audio component to be used.

For example, a headset may have two modes of operation, through earphones or in a docking station. When the Bluetooth audio connection is made, a different audio plug-in can be used in each mode to provide different audio behavior.

An audio plug-in is a VM library that meets the API for the audio library and provides some sort of audio connection mechanism.

In the case of a DSP algorithm, the plug-in also uses a DSP application (either a source code application built inside the project workspace or a `.kap` file placed in the image directory) that can provide some form of audio enhancement algorithm using the DSP. In this case, the audio plug-in can be considered to be a combination of the VM library and the DSP application.

Plug-ins can be written by anyone. In general, plug-ins are written by QTIL to provide standard functionality. In addition to this, third party vendors can also provide plug-ins as part of the eXtension Partner Program. These plug-ins provide various audio enhancement features. Once written, a plug-in can be incorporated into any VM application written using the audio library to handle audio connections.

System Integrators can select from the available one or more plug-ins to provide the audio enhancement that suits their application or write their own plug-in to meet their own requirements. See Writing audio plug-ins.

2 Application library architecture

The figure gives a conceptual representation of library interaction within Qualcomm® BlueCore™ technology applications. This should not be taken as a literal interpretation but is useful in visualizing dependencies and interfaces.

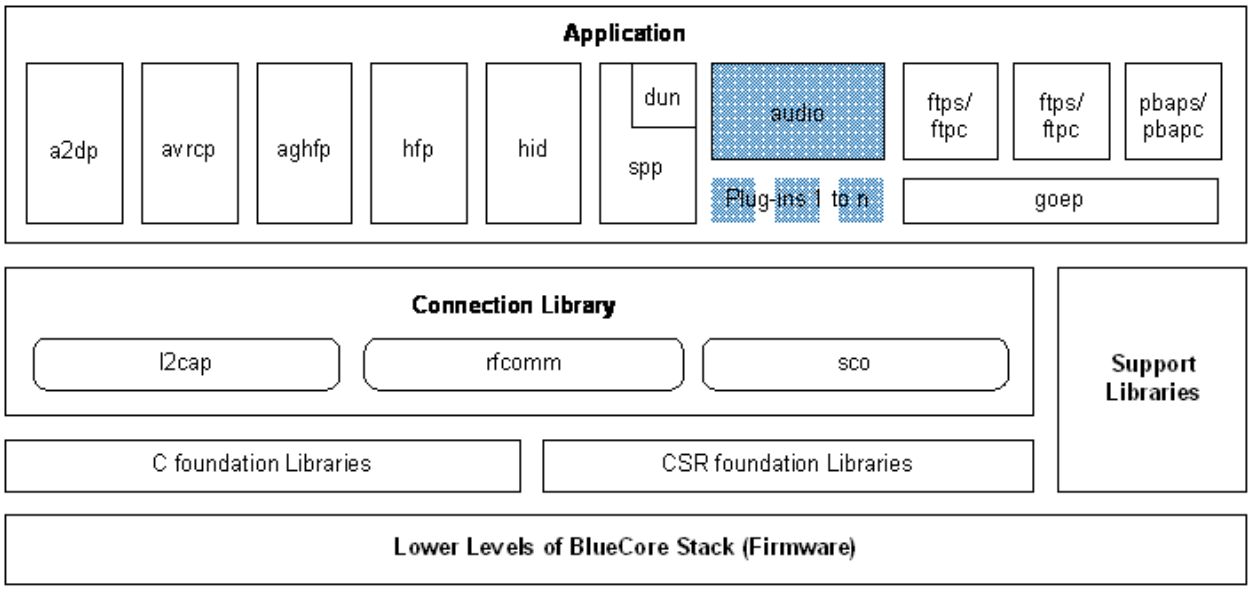


Figure 2-1 General architectural overview

The VM application calls the audio library to select the required plug-in that then handles the connections to the audio subsystem.

The `audio` library then routes these calls to the required plug-in.

The plug-in is then responsible for performing the action associated with the initial function call made by the application e.g. to connect up the audio streams as defined in the plug-in.

2.1 Audio library functionality

The `audio` library provides two fixed APIs:

- A fixed functional API between the application and the audio library that the application writer uses to add audio functionality in the application.
- A fixed message based API between the audio library and the plug-ins, This is used by engineers wishing to write a plug-in that works with the `audio` library.

The `audio` library:

- Presents a uniform interface to the application layer and to the plug-ins
- Queues audio function calls for delivery to the plug-in.

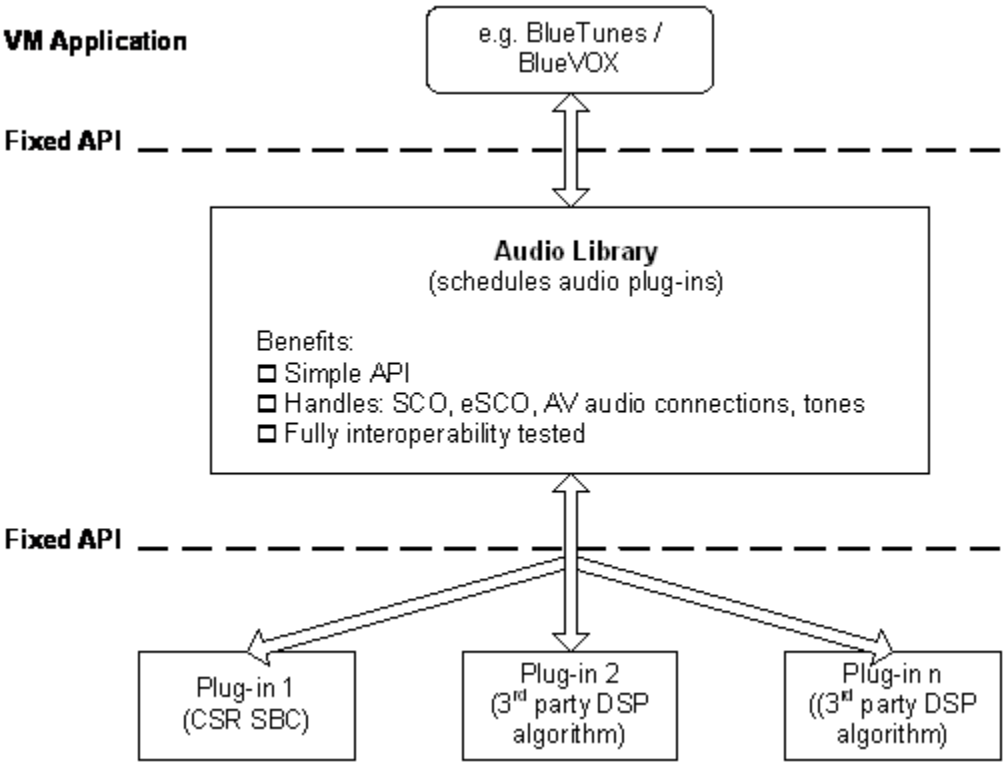


Figure 2-2 Audio library architectural interfaces

The `audio` library works by scheduling calls to the underlying audio plug-ins. The application writer can call a series of audio functions to be handled by the different underlying audio plug-ins. The order of these calls is preserved by the audio library.

The audio_busy flag:

The global `audio_busy` variable is used to schedule the calls to the underlying plug-ins. This is done by queuing all plug-in messages using the VM trap `MessageSendConditionally()` conditional on the `audio_busy` variable.

3 Use of the audio library in VM applications

The `audio` library is exposed to the VM application by a uniform API

The audio library functions are used in VM application. For details on function parameters, see xIDE on-line help and *VM and Native Reference Guide/audio.h*.

3.1 AudioConnect function

`AudioConnect` connects an audio stream to an underlying audio plug-in. The plug-in is then responsible for connecting the audio sink as dictated by the arguments passed with the function.

When an audio connection has been established to a plug-in or when a tone is being played, the `AUDIO_BUSY` flag is set. When this flag is set, any attempt to connect audio is queued in the audio library scheduler, that is, only one audio connection can be handled at one time.

If multiple calls to connect audio streams are queued they are actioned in sequence as and when the previous audio is disconnected.

It is therefore important that each call to `AudioConnect` is followed by a call to `AudioDisconnect`.

3.2 AudioDisconnect function

`AudioDisconnect` disconnects an audio stream previously connected using `AudioConnect`. It is the plug-in's responsibility to correctly handle the disconnect.

If `AudioDisconnect` is called when no plug-in is connected it is ignored.

3.3 AudioDisconnectInstance function

`AudioDisconnectInstance` disconnects an audio stream associated with the specified audio instance.

If `AudioDisconnectInstance` is called when no plug-in is connected, it is ignored.

3.4 AudioSetVolume function

`AudioSetVolume` is called by the VM application to update the volume of the currently connected audio.

The volume is set by the underlying plug-in which, may choose to interpret the request as appropriate to the specific plug-in functionality. For example, some plug-ins may interpret it as a codec gain, while some may choose to ignore the message generated by the call.

NOTE The initial volume setting is passed in to the plug-in as an argument when calling `AudioConnect`.

3.5 AudioSetGroupVolume function

`AudioSetGroupVolume` is called by the VM application to set the group volume, that is the main volume or aux volume or both, in increments of 1/60th of a dB

3.6 AudioSetMode function

`AudioSetMode` is called to update the mode of the currently connected audio.

NOTE QTIL are phasing out the use of this function, and it is not recommended that it is used when coding new Plug-ins.

3.7 AudioPlayTone function

`AudioPlayTone` is called to request that a tone is played.

The tone can be played using the separate `csr_voice prompts_plugin` or using a connected plug-in that supports tone mixing.

NOTE All QTIL plug-ins support tone mixing.

Tones can be queued. Queued tones are played sequentially as and when the previous tone completes.

3.8 AudioStopToneAndPrompt function

`AudioStopToneAndPrompt` is called to stop a tone that is being played. For example, to terminate a headset generated ring tone when a call is answered.

NOTE The implementation of `AudioStopToneAndPrompt` is plug-in specific. Some plugins may choose to ignore the request to stop playback of a tone.

3.9 AudioPlayAudioPrompt function

`AudioPlayAudioPrompt` is called to play a text to speech command. This may take the form of a fixed phrase, a name or digits.

NOTE This behavior is defined by the plug-in itself.

3.10 AudioVoicePromptsInit function

`AudioVoicePromptsInit` is called to initialize the underlying voice prompts storage.

Voice prompts are stored in the memory sequentially. For example if there are two voice prompts and three languages, the voice prompt sequence in memory is:

```
language0_prompt0, language0_prompt1, language1_prompt0,
language1_prompt1, language2_prompt0, language2_prompt1
```

3.11 AudioSetPower function

`AudioSetPower` is called to control the power usage of the connected plug-in. This allows the application to change the `AUDIO_POWER_MODE_T` power of the connected plug-in while the plug-in is connected. This is used by the application as part of the Low Battery Intelligent Power Mode (LBIPM) feature.

NOTE Not all plug-ins support this feature by default.

3.12 AudioMicSwitch function

`AudioMicSwitch` is called to change the connected microphone of the underlying plug-in. This is used by Qualcomm® cVc™ noise cancellation technology during production test and in the QTIL headset application can be configured to occur remotely using the `+MICTEST` AT command.

3.13 AudioSetMusicProcessingEnhancements function

`AudioSetMusicProcessingEnhancements` is called to toggle the status of the music processing functionality, that is either enabled or disabled.

NOTE `music_enhancements` is passed as a parameter in the `AudioConnect` function.

3.14 AudioConfigureSubWoofer function

AudioConfigureSubWoofer is called to set operating mode of the sub-woofer of the soundbar. The Soundbar application has three operating mode:

- `AUDIO_SUB_WOOFER_NONE`: No sub-woofer connection
- `UDIO_SUB_WOOFER_ESC`O: Sub-woofer connection through eSCO 3
- `AUDIO_SUB_WOOFER_L2CAP`: Sub-woofer connection through L2CAP

3.15 AudioSetSoftMute function

`AudioSetSoftMute` toggles the mute state of the audio using soft mute.

If `AudioSetSoftMute` is called when no plug-in is connected, it is ignored.

3.16 AudioStartForwarding function

`AudioStartForwarding` starts forwarding undecoded audio frames to the specified sink.

If `AudioStartForwarding` is called when no main plug-in is connected, it is ignored.

3.17 AudioStopForwarding function

`AudioStopForwarding` stops forwarding of undecoded audio frames.

If `AudioStopForwarding` is called when no main plug-in is connected, it is ignored.

3.18 AudioOutputSwitch function

`AudioOutputSwitch` swaps output during production testing.

3.19 AudioStartASR function

`AudioStartASR` starts/restarts the ASR engine.

If `AudioStartASR` is called when no plug-in is connected, it is ignored.

3.20 AudioSetInputAudioMute function

`AudioSetInputAudioMute` mutes/unmutes the input audio port for all audio sources except tones.

If `AudioSetInputAudioMute` is called when no plug-in is connected, it is ignored.

3.21 AudioSetMaximumConcurrentAudioConnections function

`AudioSetMaximumConcurrentAudioConnections` is called to set the number of concurrent audio connections that the application supports.

NOTE Applications based on BlueCore architecture do not supports concurrent audio connections, so there is no effect on calling `AudioSetInputAudioMute`.

3.22 AudioSetUserEqParameter function

`AudioSetUserEqParameter` is called to set an individual user PEQ parameter.

NOTE Only user PEQ bank 1 can be updated.

3.23 AudioApplyUserEqParameters function

`AudioApplyUserEqParameters` applies the stored set of EQ parameters and clears them from the store.

3.24 AudioClearUserEqParameters function

`AudioClearUserEqParameters` clears the stored set of EQ parameters.

3.25 AudioSetTwsChannelModes function

`AudioSetTwsChannelModes` is called to set the required Channel mode for given music inputs.

3.26 Typical message sequence chart for a simple audio request

Figure 3-1 shows a typical message sequence for a simple audio request by an application.

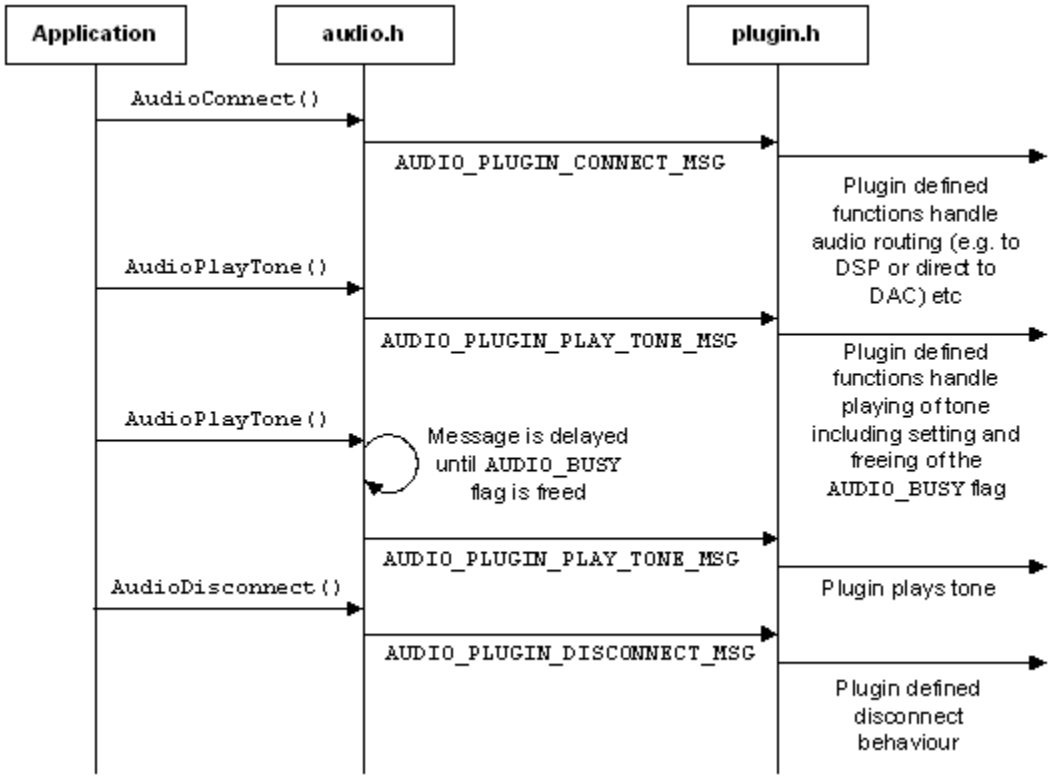


Figure 3-1 Audio library message sequence chart

4 Writing audio plug-ins

An audio plug-in consists of a VM library and/or a DSP application that meets the API for the `audio` library and provides some sort of audio connection mechanism.

A plug-in is also a QTIL SDK library with the naming convention:

`companyname_action_plugin , for example. csr_cvc_common_plugin`

The name of the folder must also follow the library naming convention above. All the c and h files that are part of the library must also follow the library naming convention. This is a requirement of the QTIL SDK toolchain.

The plug-in VM library should be added to the **<SDKName>\\src\lib** directory and can be built in the same way as other QTIL SDK VM libraries.

The header file with the same name as the library folder is then added to the public header file used for the QTIL SDK library. This is automatically placed in the **<SDKName>\tools\include\profiles\<SDK>** directory.

In general, the plug-in VM library contains at least two pairs of files. The interface to the audio library and the implementation of the audio connection are handled in separate files. Other files may be needed for more complicated audio plug-ins.

From the earlier example:

- Interface:
  - `csr_cvc_common_plugin.h`
  - `csr_cvc_common_plugin.c`
- Implementation:
  - `csr_cvc_common.h`
  - `csr_cvc_common.c`

The Interface contains a const task that receives and handles messages from the `audio` library.

NOTE A task is basically a message handling function and a structure containing the task's current state.

The message loop is used to route or schedule the messages based upon the state of the `audio_busy` flag in the same way as in the audio library. This is used to preserve the order of the application calls.

The messages the plug-in's task can receive and must handle correspond to the audio API functions, that is:

- `AUDIO_PLUGIN_CONNECT_MSG`
- `AUDIO_PLUGIN_DISCONNECT_MSG`
- `AUDIO_PLUGIN_SET_VOLUME_MSG/AUDIO_PLUGIN_SET_GROUP_VOLUME_MSG`
- `AUDIO_PLUGIN_PLAY_TONE_MSG`
- `AUDIO_PLUGIN_STOP_TONE_AND_PROMPT_MSG`
- `AUDIO_PLUGIN_PLAY_AUDIO_PROMPT_MSG`

The `audio_busy` flag can be used by an audio plug-in whenever the plug-in does not want to receive actions. This is usually the case when a tone is playing.

If the `audio_busy` flag is set when a message is received, then the message is conditionally queued until the flag is cleared. When a message is delivered the action specified in the message can take place.

For example, if a tone has started playing in a plug-in, then the plug-in does not want to be disconnected while the tone playback is in progress. In this case, the plug-in can set the value of the `audio_busy` flag for the duration of the tone playback. When the tone has completed, then the `audio_busy` flag can be released and any outstanding messages are delivered.

The `audio_busy` flag can also be set by the plug-in at other times to prevent the plug-in being interrupted, for example, the `csr_cvc_common_plugin` uses the `audio_busy` flag to prevent actions occurring while the Qualcomm® cVc™ noise cancellation technology algorithm is being initialized.

NOTE It is not advisable to set the `audio_busy` flag for long periods of time as this can cause a backlog of messages in the system.

Playing tones

Tones must be handled in the plug-in when the plug-in is connected to an audio stream.

If the plug-in decides not to play back a tone, then the plug-in must clear the `audio_busy` flag so that messages can continue to be received.

Stopping tones

Stopping a tone uses the `audio_busy` flag in a different way to the other functions, that is, the tone only needs to be stopped if it is still currently playing. Thus the `audio` library only sends the tone stop message to the plug-in when the `audio_busy` flag is set.

Document references

Document	Reference
A guide to Qualcomm Bluetooth Libraries	80-CT436-1/CS-00207480-UG
Introduction to the Audio Library	80-CT404-1/CS-00115065-AN

Terms and definitions

Term	Definition
A2DP	Advanced Audio Distribution Profile
ADC	Analog to Digital Converter
AGHFP	Audio Gateway Hands Free Profile
API	Application Programming Interface
BlueCore	Group term for the range of QTIL Bluetooth wireless technology ICs
Bluetooth SIG	Bluetooth Special Interest Group
Bluetooth	Set of technologies providing audio and data transfer over short-range radio connections
CaSiRa	QTIL Bluetooth development hardware
DSP	Digital Signal Processor: a microprocessor dedicated to real-time signal processing.
DUN	Dial Up Networking
FTP	File Transport Protocol
GAVDP	Generic Audio Visual Distribution Profile
GOEP	Generic Object Exchange Profile
HCI	Host Controller Interface
HFP	Hands Free Profile
HID	Human Interface Device Profile
I2S	Inter-Integrated Circuit Sound
IC	Integrated Circuit
L2CAP	Logical Link Controller and Adaptation Protocol
LBIPM	Low Battery Intelligent Power Mode
MMI	Man Machine Interface
OPP	Object Push Profile
PBAP	Phonebook Access Profile
QTIL	Qualcomm Technologies International, Ltd.
RFCOMM	Serial Cable Emulation Protocol based on ETSI TS 07.10.
SCO	Synchronous Connection Orientation link
SDP	Service Discovery Protocol
SPP	Serial Port Profile

Term	Definition
SPDIF	Sony/Philips Digital InterFace (also IEC 958 type II, part of IEC-60958). An interface designed to transfer stereo digital audio signals between various devices and stereo components with minimal loss.
VM	Virtual Machine; environment in the BlueCore firmware for running application-specific code produced with BlueLab
xIDE	The QTIL Integrated Development Environment