VoIP Engine V4 Software Users Guide

Adaptive Digital Technologies, Inc.
April 30, 2013

Revision History

Issue	Description	Date	Engineer
1	Initial Version – Adapted from VoIP Engine (V2) Users Guide	7/15/2011	SDK
2	Version 3.10	10/11/2011	SDK
3	Version 3.11	10/14/2011	SDK
4	Version 4 – ALPHA	4/20/2012	SDK
5	Documentation update only.	6/15/2012	SDK
6	AEC Default Canges	6/19/2012	SDK
7	Fixed VE_ADT_setGain documentation. Added VE_ADT_setRxPathGain documentation. Removed definition of VEPARAM_RX_PACKET_GAIN	6/21/2012	SDK
8	More detailed test point documentation	7/2/2012	SDK
9	PCM word width is fixed at 16-bits. Updated documentation to reflect that	8/17/2012	SDK
10	Log filter, engine number additions	9/13/2012	CPB
11	V4.20 Incorporarted AECG4-Mobile RTP update Various bug fixes Removed section discussing self-training Removed section discussing tuning procedure	11/21/2012	SDK
12	Added support for AMR-NB and AMR-WB	2/1/2013	BMY
13	Note that VE_ADT_usMalloc needs to zero out the allocated buffer	3/21/2013	SDK
14	April 5, 2013 Pre-Release	4/5/2013	SDK
15	April 9, 2013 Pre-Release	4/9/2013	SDK
16	Version 4.40	4/24/2013	SDK
17	Version 4.41	4/30/2013	SDK

Table of Contents

1. Introduction

This document describes the ADT VoIP Engine software. The VoIP Engine software is a software engine that handles all the voice processing from PCM to Packet and back. Its intended use is in VoIP enabled handsets or desktop phones. Although VoIP Engine is not tied to any particular software environment, it was designed with Android and iPhone in mind.

VoIP Engine includes the following features:

- PCM Front End (Independently Accessible)
    - Acoustic Echo Cancellation
    - Noise Reduction
    - Tone Generation
    - Gain Control
    - Automatic gain control
    - Diagnostics to assist in acoustic tuning
    - Equalization
- VoIP Handler
    - G.711 with appendices 1 (packet loss concealment) and 2 (discontinuous transmission)
    - G.729A Vocoder
    - G.722 (wideband audio) with packet loss concealment
    - G.722.2 Vocoder (Optional)
    - AMR Vocoder (Optional)
    - RTP/Jitter Buffer
    - SRTP
    - DTMF keypress tone relay transmit (IETF RFC2833)
- Conferencing

Future enhancements will include:

- G.711 Appendix 2
- G.729AB (with Appendix B)
- Plug-in Codecs

1.1 Acronyms / Definitions

AEC – Acoustic Echo Canceller
AGC – Automatic Gain Control
EQ – Equalizer
NR – Noise Reduction
PCM – Pulse Code Modulation
Receive (Direction) – Data flow from the packet network toward the speaker
Send (Direction) – Data flow from microphone toward packet network
TG – Tone Generator
RTP – Real-Time Transport Protocol
SR – Sampling Rate
SRTP – Secure RTP
VoIP - Voice Over IP
VQE – Voice Quality Enhancement

2. VoIP Engine Architecture

2.1 Data Flow

The VoIP Engine is divided into three parts – the PCM Front End, conferencing (optional), and one or more packet channels. This is shown in the block diagram below.

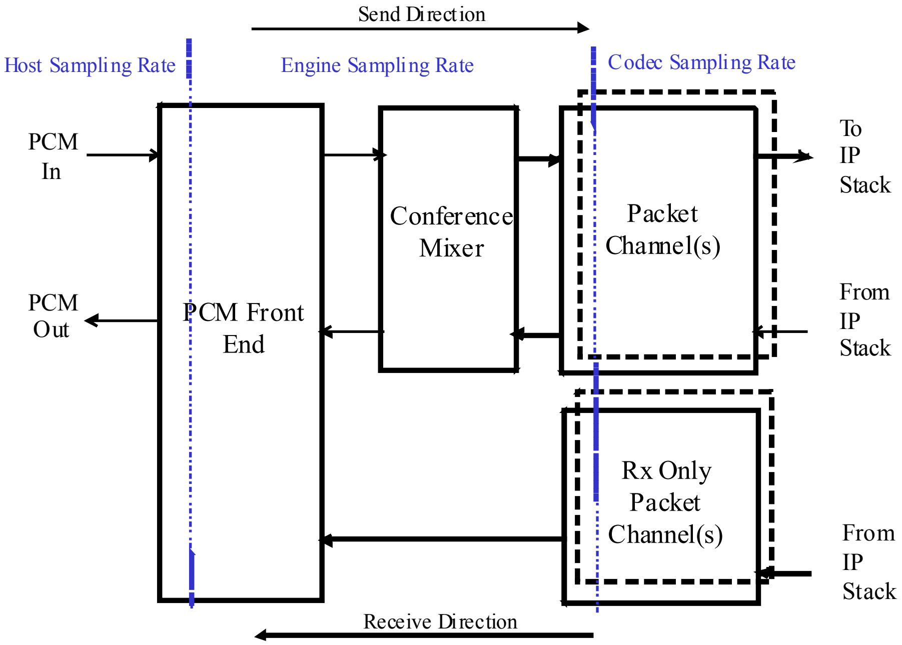
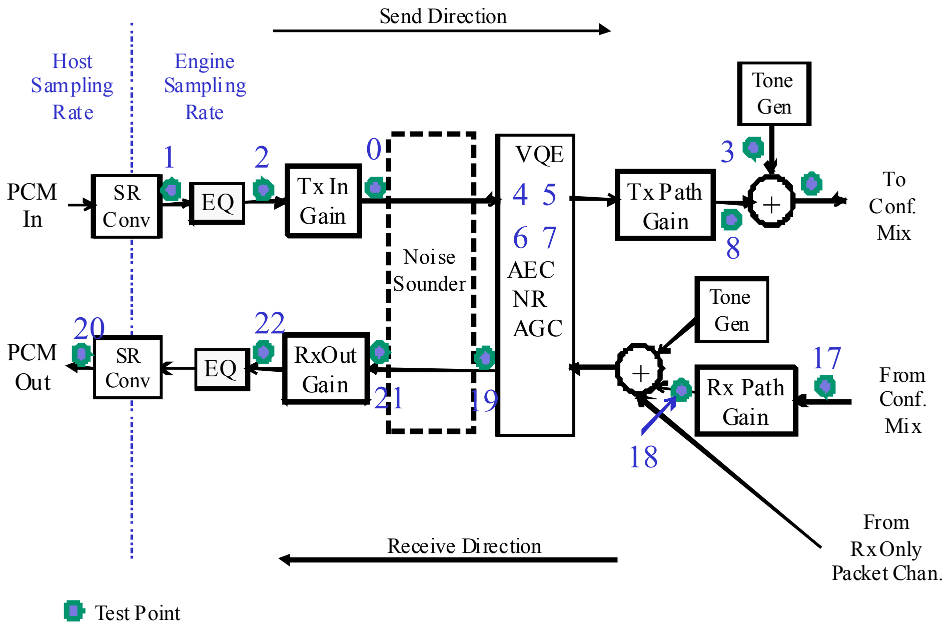


Figure 2-1 VoIP Engine Block Diagram

The PCM Front End is expanded in the block diagram below. It should be noted that the PCM front end can be accessed independently of the VoIP portion of VoIP Engine in systems that do not require the VoIP functionality or in systems that already have the VoIP functionality included.

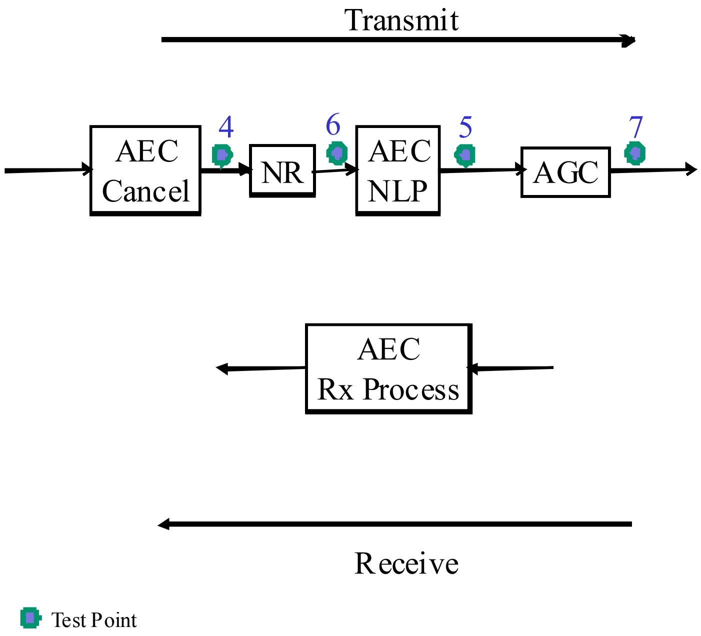


Test Point Key:

```
TP_PCMTX_MIC_FIXED_GAIN_OUT: 0,
TP_PCMTX_SAMPLING_RATE_CONVERSION_OUT: 1
TP_PCMTX_EQ_OUT: 2
TP_PCMTX_TG_OUT: 3
TP_PCMTX_AEC_PRENLP_OUT: 4
TP_PCMTX_AEC_OUT: 5
TP_PCMTX_NR_OUT: 6
TP_PCMTX_AGC_OUT: 7
TP_PCMTX_PATH_GAIN_OUT: 8
TP_PCMRX_IN: 17
TP_PCMRX_PATH_GAIN_OUT: 18
TP_PCMRX_AECRX_OUT: 19
TP_PCMRX_AEC_SAMPLING_RATE_CONVERTER_OUT: 20
TP_PCMRX_SPEAKER_GAIN_IN: 21
TP_PCMRX_SPEAKER_GAIN_OUT: 22
```

Figure 2-2 PCM Front End Block Diagram

The VQE (Voice Quality Enhancement) block is shown in greater detail in figure 2-3 below.

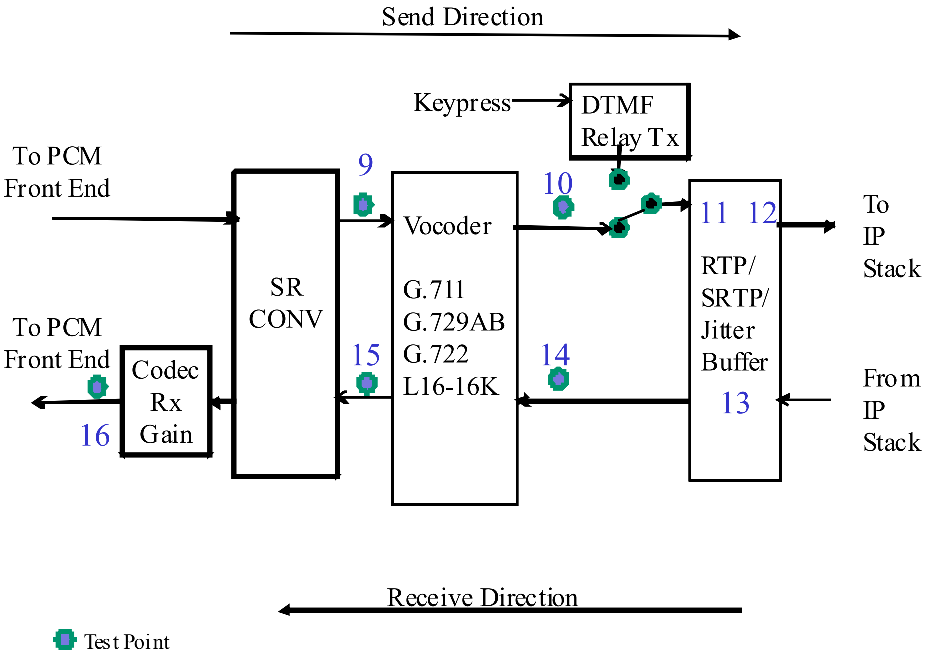


Test Point Key:

```
TP_PCMTX_AEC_PRENLP_OUT: 4
TP_PCMTX_AEC_OUT: 5
TP_PCMTX_NR_OUT: 6
TP_PCMTX_AGC_OUT: 7
```

Figure 2-3 PCM Front End Block Diagram

The packet channel(s) handle the conversion between VoIP packets and PCM data. In the transmit direction, PCM data from either the PCM front end, or from the conferencing block when a conference is active, is fed to each packet channel, where compression and RTP encapsulation is performed. RTP packets are optionally encrypted using SRTP. The RTP packets are sent to the IP stack for transmission. In order to support reliable DTMF transmission, a the DTMF relay transmit block is available to convert between keypress/depress into RTP tone payloads.



Test Point Key

```
TP_PKTTX_ENCODE_IN: 9
TP_PKTTX_ENCODE_OUT: 10
TP_PKTTX_RTPSEND_OUT: 11
TP_PKTTX_SRTP_OUT: 12
TP_RTPTOE_SRTP_OUT: 13
TP_PKTRX_RTP_PAYLOAD_OUT: 14
TP_PKTRX_DECODE_OUT: 15
TP_PKTRX_OUT: 16
```

Figure 2-4 Packet Channel Block Diagram

Software Architecture

The figure below depicts how the VoIP Engine fits into a VoIP enabled host application.

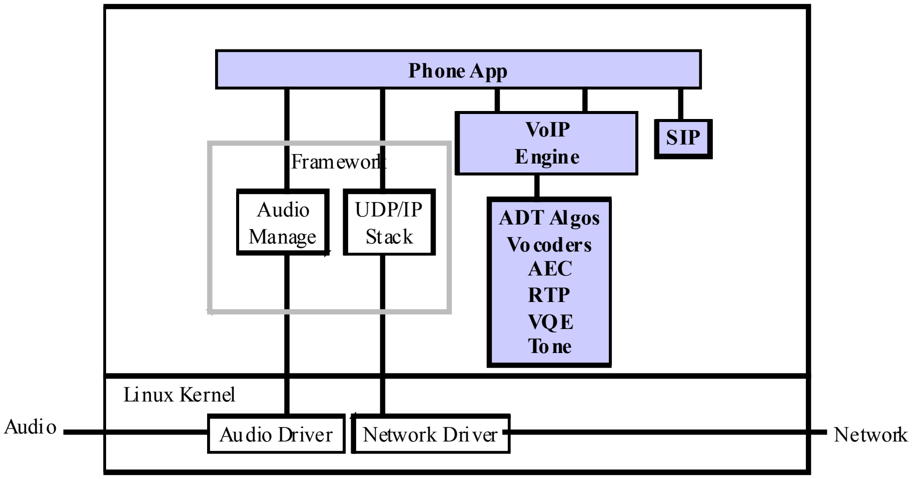


Figure 2-4 – System Architecture

The VoIP engine is purely a data processing engine. It has no interface to drivers or peripherals and performs processing solely at the request of the host application. The host application feeds the VoIP engine PCM samples from the audio input and and RTP packets from the network input. The VoIP engine in turn returns, via callbacks to the host application, PCM samples to be sent to the audio output device and RTP packets to be sent to the network interface.

The VoIP Engine processing is based upon PCM frames. Allowable frame sizes are 10, 20, and 40 milliseconds.

Data flow for iPhone and Android is described in the following two sections respectively.

2.1.1 Data Flow

In the send direction (Microphone PCM to Packet Network), the host software calls function VE_ADT_pcmToEngine( ) once per frame to feed the engine with a frame of PCM data. The engine processes the PCM data. If one or more new RTP packets are available for transmission, the engine will call the host-supplied callback function VE_ADT_cbOnSendRTPToNetwork( ) so the host application can send the RTP packet via the IP network stack.

The VE_ADT_pcmToEngine also processes data in the receive direction. If an RTP packet is available, the engine synthesizes a frame of speech. If no packet is available, the engine fills in the PCM frame using packet loss concealment and sends the frame to the host.

In the receive direction, the host application calls VE_ADT_rtpToEngine( ) function with each new RTP packet. Due to the jittery nature of the IP network, this function call is not necessarily in sync with the PCM framing. The packet is stored in the RTP/Jitter buffer. When using only the PCM Front End, VE_ADT_rtpToEngine is not called.

3. VoIP Engine API

3.1 A Note About API Functions

In order to make VoIP Engine multi-thread capable, the API functions have locking mechanisms. The API functions all make use of user-supplied VE_ADT_usLock and VE_ADT_usUnlock. These functions can be dummy functions when operating in a single-thread environment.

When locking is used, the API functions do block each other. The duration of a block should be limited to a fraction of the frame size. The function that blocks for the longest (by far) is VE_ADT_pcmToEngine since it handles all the heavy duty signal processing.

The API functions all perform a check to ensure that the passed Handle is valid and that the engine has been initialized. Each API function returns appropriate result codes when these checks fail. In order for these checks to be effective, it is critical that the host software call VE_ADT_create before calling any other VoIP Engine API functions. And the host software should not call any API functions after calling VE_ADT_delete.

To make the checks most effective, the user’s Handle variable should be set to zero after calling VE_ADT_delete.

3.2 VoIP Engine Object Life Cycle

The VoIP Engine can be looked at as an object that can be created, initialized, started, stopped, and deleted. The life cycle is shown in the figure below.

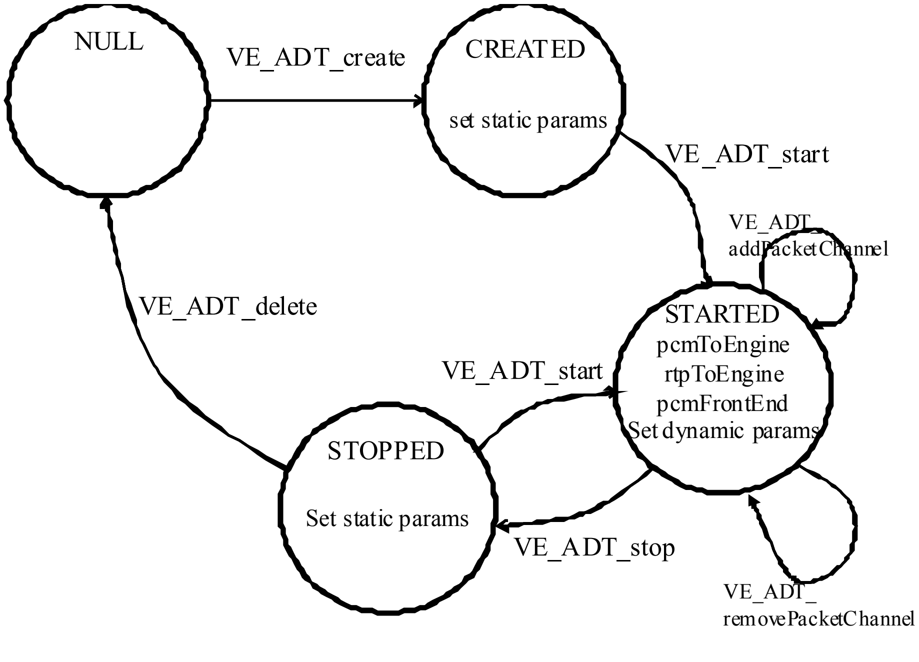


Figure 3-1 – VoIP Engine Object Life Cycle

A VoIP Engine object is created using VE_ADT_create. Once created, the user can configure the engine via numerous parameter settings. The function that sets parameters is VE_ADT_setParam. Unless the application changes parameter values, they are set to default values. In addition to VE_ADT_setParam, there are a few additional functions that are used to initialize items that have grouped parameters and would otherwise be cumbersome to set using individual parameters. These functions are discussed later in this document as well as the individual parameters are listed and explained later in this document.

While most parameters must be set before starting the engine, some can be modified after the engine has already been started. Similarly, some API functions can be called while the engine is already started and some cannot. Parameters and API functions will be identified by their associated type.Parameters that must be set before the engine is started are referred to as static parameters. Parameters that can be set after the engine has already been started are referred to as dynamic parameters.

To start the engine, the application calls VE_ADT_start. When in the started state, the application calls VE_ADT_pcmToEngine and VE_ADT_rtpToEngine (or VE_ADT_pcmFrontEnd if using PCM Front End only.) The application can also modify dynamic parameters during the started state.

By default, an instance of VoIP Engine includes a single packet channel unless it is configured for PCM Front End mode using the VE_ADT_setParams( ) function. After an engine has been started, additional packet channels can be added to form a conference. Packet channels can be added and removed during an active call as needed using VE_ADT_addPacketChannel( ) and VE_ADT_removePacketChannel( ) respectively.

If the engine wants to change a static parameter, the engine must first be stopped using the VE_ADT_stop function. During the stopped state, the static parameters can be modified. The engine can subsequently be restarted by using the VE_ADT_start function.

When the application wants to end a phone call, the engine is stopped. It may also be deleted, but that is not required. But either way, the engine should be deleted before the application exits.

3.3 VoIP Engine Parameters

VoIP Engine parameters are categorized by the module that they affect. Each module has an associated module ID. Similarly, each parameter has an associated Parameter ID. When modifying a parameter using VE_ADT_setParam (or reading back the value of a parameter using VE_ADT_getParam), the Module ID and Parameter ID need to be specified. Module IDs and Parameter IDs are found in voipengine_user_v4.h. Module IDs and Parameter IDs are listed below for convenience.

Module IDs:

- Engine (VEMODULE_VE)
- Handset AEC (VEMODULE_HANDSET_AEC)
- Handsfree AEC (VEMODULE_HANDSFREE_AEC)
- RTP (VEMODULE_RTP)
- (Others in the future)

Each module's parameters are described in the sections that follow. Default values are listed, when appropriate as is parameter type.

3.3.1 Engine Parameters

3.3.1.1 VEPARAM_CODEC

Description:

Selects the vocoder to be used in the transmit path

Parameter Type: static

Range of Values:

- VECODEC_PCMU (mu-law)
- VECODEC_PCMA (a-law)
- VECODEC_G722 (ITU G.722)
- VECODEC_G722_2 (Optional g.722.2)
- VECODEC_G729AB (ITU G.729AB)
- VECODEC_L16_16K (16-bit linear, 16 kHz sampling rate)
- VECODEC_L8_8K (16bit linear, 8 kHz sampling rate)
- VECODEC_AMR (Optional AMR)

Default Value: VECODEC_PCMU

3.3.1.2 VEPARAM_TEST_MODE

Description:

The VoIP Engine can be run in a number of different test/diagnostic modes.

Parameter Type: static

Range of Values:

- **VE_TM_OFF** - normal voice mode of operation
- **VE_TM_NOISE_SOUNDER** – used to determine the delay and loss/gain of the echo path. This mode of operation is described in greater detail later in this document
- **VE_TM_RX_TONE** – generate a test tone of 1004 Hz at a level of -10 dBm and send it to the speaker.
- **VE_TM_TX_TONE** – generate a test tone of 1004 Hz at a level of -10 dBm and send it to the VoIP processing block for transmission on the VoIP network
- **VE_TM_RX_CSS** – generate a CSS (Composite Speech Signal) and send it to the speaker
- **VE_TM_TX_CSS** – generate a CSS (Composite Speech Signal) and send it to the VoIP processing block for transmission on the VoIP network.
- **VE_TM_TX_DTMF_RELAY** – simulate a continuous series of DTMF digit key presses with cadence of 100 msec on, 100 msec off, and send the digits to the VoIP network via RFC 2833 (tone relay) packets
- **VE_TM_FIREHOSE** – this is a place holder for a test mode that is handled by Adaptive Digital's sample applications. In Firehose Mode, the test applications will duplicate each

transmit packet and send it on them using different RTP SSRCs. This is helpful for testing a gateway's channel capacity. Note that the number of channels is controlled by a separate engine parameter VEPARAM_FIREHOSE_CHANNELS

- **VE_TM_TX_TONE_SWEEP** – generate a tone with continuously increasing frequency and send it to the VoIP handler for transmission via the VoIP network. (Future)
- **VE_TM_RX_TONE_SWEEP** – generate a tone with continuously increasing frequency and send it to the speaker. (Future)

Default Value:

VE_TM_OFF

3.3.1.3 VEPARAM_ENGINE_FRAME_SIZE_SAMPLES

Description:

This parameter defines the number of samples per processing frame. Each frame of PCM is encoded into a single packet. It should be noted that the receive packet duration/frame size could be different since it that is being controlled by the equipment at the other end of the packet network.

Parameter Type: static

Range of Values: 80, 160, 320

Default Value: 160

3.3.1.4 VEPARAM_HOST_FRAME_SIZE_SAMPLES

Description:

This parameter defines the number of samples per frame that are sent from the host software to the engine. Typically this is the same as the engine size, but that is not always the case.

Parameter Type: static

Range of Values:

80..1000. But from a practical standpoint, the host frame size should reflect the packet duration. In other words, if the packet duration is 20 msec, and the host and engine sampling rates are 8 kHz, the engine frame size will be 160 samples. The host frame size should reflect this. If the host frame size is longer, the packet tranmissions will be more bursty. If the sampling rates are not equal, the frame durations should still be kept as close to each other as possible. For example, if the packet duration is 20 msec, the host sampling rate is 44.1 kHz, and the engine sampling rate is 8 kHz, the engine frame size would still be 160 samples, but the host frame size can be as high as 882 samples.

Default Value:

160

3.3.1.5 VEPARAM_TXMUTE

Description:

This Boolean parameter, when set, causes the transmit (microphone) path to be muted.

Parameter Type: dynamic

Range of Values: 0, 1

Default Value: 0

3.3.1.6 VEPARAM_RXMUTE

Description:

This Boolean parameter, when set, causes the receive (speaker) path to be muted.

Parameter Type: dynamic

Range of Values: 0, 1

Default Value: 0

3.3.1.7 VEPARAM_FIREHOSE_CHANNELS

This parameter has no effect on the engine. It is a placeholder for functionality that is used by Adaptive Digital's sample programs.

3.3.1.8 VEPARAM_NOISEREDUCTION

Description:

This Boolean parameter, when set, turns on noise reduction in the transmit path.

Parameter Type: static

Range of Values: 0, 1

Default Value: 1

3.3.1.9 VEPARAM_RX_PATH_GAIN

Description:

This parameter, specified in dB, controls a fixed gain that is placed in the receive path. The placement of the gain block can be seen in the PCM Front End block diagram in section 2 of this document. Loss can be specified using negative gain values.

Parameter Type: static

Range of Values: -12,.12

Default Value: 0

3.3.1.10 VEPARAM_RX_OUT_GAIN

Description:

This parameter, specified in dB, controls a fixed gain that is placed in the receive path prior to the speaker output. The placement of the gain block can be seen in the PCM Front End block diagram in section 2 of this document. Loss can be specified using negative gain values.

Parameter Type: static

Range of Values-12,.12

Default Value: 0

3.3.1.11 VEPARAM_TX_PATH_GAIN

Description:

This parameter, specified in dB, controls a fixed gain that is placed in the transmit path. The placement of the gain block can be seen in the PCM Front End block diagram in section 2 of this document. Loss can be specified using negative gain values.

Parameter Type: static

Range of Values: -12,.12

Default Value: 0

3.3.1.12 VEPARAM_TX_IN_GAIN

Description:
This parameter, specified in dB, controls a fixed gain that is placed in the transmit path at the microphone input. The placement of the gain block can be seen in the PCM Front End block diagram in section 2 of this document. Loss can be specified using negative gain values.

Parameter Type: static

Range of Values: -12,.12

Default Value: 0

3.3.1.13 VEPARAM_PCM_DELAY_SAMPLES

Description:

This parameter, expressed in units of samples, must reflect the round-trip delay between the AEC's receive output and its transmit input. This delay must include any buffering delay due to the application as well as that due to the audio drivers – both speaker and microphone. This delay must not include acoustic delay between speaker and microphone.

Providing an accurate measure of this delay is critical to the proper operation of the echo canceller. In order to measure the delay, the engine provides a test mode VE_TM_NOISE_SOUNDER. This is described in greater detail later in this document.

Parameter Type: static

Range of Values: 1..32767

Default Value:There is no default value

3.3.1.14 VEPARAM_USE_PCM_FRONT_END_ONLY

Description:

This Boolean parameter, when set, disables the VoIP processing, leaving only the PCM Front End running. When operating in this fashion, VE_ADT_pcmFrontEnd is used to perform processing rather than VE_ADT_pcmToEngine.

Parameter Type: static

Range of Values: 0, 1

Default Value: 0

3.3.1.15 VEPARAM_HOST_SAMPLING_RATE_HZ

Description:

This parameter, specified in Hz, is the sampling rate of the PCM data that is passed to between the host application and the engine. Normally this would be the same as the engine sampling rate. But there may be cases where the framework (Android, in particular) doesn't support a particular sampling rate. In this case, the host sampling rate could be set to 44.1 kHz, which is Android's native sampling rate.

Parameter Type: static

Range of Values: 8000, 16,000, 44,100

Default Value: 8000

3.3.1.16 VEPARAM_ENGINE_SAMPLING_RATE_HZ

Description:

This parameter, specified in Hz, reflects the vocoder sampling rate, which is what drives the RTP time stamp sampling rate.

Parameter Type: static

Range of Values: 8000, 16000

Default Value:8000

3.3.1.17 VEPARAM_SRTP_ENABLE

Description:

This parameter enables or disables SRTP.

Parameter Type: static

Range of Values: 1 enables SRTP, 0 disables SRTP

Default Value:0 (disable)

3.3.1.18 VEPARAM_RTP_LOG_ENABLE

Description:

This parameter is used to enable or disable RTP logging

Parameter Type: static

Range of Values: 1 enables RTP logging, 0 disables RTP logging

Default Value: 0 (disable)

3.3.1.19 VEPARAM_ANDROID_OVERFLOW_FIX_ENABLE

Description:

It has been found that in some Android implementations, there can be an overflow in microphone input PCM. This parameter, when enabled, causes the engine to conceal overflows as best it can.

Parameter Type: static

Range of Values: 1 enables overflow concealment, 0 disables overflow concealment

Default Value: 0 (disable)

3.3.1.20 VEPARAM_TXAGC_ENABLE (Deprecated)

Description:

This parameter allows the transmit AGC to be enabled or disabled.

Parameter Type: static

Range of Values: 1 enables Tx AGC, 0 disables Tx AGC

Default Value: 0 (disable)

3.3.1.21 VEPARAM_LOG_FILTER

Description:

The parameters allows the user to set the amount of log statements shown.

Parameter Type: static

Range of Values: 0x1 to 0xff

Debug: 0x1
Info: 0x2
Warning: 0x4
Error: 0x8
Init Debug: 0x10
Init Info: 0x20
Init Warning: 0x40
Init Error: 0x80

Default Value: 0 (Logging disabled)

3.3.1.22 VEPARAM_DTX_ENABLE

Description:

This parameter enables or disables DTX (Discontinuous Transmission) for the vocoders in the engine instance that support this feature. It is currently supported by

- G.722.2
- AMR

Parameter Type: static

Range of Values: 0, 1

Disabled: 0x0
Enabled: 0x1

3.3.2 AEC Parameters

As we transition from our older AEC technology (AEC-G2) to our newer HD AEC for Mobile, we still maintain two sets of AEC parameters in this document. You should use the appropriate AEC parameters based upon which VoIP Engine library you are using.

3.3.2.1 VE_PARAM_AEC_ENABLE

Description:

This parameter allows the host to enable and disable the echo canceller

Parameter Type: static

Range of Values: 0..1

3.3.2.2 HD AEC – Mobile Parameters

The AEC parameters control the engine's handset and handsfree acoustic echo canceller. The engine maintains a separate set of parameters for handset and handsfree. When setting the handset or handsfree parameters using VE_ADT_setParam, the two modes are distinguished by the module ID (VEMODULE_HANDSET_AEC for handset and VEMODULE_HANDSFREE_AEC for handsfree.)

3.3.2.2.1 VEPARAM_AECHD_MAX_AUDIO_FREQ_HZ

This parameter, specified in Hz, is the highest frequency component in the audio signal. In most applications, a bandpass filter is used to prevent aliasing due to the sampling process. The theoretical maximum audio frequency is equal to half the sampling rate, but with a bandpass filter in place, the maximum audio frequency will be somewhat less. By setting maxAudioFrequency less than half the sampling frequency, CPU utilization (MIPS) can be reduced.

3.3.2.2.2 VEPARAM_AECHD_INITIAL_BULK_DELAY_MSEC

This parameter sets the initial bulk delay. It is specified in milliseconds. This parameter must be greater than or equal to fixedBulkDelayMSec and less than fixedBulkDelayMSec + variableBulkDelayMSec – ActiveTailLengthMSec. Alternately, it can be set to zero to select the default.

This parameter allows the application to take a guess at what the variable bulk delay might be prior to the automatic tail search's measurement of the actual bulk delay.

If you don't have a guess, don't sweat it. Just set this parameter to zero.

3.3.2.2.3 VEPARAM_AECHD_TAIL_SEARCH_WINDOW_MSEC

This parameter, specified in milliseconds, when set to a value greater than zero, enables the echo tail search feature of the AEC. In some software environments, primarily mobile and PC and especially Android, bulk delay is not known or predictable. This parameter allows the user to specify the maximum possible bulk delay (in samples). The AEC will find the location of the actual echo tail and automatically compensate for it, thereby allowing the AEC's active tail region to operate on a region that has actual echo.

3.3.2.2.4 VEPARAM_AECHD_ACTIVE_TAIL_LENGTH_MSEC and VEPARAM_AECHD_TOTAL_TAIL_LENGTH_MSEC

The VEPARAM_AECHD_ACTIVE_TAIL_LENGTH_MSEC parameter defines the tail length, in milliseconds, that the AEC cancels using the adaptive filter. The active tail length should be set according to the expected room acoustics. The acoustic echo attenuation in an acoustic environment tends to increase as the delay increases. When the attenuation reaches a reasonable level, cancellation need no longer be applied. Suppression (nonlinear processing) can be applied instead. The activeTailLength should therefore be set at the delay that corresponds to a reasonable acoustic attenuation level – perhaps 30 dB.

The transmit nonlinear processor (TxNLP) can operate at delays beyond the active tail length in order to suppress residual echo. The VEPARAM_AECHD_TOTAL_LENGTH_MSEC defines the total coverage of the TxNLP.

Note that these parameters must be a multiple of 8 msec.

3.3.2.2.5 VEPARAM_AECHD_MAX_TX_LOSS_ST_DB and VEPARAM_AECHD_MAX_TX_LOSS_DT_DB

The transmit nonlinear processor (TxNLP) suppresses residual echo that is not completely cancelled by the adaptive filter. The TxNLP suppresses primarily during periods of single talk in order to maintain a full-duplex sound to the voice conversation.

These parameters limit the maximum attenuation (loss) that is applied by the transmit NLP when in single-talk (ST) and double-talk (DT) respectively.

3.3.2.2.6 VEPARAM_AECHD_INITIAL_RXOUT_ATTEN_DB

Under circumstances where the speaker to microphone gain is excessive, it is sometimes necessary to attenuate the receive out (speaker out) signal. This parameter sets the initial value of that attenuation amount to improve the performance of the echo canceller at the beginning of a call – before it is able to better determine how to set the receive out loss.

3.3.2.2.7 VEPARAM_AECHD_MAX_RX_LOSS_DB

The receive nonlinear processor (RxNLP) attenuates the receive signal to improve echo attenuation during double-talk periods. This parameter defines the maximum attenuation, expressed in dB, that the RxNLP will apply to the receive signal. Since this RxNLP engages during double-talk, an overly aggressive maxRxLossdB setting will cause a half-duplex sounding conversation.

This loss is different from the RXOUT loss in that this loss is applied at a different part of the receive signal path in the AEC.

3.3.2.2.8 VEPARAM_AECHD_NOISE_REDUCTION_SETTING

This controls if and how noise reduction is configured.

0: Disabled

1: Low Complexity Noise Reduction Enabled
2-31: High Complexity Noise Reduction Enabled. A setting of 2 provides little noise reduction. A setting of 31 provides maximum noise reduction.

It is STRONGLY recommended that noise reduction be enabled.

3.3.2.2.9 VEPARAM_AECHD_MIPS_MEM_REDUCTION_SETTING

This parameter can reduce the MIPS and memory utilization of the AEC. The value ranges between 0 and 4. A value of 0 results in no MIPS or memory reduction, and a value of 4 results in the maximum amount of MIPS and Memory reduction.

With the reduction in MIPS and memory comes a change in performance. In particular, the initial convergence and subsequent reconvergences will be slowed down.

3.3.2.2.10 VEPARAM_AECHD_MIPS_REDUCTION_SETTING_2

This parameter can reduce MIPS that are utilized in the background. Values range from 0 to 4 with 0 being no reduction and 4 being the maximum reduction. A higher setting can causes slower initial convergence and subsequent reconvergence.

Since this setting affects only background processing, the resulting MIPS reduction does not ease real-time constraints. But the reduction can improve power utilization.

3.3.2.2.11 VEPARAM_AECHD_RX_BYPASS_ENABLE

This parameter, when set to 1, will cause all receive signal processing to be bypassed. This includes RxNLP, RxAGC and RxOut attenutation. This is used primarily for testing. Otherwise, it is normally set to zero.

3.3.2.2.12 VEPARAM_AECHD_WORST_EXPECTED_ERL_DB

This setting reflects the worst case expected echo return loss, or the loss between the speaker and microphone as perceived by the AEC. If there is a gain, this parameter will be negative. This parameter is worst-case in two ways. First, it is the worst expected loss for any acoustic conditions at the current volume setting. Second it is the worst loss across the audio frequency band. So if there is a resonance or peak in the frequency response, this loss reflects the loss at that peak.

Note that the worst loss is least amount of loss or the greatest amount of gain.

3.3.2.2.13 VEPARAM_AECHD_TX_NLP_AGGRESSIVENESS

The AEC’s TxNLP can be made more or less aggressive by using the this parameter. The nominal setting is zero. The more positive the setting, the more aggressive the NLP will be. The more negative the setting, the less aggressive the NLP will be. The range is -40..40.

3.3.2.2.14 VEPARAM_AECHD_CNG_ENABLE

This parameter, when set to 1, enables the AEC's comfort noise generator. This is typically turned on.

3.3.2.2.15 VEPARAM_AECHD_TXAGC_ENABLE

If set to 1, the AEC's transmit AGC is enabled. If set to 0, it is disabled.

3.3.2.2.16 VEPARAM_AECHD_TXAGC_TARGETPOWER_DBM

Description:

These parameters defines the AGC target output signal level.

Parameter Type: static

Range of Values: 0..-30 dBm

Default Value: -10 dBm

3.3.2.2.17 VEPARAM_AECHD_TXAGC_MAX_LOSS_DBand VEPARAM_AECHD_TXAGC_MAX_GAIN_DB

Description:

These parameters specify the maximum gain and loss that will be applied.

Parameter Type: static

Range of Values: 0..23 dB

Default Value: Gain: 0, Loss: 0

3.3.2.2.18 VEPARAM_AECHD_TXAGC_LOW_SIG_THRESHOLD_DBM

Description:

This parameter defines the input signal level below which gain will not be applied. This reduces the chance that background noise will be amplified.

Parameter Type: static

Range of Values: - 20 .. -65 dBm

Default Value: -40 dBm

3.3.2.3 AEC G2 Parameters (Deprecated)

The AEC parameters control the engine's handset and handsfree acoustic echo canceller. The engine maintains a separate set of parameters for handset and handsfree. When setting the handset or handsfree parameters using VE_ADT_setParam, the two modes are distinguished by the module ID (VEMODULE_HANDSET_AEC for handset and VEMODULE_HANDSFREE_AEC for handsfree.)

3.3.2.3.1 VEPARAM_TAP_LENGTH (Deprecated)

Description:

This parameter, specified in units of samples, defines the duration of the echo tail. This duration is intended to cover the acoustic echo delay between speaker and microphone. This duration does not include the buffering delay, which will be compensated for by the VEPARAM_PCM_DELAY_SAMPLES.

Parameter Type: static

Range of Values: 64..192, in increments of 16

Default Value: 192

3.3.2.3.2 VEPARAM_SUPP_TAP_LENGTH (Deprecated)

Description:

This parameter, specified in units of samples, defines a timing window beyond VEPARAM_TAP_LENGTH in which there may be a weak echo component due to sound reflecting multiple times off of walls and objects in the room.

Parameter Type: static

Range of Values: 0..1024, in increments of 16

Default Value: Handset: 0, Handsfree: 1024

3.3.2.3.3 AEC Nonlinear Processor (NLP) Parameters (Deprecated)

An echo canceller removes echo in two ways. First, it tries to model the echo characteristics in order to estimate the echo based upon the receive signal. It does this by comparing the speaker output to the microphone input and generating a model. With that model, it takes subsequent speaker output, runs it through the model, and estimates the echo. The echo estimate is subtracted from the microphone input, hopefully cancelling the echo.

That sounds great on paper, but the degree of cancellation that can be achieved is only as good as the derived echo model. And the accuracy of the model is degraded by background noise, mathematical approximations in the echo canceller algorithm, nonlinearity in the acoustic echo path, etc.

As a result of these degradations, there will be some residual echo that bleeds through the canceller. A nonlinear processor (NLP) is employed to remove the remaining residual echo. When there is only receive speech (network to speaker) and no transmit speech (from the microphone toward the network, the nonlinear processor can attenuate the residual echo with no effect on speech quality. If there is, however, somebody speaking at the microphone input side, it is undesirable to attenuate because not only will the residual echo be attenuated, so will be the person's speech that we want to allow to pass from microphone to network. And it is less important that the residual echo be completely removed when both parties are talking because the residual echo will be difficult to hear when somebody is speaking over it.

The nonlinear processor must decide when to engage (attenuate) and when not to attenuate. To that end, the echo canceller maintains a statistic that is the ratio of the speaker output power to the residual echo power. In that ratio is high, it is deemed OK to engage the NLP. Otherwise, the NLP should not be engaged.

A number of thresholds are provided to the host software to customize the NLP. These thresholds are compared to the aforementioned power ratio. The particular threshold that is used is a function of the state of the echo canceller and the state of the NLP. These thresholds are discussed in further detail in the sections that follow.

Beyond just the thresholds, the NLP can be configured to act in ways other than simple attenuation. Attenuation affects background noise in the micropohone to network (transmit) direction. When the NLP engages, the noise can virtually disappear, but when the NLP disengages, the noise returns. This can be annoying to the person at the other end of the call. So instead of simply attenuating, the NLP can replace the residual echo with "comfort noise."

3.3.2.3.3.1 VEPARAM_NLPTYPE (Deprecated)

Description:

This parameter controls what type of action the NLP takes when it engages.

Parameter Type: static

Range of Values:

- **AEC_NLP_OFF:** disables the NLP. This is used primarily for testing
- **AEC_NLP_MUTE:** when the NLP is activated, it completely mutes (zeros out) the residual signal
- **AEC_NLP_HOTH_CNG:** when the NLP is activated, the residual is replaced with Hoth noise at a level that matches the actual level of the background noise
- **AEC_NLP_SUPP_AUTO:** when the NLP is activated, it attempts to suppress the residual down to a target level, and if necessary it adds comfort noise.
- **AEC_NLP_HOTH_FIXED:** when the NLP is activated, it replaces the residual echo with low fixed-level Hoth noise

Default Value: Handset: AEC_NLP_HOTH_CNG, Handsfree: AEC_NLP_MUTE

3.3.2.3.3.2 VEPARAM_NLP_THRESHOLD (Deprecated)

Description:

This NLP Threshold, expressed in dB, is used when the echo canceller is converged and the NLP is currently in a non-engaged state.

Parameter Type: static

Range of Values: 0..36

Default Value: Handset: 24, Handsfree: 18

3.3.2.3.3.3 VEPARAM_NLP_THRESHOLD_UPPER_LIMIT_CONVERGED (Deprecated)

Description:

This NLP Threshold, expressed in dB, is used when the echo canceller is converged and the NLP is currently already engaged. It is used to introduce hysteresis to prevent rapid cycling between NLP Engaged and NLP Not Engaged state. This setting should therefore be be lower than VEPARAM_NLP_THRESHOLD.

Parameter Type: static

Range of Values: 0..36

Default Value: Handset: 18, Handsfree: 12

3.3.2.3.3.4 VEPARAM_NLP_THRESHOLD_UPPER_LIMIT_UNCONVERGED (Deprecated)

Description:
This NLP Threshold, expressed in dB, is used when the echo canceller is not yet converged. It is intended to let the NLP be aggressive until the canceller **has a chance to converge. This number should therefore be less than** VEPARAM_NLP_THRESHOLD_UPPER_LIMIT_UNCONVERGED.

Parameter Type: static

Range of Values: 0..36

Default Value: Handset: 12, Handsfree: 0

3.3.2.3.3.5 VEPARAM_DYNAMIC_NLP_AGGRESSIVENESS (Deprecated)

Description:

Using this parameter, the host can let the AEC dynamically determine the NLP threshold under certain situations. This parameter, expressed as a number, informs the NLP how aggressive it can be. A higher number will cause the NLP to engage more often / under more conditions. A value of -1 will disable the dynamic nature of the NLP.

Parameter Type: static

Range of Values: -1..20

Default Value: Handset: -1 (disabled), Handsfree: 20

3.3.3 RTP Parameters

3.3.3.1 VEPARAM_RTP_SSRC

Description:

Unique session member ID. Zero indicates randomly generated by RTP

Parameter Type: static

Range of Values: 0..65535

Default Value: 0

3.3.3.2 VEPARAM_RTP_DELAY_TARGET_MIN

Description:

This parameter is the minimum jitter buffer size for dynamic jitter buffer or threshold of jitter buffer underflow for static jitter buffer. It is expressed in RTP packet timestamp unit. This parameter must be less than VEPARAM_RTP_DELAY_TARGET.

Parameter Type: static

Range of Values: 80..

Default Value: 480

3.3.3.3 VEPARAM_RTP_DELAY_TARGET

Description:

This parameter is used as the initial jitter buffer delay size. It is expressed in RTP packet timestamp units. This parameter must be greater than VEPRAM_RTP_DELAY_TARGET_MIN and less than VEPARAM_RTP_DELAY_TARGET_MAX.

Parameter Type: static

Range of Values: 80..

Default Value: 800

3.3.3.4 VEPARAM_RTP_DELAY_TARGET_MAX

Description:

This parameter is the maximum jitter buffer size for dynamic jitter buffer or threshold of jitter buffer overflow for static jitter buffer. It is expressed in RTP packet timestamp units. This parameter must be greater than VEPARAM_RTP_DELAY_TARGET.

Parameter Type: static

Range of Values: 80..

Default Value: 1600

3.3.3.5 VEPARAM_RTP_JBMAX

Description:

This parameter sets the size of the jitter buffer. It is expressed in RTP packet timestamp units. This is the amount of memory that will be allocated for the jitter buffer.

Parameter Type: static

Range of Values: 80..

Default Value: 4000

3.3.3.6 VEPARAM_RTP_JITTER_MODE

Description:
This parameter controls whether the jitter buffer is static or dynamic. A static jitter buffer (VEPARAM_RTP_JITTER_MODE = 0) has a fixed delay that is equal to VEPARAM_RTP_DELAY_TARGET. A dynamic jitter buffer's delay can increase or decrease between VEPARAM_RTP_DELAY_TARGET_MIN and VEPARAM_RTP_DELAY_TARGET_MAX. The delay depends upon measured packet jitter. The dynamic jitter buffer can be programmed to operate in different manners by setting VEPARAM_RTP_JITTER_MODE anywhere between the values of 1 and 12. Set at a high value, the dynamic jitter buffer aggressively tracks changes in jitter delay and the size of the buffer increase quickly and shrink slowly. At a low value, the size of the buffer increases slowly and shrinks reasonably fast.

Parameter Type: static

Range of Values: 0: Static, 1-12 Dynamic

Default Value: 1

3.3.4 Transmit AGC Parameters (Deprecated – moved to HD AEC parameters)

3.3.4.1 VEPARAM_TXAGC_TARGETPOWER_LOW_DBM and VEPARAM_TXAGC_TARGETPOWER_HIGH_DBM

Description:

These parameters define the AGC's target output signal level range. The AGC applies gain or loss (within the specified maximum range of gain and loss) in an attempt to drive the output signal level so that it lies between the low and high target levels.

Parameter Type: static

Range of Values: 0..-30 dBm

Default Value: Low = -20 dBm, High = -7 dBm

3.3.4.2 VEPARAM_TXAGC_MAX_GAIN_DB and VEPARAM_TXAGC_MAX_LOSS_DB

Description:

These parameters specify the maximum gain and loss that will be applied.

Parameter Type: static

Range of Values: 0..23 dB

Default Value: Gain: 10, Loss: 10

3.3.4.3 VEPARAM_TXAGC_LOW_SIG_THRESHOLD_DBM

Description:

This parameter defines the input signal level below which gain will not be applied. This reduces the chance that background noise will be amplified.

Parameter Type: static

Range of Values: - 20 .. -65 dBm

Default Value: -40 dBm

3.4 VoIP Engine API

3.4.1 Data Structures and Types Used in API Functions

3.4.1.1 Engine Number (int)

Since multiple VoIP Engine instances can be running at the same time, it is important that the user defines an 'EngineNumber' variable of type int. This should be sequentially incremented for each new instances of the engine, starting at zero.

3.4.1.2 Phone Profile Structure (PhoneProfile_t) (Deprecated)

The user must pass Phone Profile to the Voice Engine to provide both handset and handsfree gain characteristics. The following table describes the entries in the phone profile structure.

Parameter Name	Description
NominalERLdB	The speaker to microphone acoustic coupling loss when the speaker volume is set to its nominal (midpoint) value
NSpeakerGainClicks	The number of possible discrete speaker gain settings. Must be less than MAX_SPEAKER_GAIN_CLICKS
SpeakerGainAtClick	An array of relative gain values of length NSpeakerGainClicks. The gain is relative to the midpoint gain.

Table 3-1 - Phone profile parameters

3.4.1.3 Status Structure VE_ADT_Status_t

The status structure returns a variety of information about the state of the VoIP Engine and it's algorithms. The following table describes the data contained in the status structure.

Parameter Name	Description
ERL1	Echo Return Loss (in units of dB) reported by the AEC, computed using AEC coefficients
ERL2	Echo Return Loss (in units of dB) reported by the AEC, computed using measured signal levels
ERLE	Echo Return Loss Enhancement (in units of dB) reported by the AEC. ERLE is a measure of how much cancellation is being done by the AEC before nonlinear processing. The ERLE measure is most accurate during single-talk. It is less accurate during double-talk or prolonged periods of silence.
Echo Delay	Measured echo delay, not including the bulk delay (PCMDelaySamples). This statistic is not reported (and not needed) in iOS builds.
MicInPowerdBm	Microphone input power, measured in dBm
PeakMicInPowerdBm	Peak Microphone input power, measured in dBm
SpeakerOutPowerdBm	Speaker output power, measured in dBm
PeakSpeakerOutPowerdBm	Peak speaker output power, measured in dBm
Noise Sounder ERL	ERL (Echo Return Loss), measured by the Noise Sounder – in units of dB
NoiseSounderDelay	Round trip delay between the VoIP Engine PCM output (to speaker) signal and VoIP Engine input (from microphone), measured in samples.
NoiseSounderResultCount	This count is incremented once per NoiseSounder trial to indicate that a new set of NoiseSounderERL and NoiseSounderDelay are available.
JitterEstimate	An estimate of the network jitter as measured by RTP
JitterBufferDelay	The current delay through the jitter buffer.

Table 3-2 – Status Structure

3.4.1.4 Start Parameters VE_ADT_startParams_t

Parameter Name	Description
PCMDelaySamples	This parameter, expressed in units of samples, must reflect the round-trip delay between the AEC's receive output and its transmit input. This delay must include any buffering delay due to the application as well as that due to the audio drivers – both speaker and microphone. This delay must not include acoustic delay between speaker and microphone.

3.4.1.5 Event Structure VE_ADT_Event_t

Future

3.4.1.6 VE_ADT_Handle_t

In order to keep track of instance data, the VE_ADT_Handle_t data type is defined. This handle points to memory block that is allocated to a particular instance. It is the responsibility of the host software to set the value of the handle to point to the allocated instance data. This handle is passed to all APIs except for VE_ADT_getMemSize. For single channel static APIs, Handle should be set to null.

3.4.1.7 PCM_t

PCM_t is a generic type that defines the host's PCM sample type. PCM samples can be either 16-bits wide or 32 bits wide. The width is currently hard-coded in the engine at 16-bits for Android and other platforms.

3.4.2 API Functions

API Functions are divided into the following categories:

- Setup Functions
- Frame and Packet Processing Functions
- Event Functions
- Callback / User Supplied Functions
- Parameter Handling Functions
- Codec Plug-in Functions

3.4.2.1 Setup Functions

3.4.2.1.1 VE_ADT_create

VE_ADT_create creates a new instance of VoIP Engine.

Prototype:

```
VE_ADT_Handle_t VE_ADT_create(
      createResult_t *pResultCode,
      VE_Device_e Device
      );
```

Inputs:

pResultCode – pointer to location where result will be returned

Device – the type of device that is in use. Default parameters are set up based upon the device selection. Device types are listed below:

```
VEDEV_UNKNOWN
VEDEV_IPHONE3
VEDEV_IPHONE3GS
VEDEV_IPHONE4
VEDEV_IPODTOUCH4
VEDEV_IPAD
VEDEV_IPAD2
VEDEV_ANDROID
```

Outputs:

*pResultCode – Result code from create.

Returns:

Handle to VoIP Engine instance

Result Codes::

```
VE_IOBJ_RESULT_OK
VE_IOBJ_RESULT_HANDLE_ERROR
VE_IOBJ_RESULT_NOT_INITIALIZED
```

3.4.2.1.2 VE_ADT_setDefaults

Prototype:

```
VE_GenericResult_t VE_ADT_setDefaults(
      VE_ADT_Handle_t Handle,
      VE_Device_e Device
      );
```

Inputs:

Handle is the handle to the VoIP Engine

Device – the type of device that is in use. Default parameters are set up based upon the device selection. Device types are listed below:

```
VEDEV_UNKNOWN
VEDEV_IPHONE3
VEDEV_IPHONE3GS
VEDEV_IPHONE4
VEDEV_IPODTOUCH4
VEDEV_IPAD
VEDEV_IPAD2
VEDEV_ANDROID
```

Outputs:

Returns:

VE_GenericResult_t (see voipengine_user_v4.h)

3.4.2.1.3 VE_ADT_setParamValue

VE_ADT_setParamValue is used to set individual parameters. The parameters and modules are defined and explained in a previous section in this document.

Prototype:

```
int  VE_ADT_setParamValue(
      VE_ADT_Handle_t Handle,
      ADT_UInt8 ModuleID,
      ADT_UInt8 ParamID,
      ADT_Int32 Value);
```

Inputs:

Handle is the handle to the VoIP Engine
ModuleID - module ID associated with the parameter to be set
ParamID – parameter ID
Value – the value used to set the selected parameter

Outputs:

Returns:

VEPARAM_ERROR
VEPARAM_OK

3.4.2.1.4 VE_ADT_configureSRTP

This function is not supported at this time, but may be supported in the future. SRTP is configured with defaults as follows:

Key Scheme: Pre-shared master
Encryption Type: Counter Mode
Authentication Type: HMAC SHA
Key Size: 16
Salt Size: 14
Authentication Key Size: 20
Authentication Tag Length: 10

3.4.2.1.5 VE_ADT_start

VE_ADT_start initializes the VoIP Engine and places it in the start state. All static parameter initialization must be completed before calling VE_ADT_start.

Prototype:

```
VE_startResult_t VE_ADT_start(
      VE_ADT_Handle_t Handle,
      VE_ADT_startParams_t *pParams,
      VE_ADT_Callbacks_t *pCallbacks);
```

Inputs:

Handle is the handle to the VoIP Engine

pParams – parameters that must be specified at start time. While most parameters can be left at their default values, these parameters must be set explicitly. The start parameters are listed here.

pCallbacks – pointer to a structure that contains pointers to the RTP callback functions. See VE_ADT_Callbacks_t in voipengine_user_v4.h for the format of the structure. Refer to the RTP callback API and SRTP callback API and Test Point callback API later in this document for a description of the callback RTP callback functions that need to be supplied by the host software.

Outputs:

Returns:

VE_startResult_t (see voipengine_user_v4.h)

3.4.2.1.6 VE_ADT_stop

VE_ADT_stop stops the engine. This function can be called at the end of a call prior to deleting the engine, or it can be called to pause the engine in order to change static parameters.

Prototype:

```
      VE_stopResult_t VE_ADT_stop(VE_ADT_Handle_t Handle);
```

Inputs:
Handle is the handle to the VoIP Engine

Outputs:

Returns:
VE_stopResult_t (see voipengine_user_v4.h)

3.4.2.1.7 VE_ADT_delete

VE_ADT_delete deletes a VoIP Engine instance.

Prototype:

```
      VE_GenericResult_t VE_ADT_delete(VE_ADT_Handle_t Handle);
```

Inputs:
Handle is the handle to the VoIP Engine

Outputs:

Returns:
VE_GenericResult_t (see voipengine_user_v4.h)

3.4.2.1.8 VE_ADT_addPacketChannel

Adds a packet channel for use in a conference call.Note that there is a hard-coded limit of 4 packet channels. If a higher limit is needed, please contact Adaptive Digital.

Prototype:

```
VE_GenericResult_t VE_ADT_addPacketChannel(
      VE_ADT_Handle_t Handle,
      VE_Codec_e TxCodec,
      ADT_UInt8 RxOnly,
      ADT_UInt8 BypassConference,
      ADT_UInt8 *pChannelID
      );
```

Inputs:

Handle is the handle to the VoIP Engine
TxCodec – specifies the transmit codec type
RxOnly – if 1, disable the transmit path, 0 otherwise
BypassConference – if 1, bypass the conference mixer. 0 otherwise

Outputs:

*pChannelID – contains the ID (index) of the packet channel. This ID is used by other API functions such as VE_ADT_removePacketChannel( ), VE_ADT_rtpToEngine( ), and VE_ADT_onSendRTPToNetwork( ).

Returns:

VE_GenericResult_t (see voipengine_user_v4.h)

3.4.2.1.9 VE_ADT_removePacketChannel

Removes a packet channel from a conference call.

Prototype:

```
VE_GenericResult_t VE_ADT_removePacketChannel(
      VE_ADT_Handle_t Handle,ADT_UInt8 ChannelID);
```

Inputs:

Handle is the handle to the VoIP Engine
ChannelID – the ID (index) of the packet channel to be removed from the engine.

Outputs:

Returns:

VE_GenericResult_t (see voipengine_user_v4.h)

3.4.2.1.10 VE_ADT_setCapturePoints

VE_ADT_setCapturePoints is provide four buffers to the engine for the purpose of capturing the AEC input and output buffers. This can be very helpful in diagnosing voice quality problems. The engine will save EngineFrameSizeSamples (16-bit PCM) into each of the four buffers. The host must therefore allocate these buffers accordingly. By default, capture is disabled.

Prototype:

```
VE_GenericResult_t VE_ADT_setCapturePoints(
      VE_ADT_Handle_t Handle,
      PCM_t *pAECRxIn,
      PCM_t *pAECRxOut,
      PCM_t *pAECTxIn,
      PCM_t *pAECTxOut);
```

Inputs:

Handle is the handle to the VoIP Engine
pAECRxIn – host buffer where AEC Rx Input will be saved
pAECRxOut – host buffer where AEC Rx Output will be saved
pAECTxIn – host buffer where AEC Tx Input buffer will be saved
pAECTxOut – host buffer where AEC Tx Output buffer will be saved

Outputs:

Captured buffers will be saved in pAECRxIn, pAECRxOut, pAECTxIn, and pAECTxOut

Returns:

VE_GenericResult_t (see voipengine_user_v4.h)

3.4.2.1.11 VE_ADT_setEQFilter

VE_ADT_setEQFilter sets one of the equalizer filter characteristics. There are four equalizers – a transmit and receive equalizer for handset mode and a transmit and receive equalizer for handsfree mode. The transmit equalizer is intended to equalize the microphone frequency response. The receive equalizer is intended to equalize the speaker frequency response.

Prototype:

```
VE_GenericResult_t VE_ADT_setEQFilter(
      VE_ADT_Handle_t Handle,
      ADT_UInt8 Enable,
      const ADT_Int16 pCoef[],
      ADT_Int16 Length,
      VE_ADT_Route_e Route,
      ADT_UInt8 TxFlag);
```

Inputs:

Handle is the handle to the VoIP Engine
Enable – 1 to enable the selected equalizer, 0 to disable it

pCoef – pointer to the filter coefficients, specified in 16-bit Q15 format

Length – number of filter coefficients. THE MAXIMUM SIZE IS 128.

Route – VEROUTE_HANDSET or VEROUTE_SPEAKER

TxFlag – 1 to select transmit (microphone) equalizer, 0 to select receive (speaker) equalizer

Outputs:

Returns:

VE_GenericResult_t (see voipengine_user_v4.h)

3.4.2.1.12 VE_ADT_setPhoneProfile (Deprecated)

Note that phone profiles are no longer needed with the introduction of AEC-HD.

VE_ADT_setPhoneProfile is used to set the phone's handset or handsfree acoustic characteristics. In particular, it is necessary to let the engine know what the speaker to microphone loss is at each user-controlled volume setting. It is also necessary to let the engine know how many volume settings there $A = \pi r^2$.

The iOS devices have default phone profiles built-into the VoIP Engine. Since there are so many different models of Android phones, the profile is left to the host. The same holds for other platforms like the PC.

Prototype:

```
VE_GenericResult_t VE_ADT_setPhoneProfile(
        VE_ADT_Handle_t Handle,
        const PhoneProfile_t *pProfile,
        VE_ADT_Route_e Route);
```

Inputs:

Handle is the handle to the VoIP Engine
pProfile – See definition of PhoneProfile_t in an earlier section of this document
Route – VEROUTE_HANDSET or VEROUTE_SPEAKER

Outputs:

Returns:

VE_GenericResult_t (see voipengine_user_v4.h)

3.4.2.1.13 VE_ADT_enableStatistics

VE_ADT_enableStatistics is used to tell the engine to compute various statistics.

Prototype:

```
VE_GenericResult_t VE_ADT_enableStatistics(
      VE_ADT_Handle_t _Handle,
      ADT_Int8 AECEnable,
      ADT_Int8 MicInEnable,
      ADT_Int8 SpeakerOutEnable);
```

Inputs:

Handle is the handle to the VoIP Engine
AECEnable – 1: enable computation of AEC statistics, 0: disable
MicInEnable – 1: enable computation of microphone in put level, 0: disable
SpeakerOutEnable – 1: enable computation of speaker output level, 0: disable

Outputs:

Returns:

VE_GenericResult_t (see voipengine_user_v4.h)

3.4.2.1.14 VE_ADT_enableTestPoint and VE_ADT_disableTestPoint

These functions enable and disable test points. When enabled, the engine calls the user-supplied Test Point callback function to write and/or read data from/to the test point within the engine data flow. If the test point is part of the packet processing, a channel ID is needed to specify which packet channel test point is being enabled/disabled.

Prototypes:

```
VE_GenericResult_t VE_ADT_enableTestPoint(
      VE_ADT_Handle_t Handle,
      ADT_UInt8 ChannelID,
      TestPoint_e TP);

VE_GenericResult_t VE_ADT_disableTestPoint(
      VE_ADT_Handle_t Handle,
      ADT_UInt8 ChannelID,
      TestPoint_e TP);
```

Inputs:

Handle – handle to VoIP Engine instance
ChannelID – specifies packet channel ID (if test point is within the packet processing)
TP – indicates which test point is being enabled or disabled

Outputs:

Returns:

Always returns VE_RESULT_OK

3.4.2.1.15 VE_ADT_setEncodeMode

This function is used to set the encode mode and output format used by the G.722.2 and AMR codec.

Prototypes:

```
ADT_API VE_GenericResult_t VE_ADT_setEncodeMode(
      VE_ADT_Handle_t Handle,
      ADT_UInt8 ChannelID,
      ADT_UInt16 Mode,
      ADT_UInt16 OutputFormat)
```

Inputs:

Handle – handle to VoIP Engine instance

ChannelID – ID of the G.722.2 or AMR channel you are setting

Mode – Encoding mode to set the channel to. Valid values depend on the Codec type, and are enumerated in voipengine_user_v4.h. See: VE_AMR_MODE_e and VE_G722_2_MODE_e

OutputFormat – Selects the format of the output from the Encoder. Valid values depend on the Codec type, and are enumerated in voipengine_user_v4.h. See: VE_AMR_OUTPUTFORMAT_e and VE_G722_2_OUTPUTFORMAT_e;

Outputs:

None

Returns:

VE_GenericResult_t (see voipengine_user_v4.h)

3.4.2.1.16 VE_ADT_setCMR

This function sets the Codec Mode Request (CMR) value used by the G.722.2 and AMR encoder.

Prototypes:

```
ADT_API VE_GenericResult_t VE_ADT_setCMR(
      VE_ADT_Handle_t Handle,
      ADT_UInt8 ChannelID,
      ADT_UInt16 CMRValue)
```

Inputs:

Handle – handle to VoIP Engine instance

ChannelID – ID of the G.722.2 or AMR channel you are setting

CMRValue – CMR Value to set the encoder to. Valid values depend on the Codec type, and are enumerated in voipengine_user_v4.h. See: VE_AMR_MODE_e and VE_G722_2_MODE_e

Outputs:
None

Returns:
VE_GenericResult_t (see voipengine_user_v4.h)

3.4.2.1.17 VE_ADT_getCMR

This function gets the most recent received Codec Mode Request value from the G.722.2 or AMR decoder.

Prototypes:

```
ADT_API VE_GenericResult_t VE_ADT_getCMR(
      VE_ADT_Handle_t Handle,
      ADT_UInt8 ChannelID,
      ADT_UInt16 *CMRValue)
```

Inputs:

Handle – handle to VoIP Engine instance

ChannelID – ID of the G.722.2 or AMR channel you are setting

Outputs:
CMRValue – Last CMR Value received by decoder.

Returns:
VE_GenericResult_t (see voipengine_user_v4.h)

3.4.2.2 Frame and Packet Processing Functions

The functions listed in this section deal with PCM and Packet handling on a frame-by-frame or packet by packet basis.

The engine can be used in one of two ways:

- PCM Front End – includes only the PCM handling such as sampling rate conversion, equalization, acoustic echo cancellation, noise reduction, and AGC
- Complete VoIP – includes all the PCM Front End functionality plus speech compression and RTP/Jitter Buffer

You may wish to refer to the block diagrams earlier in this document for a refresher on how these functionalities fit together.

When operating as a PCM Front End only, the only frame processing function that should be called is VE_ADT_pcmFrontEnd.

When operating as a complete VoIP engine, VE_ADT_pcmToEngine and VE_ADT_rtpToEngine should be called, but VE_ADT_pcmFrontEnd should not be called. The PCM front end functionality is included in VE_ADT_pcmToEngine.

3.4.2.2.1 VE_ADT_pcmToEngine

The host software calls VE_ADT_pcmToEngine to pass a frame of PCM data to the engine. VE_ADT_pcmToEngine performs (PCM to Packet) Transmit processing. It also performs Receive (Packet to PCM) processing if packets are available in the RTP receive jitter buffer.

Prototype:

```
VE_pcmToEngineResult_t VE_ADT_pcmToEngine(
      VE_ADT_Handle_t Handle,
      PCM_t *pPCMTxIn, PCM_t *pPCMRxOut);
```

Inputs:

Handle is the handle to the VoIP Engine

pPCMTxIn – pointer to PCM input from microphone. The format may be either 16 bit or 32-bit PCM, depending upon the setting of the setting of PCMWordSizeInBytes. The number of samples in the buffer is defined at initialization time via the HostFrameSizSamples parameter.

Outputs:

pPCMRxOut – pointer to PCM output that is to be sent to the microphone. The format may be either 16 bit or 32-bit PCM, depending upon the setting of the setting of PCMWordSizeInBytes. The number of samples in the buffer is defined at initialization time via the HostFrameSizSamples parameter.

Calls: VE_ADT_cbOnSendRTPToNetwork, which sends the RTP packet to the network.

Returns:

VE_ADT_pcmToEngineResult_t (see voipengine_user_v4.h)

3.4.2.2.2 VE_ADT_rtpToEngine

The host calls VE_ADT_rtpToEngine to pass an RTP packet to the engine.

Prototype:

```
VE_rtpToEngineResult_t VE_ADT_rtpToEngine(
      VE_ADT_Handle_t Handle,
      ADT_UInt8 ChannelID,
      ADT_UInt8 *pRTPPacket,
      ADT_Int32 Size);
```

Inputs:

Handle is the handle to the VoIP Engine

Channel ID is the ID of the packet channel to which the RTP packet is being fed

pRTPPacket – pointer to RTP packet

Size – size in bytes of RTP packet.

Outputs:

Returns:

VE_rtpToEngineResult_t (see voipengine_user_v4.h)

3.4.2.2.3 VE_ADT_pcmFrontEnd

VE_ADT_pcmFrontEnd handles all processing between the speaker / microphone and a user-supplied VoIP handler.

Prototype:

```
VE_pcmToEngineResult_t VE_ADT_pcmFrontEnd(
      VE_ADT_Handle_t Handle,
      PCM_t *pTxIn,
      ADT_Int16 *pTxOut,
      ADT_Int16 *pRxIn,
      PCM_t *pRxOut);
```

Inputs:

Handle is the handle to the VoIP Engine

pTxIn – pointer to transmit input, a buffer of size HostFrameSizeSamples of type PCM_t.

pTxOut – pointer to transmit output, a buffer of size EngineFrameSizeSamples

pRxIn – pointer to receive input, a buffer of size EngineFrameSizeSamples

pRxOut – pointer to receive output, a buffer of size HostFrameSizeSamples of type PCM_t.

Outputs:

Transmit and Receive output are placed in pTxOut and pRxOut respectively

Returns:

VE_pcmToEngineResult_t (see voipengine_user_v4.h)

3.4.2.2.4 VE_ADT_backgroundHandler

VE_ADT_background handler must be called periodically in an idle or low-priority thread. The recommended call frequency is between 10 and 20 milliseconds, but you should note that based upon the background processing and other processing, the background handler may take more than amount of time. If VE_ADT_backgroundHandler is is called in an idle thread, it can be called in a loop continuously. If there is nothing to be done, it will return immediately.

3.4.2.3 Event Functions

Event Functions are those functions that can happen asynchronously during an active call (while the engine is in the start state.)

3.4.2.3.1 VE_ADT_toneGenStart

The host calls VE_ADT_toneGenStart to initiate tone generation in the receive direction (toward the speaker). Tone generation can be continuous, as single pulse, or a repeating cadence. See the Tone Generator Users Guide for details on configuring the tone generator.

Prototype:

```
VE_GenericResult_t VE_ADT_toneGenStart(
      VE_ADT_Handle_t Handle,
      ADT_UInt8 Direction,
      TGParams_t *pTGParams);
```

Inputs:

Handle is the handle to the VoIP Engine

`pTGParams –` pointer to tone generator parameter structure (See Tone Generator Users Guide).

Direction – determines the direction in which the tone is to be sent. 0 for receive, 1 for send.

Outputs:

Returns:

VE_GenericResult_t (see voipengine_user_v4.h)

3.4.2.3.2 VE_ADT_toneGenEnd

The host calls VE_ADT_toneGenEnd to stop the tone generator.

Prototype:

```
VE_GenericResult_t VE_ADT_toneGenEnd(
      VE_ADT_Handle_t Handle,
      ADT_UInt8 Direction);
```

Inputs:

Handle is the handle to the VoIP Engine

Direction – determines the direction in which the tone is to be sent. 0 for receive, 1 for send.

Outputs:

Returns:

VE_GenericResult_t (see voipengine_user_v4.h)

3.4.2.3.3 VE_ADT_analogSpeakerGainIndexIs (Deprecated)

With the introduction of AEC-HD, this function is replaced by VE_ADT_speakerGainChange.

The host calls VE_ADT_analogSpeakerGainIndexIs to inform the engine that the user-controlled speaker volume has changed. This function is usually called when the volume up / down button is pressed on the phone.

Prototype:

```
VE_GenericResult_t VE_ADT_analogSpeakerGainIndexIs(
     VE_ADT_Handle_t Handle,
     ADT_Int8 Index);
```

Inputs:

Handle is the handle to the VoIP Engine

Index – specifies the speaker gain as an index. The relationship between the index and the gain in dB is defined in the phone profile.

Outputs:

Returns:

VE_GenericResult_t (see voipengine_user_v4.h)

3.4.2.3.4 VE_ADT_speakerGainChange

The host calls VE_ADT_speakerGainChange to inform the engine that the user-controlled speaker volume has changed. This function is usually called when the volume up/down button is pressed on the phone.

Prototype:

```
VE_GenericResult_t VE_ADT_speakerGainChange(
     VE_ADT_Handle_t Handle,
     ADT_Int8 GainChangedB);
```

Inputs:

Handle is the handle to the VoIP Engine

GainChangedB – specifies the change in speaker gain, expressed in dB. If the amount is unknown, use a value of 127.

Outputs:

Returns:

VE_GenericResult_t (see voipengine_user_v4.h)

3.4.2.3.5 VE_ADT_setSpeakerMode

The host calls VE_ADT_setHandsetMode to change from hands-free mode to handset mode.

Prototype:

```
      VE_GenericResult_t VE_ADT_setHandsetMode(
            VE_ADT_Handle_t Handle,
            HandsfreeActive);
```

Inputs:

Handle is the handle to the VoIP Engine

HandsfreeActive: 1 for hands-free operation, 0 for handset operation

Outputs:

Returns:

VE_GenericResult_t (see voipengine_user_v4.h)

3.4.2.3.6 VE_ADT_setHostControlledSpeakerGain

This function enables the host to set a fixed gain or loss just before the PCM output toward the speaker. This can be used in case the analog gain / loss is either insufficient or not controllable. This gain is used in conjunction with the RX_OUT_GAIN, Both the host controlled speaker gain and the RX_OUT_GAIN are applied at the same point in the signal flow. The difference is that RX_OUT_GAIN is used for tuning whereas Host Controlled Speaker gain is intended to be used in situations where the host environment isn’t handling volume control correctly. But in the end, the sum of the gains is applied at the same point.

Prototype:

```
      VE_GenericResult_t VE_ADT_setHostControlledSpeakerGain(
            VE_ADT_Handle_t Handle,
            ADT_Int16 GaindB);
```

Inputs:

Handle is the handle to the VoIP Engine

GaindB: Gain, specified in dB. (Loss can be applied by specifying a negative number.)

Outputs:

Returns:

VE_GenericResult_t (see voipengine_user_v4.h)

3.4.2.3.7 VE_ADT_getStatus

Prototype:

```
VE_GenericResult_t VE_ADT_getStatus(
      VE_ADT_Handle_t Handle,
      VE_ADT_Status_t *pStatus);
```

Inputs:

Handle is the handle to the VoIP Engine

pStatus – pointer to a status buffer where the status will be returned.

Outputs:

pStatus Status of the voice engine and all its algoritms are placed in the status structure pointed to by pStatus. See the Status_t table earlier in this document.

Returns:

VE_GenericResult_t (see voipengine_user_v4.h)

3.4.2.3.8 VE_ADT_setGain

This function is available so the host software can set a variety of fixed gains in the transmit and receive path.

Prototype:

```
VE_GenericResult_t VE_ADT_setGain(VE_ADT_Handle_t Handle,

      GainStage_e Stage,
      ADT_Int16 GainIndB);
```

Inputs:

Handle is the handle to the VoIP Engine
Stage – specifies the position in the signal flow where the gain will be applied.
RX_PATH_GAIN_E: in the receive path
RX_OUT_GAIN_E: just before the output to the speaker
TX_PATH_GAIN_E: in the transmit path
TX_IN_GAIN_E: just after the microphone input
GainIndB – specifies the gain in dB. The allowable range is -12..12.

Outputs:

Results:

VE_GenericResult_t (see voipengine_user_v4.h)

3.4.2.3.9 VE_ADT_setRxPathGain

This function is available so the host can set the Receive Packet Channel gain. This parameter cannot be set by VE_ADT_setGain because VE_ADT_setGain has no argument to specify the channel ID.

Prototype:

```
ADT_API VE_GenericResult_t    VE_ADT_setRxPacketGain(

      VE_ADT_Handle_t _Handle,
      ADT_UInt8 ChannelID,
      ADT_Int16 GainIndB);
```

Inputs:

Handle is the handle to the VoIP Engine
ChannelID is the channel ID. This is used only for gain stage RX_CODEC_GAIN_E since that is the only gain stage that is associated with a packet channel.

GainIndB – specifies the gain in dB. The allowable range is -12..12.

Outputs:

Results:

VE_GenericResult_t (see voipengine_user_v4.h)

3.4.2.3.10 VE_ADT_dtmfKeyPress

This function is indicates to the engine that a DTMF key has been pushed down. This initiates the transmission of RTP tone events.

Prototype:

```
VE_GenericResult_t VE_ADT_dtmfKeyPress(
      VE_ADT_Handle_t Handle,
      ADT_UInt8 Digit);
```

Inputs:

Handle is the handle to the VoIP Engine

Digit – the DTMF digit that has been pressed. Digit definitions can be found in voipengine_user.h

Outputs:

Results:

VE_GenericResult_t (see voipengine_user_v4.h)

3.4.2.3.11 VE_ADT_dtmfKeyRelease

This function is indicates to the engine that a DTMF key has been released.This initiates the end of RTP tone event transmission.

Prototype:

```
VE_GenericResult_t VE_ADT_dtmfKeyReleasePress(
      VE_ADT_Handle_t Handle);
```

Inputs:

Handle is the handle to the VoIP Engine

Outputs:

Results:

VE_GenericResult_t (see voipengine_user_v4.h)

3.4.2.3.12 VE_ADT_txMuteEnable

This function causes the engine to enable or disable muting of the transmit signal path.

Prototype:

```
VE_GenericResult_t VE_ADT_txMuteEnable(
VE_ADT_Handle_t Handle,       ADT_Int8 Enable);
```

Inputs:

Handle is the handle to the VoIP Engine

Enable: If set, mute the transmit path, otherwise don't mute.

Outputs:

Returns:

VE_GenericResult_t (see voipengine_user_v4.h)

3.4.2.3.13 VE_ADT_rxMuteEnable

This function causes the engine to enable or disable muting of the receive signal path.

Prototype:

```
VE_GenericResult_t VE_ADT_rxMuteEnable(VE_ADT_Handle_t Handle,
ADT_Int8 Enable);
```

Inputs:

Handle is the handle to the VoIP Engine

Enable: If set, mute the receive, otherwise don't mute.

Outputs:

Returns:

VE_GenericResult_t (see voipengine_user_v4.h)

3.4.2.3.14 VE_ADT_getRTPStatistics

This function returns current RTP statistics.

Prototype:

```
	VE_GenericResult_t VE_ADT_getRTPStatistics(VE_ADT_Handle_t
	_Handle, RTPStatistics_t *pRTPStatistics)
```

Inputs:

Handle is the handle to the VoIP Engine

pRTPStatistics – pointer to the statistics structure in which statistics will be returned

Outputs:

pRTPStatistics – contains returned RTP statistics

Returns:

VE_GenericResult_t (see voipengine_user_v4.h)

The currently implemented statistics include:

JitterEstimate – estimate of network jitter, expressed in units of samples
JitterDelay – current jitter buffer delay, expressed in units of samples

3.4.2.3.15 VE_ADT_setRTPPayloadType

This function sets the RTP payload type to be associated with a Codec for which there is no predefined RTP payload type.

Prototype:

```
      VE_GenericResult_t VE_ADT_setRTPPayloadType (
            VE_ADT_Handle_t _Handle,
            Codec_e Codec,
            ADT_UInt8 PayloadType
            );
```

Inputs:

Handle – VoIP Engine Handle
Codec – Codec type
PayloadType – Payload type to be associated with specified Codec

Outputs:

None

Returns:

```
      VE_RESULT_OK
      VE_RESULT_NOTOK (invalid codec or RTP payload for selected codec
cannot be modified.)
```

3.4.2.4 Callback and User-Supplied Callback Functions

The following functions need to be supplied by the host. These functions are called by the voice engine.

3.4.2.4.1 VE_ADT_cbOnSendRTPToNetwork

The VoIP Engine calls this function to send an RTP packet to the network stack. (This function is not necessary when using PCM Front End mode.)

The Channel ID indentifies which packet channel is calling this callback. It is the responsibility of the host software to send the RTP packet out using the correct destination IP address / UDP port based upon the Channel ID.

Prototype:

```
      void VE_ADT_cbOnSendRTPToNetwork(
            VE_Handle_t Handle,
            ADT_UInt8 ChannelID,
            ADT_UInt8 *pRTPPacket,
            ADT_Int32 Size);
```

Inputs:

Handle is the handle to the VoIP Engine

ChannelID identifies the packet channel associated with the RTP packet to be transmitted

pRTPPacket – points to RTP packet to be sent

Size – size in bytes of RTP packet

Outputs:

Returns:

3.4.2.4.2 VE_ADT_cbOnSendPCMToSpeaker

The VoIP engine calls this function to send a frame of PCM data to the speaker. (This function is not necessary when using PCM Front End mode.)

Prototype:

```
      void VE_ADT_cbOnSendPCMToSpeaker(
            VE_Handle_t Handle,
            ADT_Int16 *pPCM);
```

Inputs:

Handle is the handle to the VoIP Engine

pPCM – points to the PCM buffer. Size of the PCM buffer is determined at initialization time by FrameSize parameter.

Outputs:

Returns:

3.4.2.4.3 VE_ADT_cbOnEvent

(Not yet implemented)

Prototype:

```
      void VE_ADT_cbOnEvent(
            VE_Handle_t Handle,
            ADT_UInt16 EventSource,
            ADT_UInt16 EventCode);
```

Inputs:

`EventSource –` defines the source of the event. This can be an algorithm or the Voice Engine itself. Event Source definitions TBD.

`Event Code –` Event Codes TBD.

Outputs:

Returns:

3.4.2.4.4 VE_ADT_usMalloc

This function allocates a memory buffer.

Prototype:

```
      void *VE_ADT_usMalloc(ADT_Int32 Size);
```

Inputs:
Size – size in bytes of requested allocation
Outputs:

Returns:
pointer to allocated buffer. 0 if not successful

Example:

```
void *VE_ADT_usMalloc(ADT_Int32 Size)
{
      return(malloc(Size));
}
```

3.4.2.4.5 VE_ADT_usCalloc

This function allocates a memory buffer and zeros it out.

Prototype:

```
      void *VE_ADT_usMalloc(
            ADT_Int32 NumElements,
            ADT_Int32 ElementSize);
```

Inputs:
NumElements – the number of data elements to be allocated
ElementSize – the size, in bytes, of each data element
Outputs:
Clears out the allocated buffer
Returns:
pointer to allocated buffer. 0 if not successful

Example:

```
void *VE_ADT_usCalloc(ADT_Int32 NumElements, ADT_Int32 ElementSize)
{
      return(calloc(NumElements, ElementSize);
}
```

3.4.2.4.6 VE_ADT_usFree

This function frees a memory buffer.

Prototype:

```
VE_GenericResult_t VE_ADT_usFree(void *pPointer);
```

Inputs:

pPointer – pointer to buffer to be freed

Outputs:

Returns:

```
VE_RESULT_OK
VE_RESULT_NOTOK
```

Example:

```
VE_GenericResult_t VE_ADT_usFree(void *pPointer)
{
        if (pPointer != 0)
        {
                free(pPointer);
                return(VE_RESULT_OK);
        }
        else
                return(VE_RESULT_NOTOK):
}
```

3.4.2.4.7 VE_ADT_usCreateLock

This function creates a new instance of a mutex lock.

Prototype:

```
VE_ADT_Lock_t VE_ADT_usCreateLock(char *Name);
```

Inputs:

Name – a name for debug/diagnostic purposes.

Outputs:

Returns:

handle to a newly created lock

Example:

```
VE_ADT_Lock_t VE_ADT_usCreateLock(char *Name)
{
      sem_t *pSem = 0;
      int retval;
      pSem = malloc(sizeof(sem_t));
      if (pSem == 0)
      {
            printError("anVoice JNI","Unable to allocate a semaphore");
            return(0);
      }
      retval = sem_init(pSem,0,1);
      if (retval != 0)
      {
            printError("anVoice JNI", "Unable to initialize a
            sempahore");
            return(0);
      }
      return((VE_ADT_Lock_t) pSem);
}
```

iOS Example:

```
VE_ADT_Lock_t VE_ADT_usCreateLock(char *Name)
{
      dispatch_semaphore_t Sem = 0;
      int err;
      int retval;
      Sem = dispatch_semaphore_create(1);

      if (Sem == 0)
      {
            printf("Unable to allocate a semaphore");
            return(0);
      }

#ifdef LOCK_DEBUG
      printf("Created semaphore %s %d", Name, Sem);
#endif
      return((VE_ADT_Lock_t) Sem);
}
```

3.4.2.4.8 VE_ADT_usLock

This function acquires a lock.

Prototype:

```
      void VE_ADT_usLock(VE_ADT_Lock_t, char *pFunctionCallingName);
```

Inputs:

handle to lock

*pFunctionClalingName – pointer to string name of calling function – for debug/diagnostics

Outputs:

Returns:

Example

```
void VE_ADT_usLock(VE_ADT_Lock_t Lock, char *pCallingFunctionName)
{
      int retval;
#ifdef LOCK_DEBUG
      if (pCallingFunctionName != 0)
            printDebug("%s attempting lock %d", pCallingFunctionName,
Lock);
#endif
      retval = sem_wait((sem_t *) Lock);
      if (retval != 0)
            if (pCallingFunctionName != 0)
                  printError("anVoice JNI","lock failed! Lock Attempted
                  by %s, sem: %d return value: %d",pCallingFunctionName,
                  Lock, retval);
            else
                  printError("anVoice JNI","lock failed! sem: %d return
                  value: %d",Lock, retval);
#ifdef LOCK_DEBUG
      else
            if (pCallingFunctionName != 0)
                  printDebug("anVoice JNI","Lock succeeded!  Locked by
                  %s, sem: %d return value: %d",pCallingFunctionName,
                  Lock, retval);
#endif
```

}**iOS Example**

```
void VE_ADT_usLock(VE_ADT_Lock_t Lock, char *pCallingFunctionName)
{
      long retval;
#ifdef LOCK_DEBUG
      if (pCallingFunctionName != 0)
            printf("%s attempting lock %d ...", pCallingFunctionName,
            Lock);
#endif
      retval = dispatch_semaphore_wait((dispatch_semaphore_t) Lock,
      DISPATCH_TIME_FOREVER);
      if (retval != 0)
            if (pCallingFunctionName != 0)
                  printf("lock failed! return value: %d\n", retval);
            else
                  printf("lock failed! return value: %d\n", retval);
#ifdef LOCK_DEBUG
      else
            if (pCallingFunctionName != 0)
                  printf("Lock succeeded! return value: %d\n", retval);
#endif

}
```

3.4.2.4.9 VE_ADT_usUnlock

This function releases a lock.

Prototype:

```
void VE_ADT_usUnlock(VE_ADT_Lock_t, char *pCallingFunctionName);
```

Inputs:

handle to lock
*pCallingName – pointer to string name of calling function – for debug/diagnostics.

Outputs:

Returns:

Example

```
void VE_ADT_usUnlock(VE_ADT_Lock_t Lock, char *pCallingFunctionName)
{
      int retval;
#ifdef LOCK_DEBUG
      if (pCallingFunctionName != 0)
            printDebug("anVoice JNI", "%s attempting unlock %d",
pCallingFunctionName, Lock);
#endif

      retval = sem_post((sem_t *) Lock);
      if (retval != 0)
            if (pCallingFunctionName != 0)
                  printError("anVoice JNI","Unlock failed! Unlock
attempted by %s, sem: %d value: %d",pCallingFunctionName, Lock, retval);
            else
                  printError("anVoice JNI","Unlock failed! sem: %d
value: %d",Lock, retval);
#ifdef LOCK_DEBUG
      else
            if (pCallingFunctionName != 0)
                  printDebug("anVoice JNI","Unlock succeeded! Unlocked
by %s, sem: %d value: %d",pCallingFunctionName, Lock, retval);
#endif
}
```

iOS Example

```
void VE_ADT_usUnlock(VE_ADT_Lock_t Lock, char *pCallingFunctionName)
{
      int retval;
#ifdef LOCK_DEBUG
      if (pCallingFunctionName != 0)
            printf("%s attempting unlock... %d", pCallingFunctionName,
Lock);
#endif
      retval = dispatch_semaphore_signal((dispatch_semaphore_t) Lock);

#ifdef LOCK_DEBUG
      if (pCallingFunctionName != 0)
            printf("Unlock succeeded!  ret value = %d\n", retval);
#endif
}
```

3.4.2.4.10 VE_ADT_usDeleteLock

This function deletes (deallocates) a lock.

Prototype:

```
      void VE_ADT_usDeleteLock(VE_ADT_Lock_t);
```

Inputs:

handle to lock

Outputs:

Returns:

Example:

```
void VE_ADT_usDeleteLock(VE_ADT_Lock_t Lock)
{
      VE_ADT_usFree(Lock);
}
```

iOS Example:

```
void  VE_ADT_usDeleteLock(VE_ADT_Lock_t Lock)
{
      NSLock *myLock = (NSLock *) Lock;
      return([NSLock release]);
}
```

3.4.2.4.11 VE_ADT_cbOnCMRUpdate

This function is called when a new Codec Mode Request (CMR) is received by the AMR or G.722.2 Decoder

Prototype:

```
      void VE_ADT_cbOnCMRUpdate(
            VE_ADT_Handle_t Handle,
            ADT_UInt8 ChannelID,
            ADT_UInt16 NewCMRValue,
            ADT_UInt16 PrevCMRValue);
```

Inputs:

Handle – handle to VoIP Engine instance

ChannelID – ID of the G.722.2 or AMR channel the request was received on

NewCMRValue – Value of the newly received CMR request

PrevCMRValue – Value of the old CMR

Outputs:

Returns:

3.4.2.4.12 RTPTime

Prototype: ADT_UInt32 RTPTime(void);

This is an RTP support function to read current receiver clock value in units of RTP time stamp (samples.)

A pointer to this function is passed to the engine via the VE_ADT_start function, parameter Callbacks.

3.4.2.4.13 SRTPCallBack - GetKeys

Key values are provided to the VoIP Engine via an application dependent callback function. This function, coded by the SRTP user to meet the needs of the application's key scheme, is called whenever a new key value is required for encryption or decryption. The number of keys required for a session is dependent on the selected key management scheme, the key's derivation rate, and the duration of the session. For the Pre-Shared Master key scheme, the callback initially occurs prior to encrypting or decrypting the first packet and each time the key's lifetime has expired. For the Master Key Identifier scheme the callback occurs each time the MKI value is changed. For the From-To key schemes, the callback occurs whenever the sequence number is outside of the range for the current key.

Note that VoIP Engine currently supports only the Pre-Shared Master key scheme, which is the default scheme specified for use with RTP.

An application defined 'handle' is provided in the callback to identify the particular SRTP instance to the callback routine. The handle may be any value of convenience to the application. The callback returns status to indicate if the key has been provided.

The prototype of the callback function is:
Prototype:

```
ADT_Bool GetKey(void *pHandle, SrtpKeyInfo_t *pKeyInfo);
```

Inputs

pHandle is used to identify the packet channel in question as well as whether or not the callback is being called for the encrypt operation or the decrypt operation. pHandle is bit mapped as follows:

Bit 0: 1 for decrypt, 0 for encrypt
Bits 7..1: reserved
Bits 31..8: Channel ID

pKeyInfo is a pointer to the location where the key information will be returned

Outputs

pKeyInfo: Points to a structure defining a key value and its parameters. The structure contains common and key management scheme specific elements. The application must use only those elements applicable to the key management scheme selected in the instance's configuration. The elements of KeyInfo structure are listed in the table below.

Element	**Description**
pKeyValue	Pointer to 16-bit aligned buffer containing the new master key and set by caller. The master key length must match that defined for the instance.
pSaltValue	Pointer to 16-bit aligned buffer containing the new master salt and set by caller. The master salt length must match that defined for the instance.
KeyDerivationRate	The key derivation rate as a power of 2. Upon entry, initialized to a value that disables key derivation. Can be overridden by the application to a value from 0 (new key every packet) to 24 (new key every 2^{24} packets) inclusive. Upon return, enabled only if less than or equal to 24.
MaxKeyLife	Maximum key lifetime. Upon entry, initialized to the larget value allowed. Can be overridden by the application. Upon teturn, the value is limited to the largest value allowed. Applicable only to Psk and Mki schemes.
MkiValue	N/A – applies only to MKI schemes
From	N/A – appies only to from-to schemes
To	N/A - applies only to from-to schemes

SRTP Key Information Structure

3.4.2.4.14 TestPointCallback

Prototype:

```
typedef ADT_UInt8 (TestPointCallback_t)(
            VE_ADT_Handle_t Handle,
            ADT_UInt8 ChannelID,
            TestPoint_e TP,
            void *pBuffer,
            ADT_UInt16 NWords,
            ADT_UInt8 BytesPerWord
      );
```

Inputs:

Handle – VoIPEngine handle
ChannelID – Specifies the channel ID associated with the specified test point. The primary channel is always ID 0. Additional packet channels utilize subsequent channel IDs.
TP – Test point identifier. See enumerated values in voipengine_user_v4.h. The available test points are indicatd on the PCM Front End and Packet Channel block diagrams earlier in this document.
pBuffer – pointer to buffer where test point data is transferred from the engine test point to the host software and optionally from the host software back to the engine test point.
NWords – number of words in the buffer
BytesPerWord – buffer word size, measured in units of bytes.

Outputs:

pBuffer - The host may optionally overwrite the data in the transfer buffer

Returns:

1 if host has overwritten the buffer. 0 otherwise.

3.4.2.5 Parameter Handling Functions

The primary parameter handling functions are VE_ADT_setParamValue and VE_ADT_getParamValue. For a description of the parameters, and associated modules, please refer to the VoIP Engine Parameter section of this document titled

Beyond VE_ADT_setParam and VE_ADT_getParam, there are other functions that enable the host to use parameters by assigned string names. This feature is built into the engine primarily for use by Adaptive Digital's sample applications. The functions are listed below, but since they are not essential to the use of VoIP Engine and their use is straightforward, no additional explanation is provided.

```
void VE_ADT_getModuleNames(VE_ADT_Handle_t Handle, char **ppNames,
ADT_UInt8 *pNModules);

void VE_ADT_getShortModuleNames(VE_ADT_Handle_t Handle, char **ppNames,
ADT_UInt8 *pNModules);
```

```
int  VE_ADT_getParamNames(VE_ADT_Handle_t Handle, ADT_UInt8 ModuleID, 
char **ppNames, ADT_UInt8 *pNParams);

int  VE_ADT_getShortParamNames(VE_ADT_Handle_t Handle, ADT_UInt8 
ModuleID, char **ppNames, ADT_UInt8 *pNParams);

int  VE_ADT_getParamValue(VE_ADT_Handle_t Handle,ADT_UInt8 ModuleID, 
ADT_UInt8 ParamID, ADT_Int32 *pValue, char *pStringValue);

int  VE_ADT_setParamValue(VE_ADT_Handle_t Handle,ADT_UInt8 ModuleID, 
ADT_UInt8 ParamID, ADT_Int32 Value);

ADT_UInt8 VE_ADT_getModuleIDFromName(VE_ADT_Handle_t Handle, 
ParamString_t ModuleName);

ADT_UInt8 VE_ADT_getParamIDFromName(VE_ADT_Handle_t Handle, ADT_UInt8 
ModuleID, ParamString_t ParamName);

ADT_UInt8 VE_ADT_getParamIDFromModuleAndParamNames(VE_ADT_Handle_t 
Handle, ParamString_t ModuleName, ParamString_t ParamName);
```

3.5 Miscellaneous Functions

3.5.1 VE_ADT_getBuildOptions

Returns information about which features were built into the VoIP Engine library.

Prototype:

```
ADT_API VE_GenericResult_t VE_ADT_getBuildOptions(
        VE_ADT_BuildOptions_t *pOptions
);
```

Inputs

Outputs

pOptions – See definition of VE_ADT_BuildOptions_t in voipengine_user_v4.h for details.

Returns

VE_GenericResult_t (see voipengine_user_v4.h)

4. Diagnostic Support

4.1 Test Modes

The VoIP engine has a number of diagnostic features built-in. Diagnostics are peformed by placing VoIP Engine in test mode using calling function VE_ADT_setParamValue as follows before starting (VE_ADT_start) a new call.

VE_ADT_setParamValue(VEMODULE_VE, VEPARAM_TEST_MODE, <test mode>);

where <test mode> is defined in VE_testMode_t in voipengine_user_v4.h

Test modes are described briefly here. Some test modes require more than the brief descriptions. Those test modes are described in more detail below.

4.2 AEC Data Capture

AEC Data Capture is different from the test modes in that it is used during a normal voice call rather than in a special mode of operation. In order to diagnose echo problems or further characterize the speaker to microphone echo path, it is extremely useful to obtain the AEC input and output buffers during live operation.

To enable this feature, you can use API function VE_ADT_setCapturePoints.

5. Product Software

The product is distributed as a set of one or more object libraries. Table 5-1 lists the distribution files and describes each one.

File Name	Description
libVoIPEngine.a (or similar)	Object Code Library
voipengine_user_v4.h	Header files
VoIPEngine_v4_UsersGuide.pdf	Users Guide
VoIPEngineReleaseNotes_v4.pdf	Release Notes

Table 5-1: Distribution Files

Sample host files can be provided on an as-is basis.