Adaptive DSP Access Manager (ADAM) Programmer Reference Manual

Version: 1.0.0
Release date: 2011-09-28

Table of Contents

1 Revision History

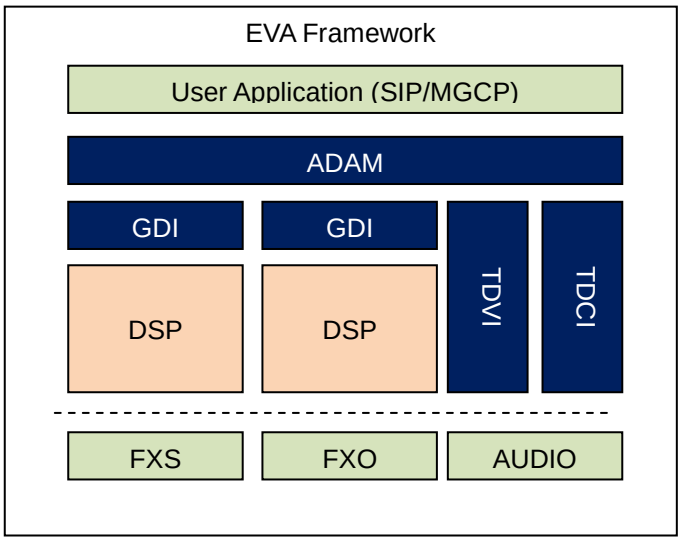
Revision history:

Revision	Author	Date	Description
1.0.0	Quark	2011/09/28	Initial version

2 Introduction

ADAM, Adaptive DSP Access Manager, is one of the modules in EVA framework who provides a consistent single access point to manipulate variant DSP(s). It does not just provide consistent API to application such that application requires no change when DSP changed, but also gives the developer the freedom to add his/her own logic to extend the VoIP related functions, such as, call logs, stream recording, etc. Also, ADAM is designed to be able to handle multiple DSP simultaneously, therefore, it is highly flexible to design scalable VoIP product from low channels CPE to high density gateway products.

2.1 EVA Framework

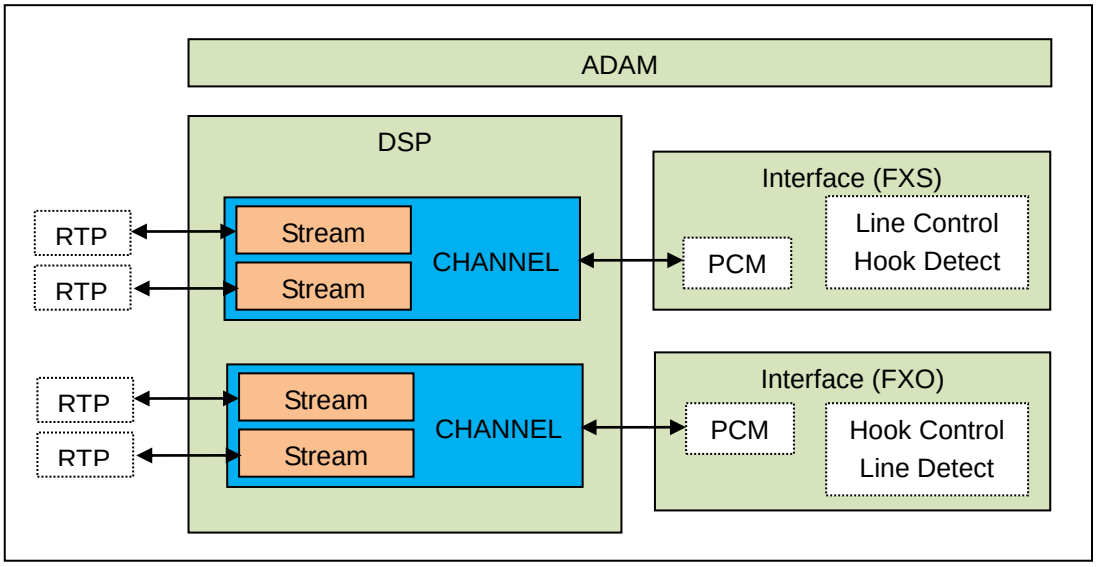


EVA, Enhanced VoIP Architecture, is a framework to redefine VoIP components in clear layers and object model. It provides the portability, scalability, flexibility, and transparency in developing VoIP product. The ADAM can provide a consistent DSP access interface to one or many DSPs and the application requires no change when the underlying DSP changes. ADAM is also open to developers to allow developer to add his/her own logic in ADAM to twist, convert, or replace the DSP logics. For example, some DSP handles telephony hardware interface control, but developer may obsolete the DSP control and implement his/her own interface control in ADAM internal function(s). However, to the application who calls ADAM APIs, it does not know the function has been replaced by different implementation.

The GDI, Generic DSP Interface, is the middle layer framework for consistent DSP functionalities abstract. ADAM control DSP(s) through GDI API. GDI framework is open to user who wants to add different DSP modules. However, GDI implementation might be closed due to license issue based on the DSP module provider. Fortunately, users do not need to see the GDI internal implementation which is mainly calling the DSP API, but just need to know the DSP functions are delivered as it should as defined in GDI API.

EVA is an evolving open framework to developer. New API will be added to provide better functionalities. We wish you enjoy using EVA framework and welcome your contribution to EVA. For more information about EVA, please refer to EVA_Framework_Introduction.

2.2 Object Model



EVA uses object model to simplify the DSP control process. ADAM controls and management DSP operation based on this object model concept.

Interface:
Interface is a telephony hardware user interface abstract which could be FXS, FXO, DECT module, or other telephony handset types. Interface is mainly responsible to reflect the user operation, such as off hook the phone or generate specific event to notify application for certain operation. On the other hand, ADAM control interface to deliver specific notification to the user, such as ringing.

Though PCM is exchange through the interface hardware, the Interface abstract does not get involve with any PCM control or manipulation. (** See TDCI, Telephony Device Control Interface, and TDVI, Telephony Device Voice Interface, in EVA_Framework_Introduction for further detail.)

Channel:
Channel is one independent DSP process which handles a PCM raw input and output of an Interface. A DSP may consist of one or many channels and each channel's DSP process should be able to configured independently without interference.

The PCM raw input may go through various DSP process, such as echo cancellation, tone detection, voice activity detection, noise floor estimation, gain adjustment, etc. Relative event may be generated if the PCM raw input meets the condition. And the processed PCM raw data might be mixed with a decoded stream PCM raw data if conference is required. Then, Channel pass this PCM output raw data to a Stream process and encode it to coded format and transmit it to the destination peer.

On the other direction, Channel takes a decoded PCM raw data from a Stream and put these PCM raw data through certain DSP process, such as, comfort noise generation, packet loss compensation, tone detection, noise floor estimation, etc. Then, it might mix with another Stream decoded and DSP processed PCM raw data if conference is required. And it goes through some further DSP process, such as gain adjustment, echo reference. Then finally, pass the PCM raw data output to the interface.

User need to control Channel object to decide which DSP process needs to be enable and which to be turned off.

Stream:
Stream is to represent an encoding/decoding process and its pre-/post- process. Stream is usually attached to a Channel's DSP process. A Channel may consist of one or many Stream depending on the DSP, for CPE product, it is usually two for supporting 3-way conference call.

Each Stream configuration can be configured independently without interference. For example, in a 3-way conference call, user can configure the stream direction of a Stream to put one Stream on hold.

Stream configuration can be configured at anytime regardless the Stream active state and the configuration changes take effect immediately.

DSP:
DSP is the abstract of a DSP main body which could be a physical DSP hardware or DSP software. There are certain configurations are DSP-wide, such as tone generation template and tone detection template. Moreover, DSP is the main host of Channel and Stream. To use their functions, DSP must be initialized first and shutdown properly to release the resource when the DSP service is no longer required.

2.3 ADAM API Command and Event

User may control DSP(s) and Interface(s) through ADAM API Commands and get to know DSP and Interface status by reported events. By handling the event properly with application logic and control DSP with ADAM API when application required DSP control, user can develop VoIP application easily without worrying the internal process of DSP.

According to the Object Model, ADAM APIs are categorized into five categories, ADAM Generic, DSP, Channel, Stream, and Interface. Each API perform specific task to its subject object. Some of them may be used equivalently, such as user can call adamStrmStopTone to stop a tone generation, or call adamStrmPlayTone with silence tone to get the same result. Another example is adamInfcRing, alternatively user can call adamInfcConfigLine and set the line state to RING to get the same effect. In another word, users do not need to use all API but can select the API he/she preferred and use the same API in different occasions. Here is a list of ADAM APIs:

Command List:

Command	Description
ADAM	
adamInit	Initialize ADAM and get DSP(s) handles.
adamExit	Quit ADAM, release any allocated resources.
adamQuery	Get ADAM capability, such as number of DSP hooked and interface numbers, etc.
adamPollEvent	The single event access point to retrieve event from DSP(s) and Interface(s).
DSP	

Command	Description
adamDspInvoke	Initialize and start the DSP process.
adamDspRevoke	Shutdown and terminate the DSP process.
adamDspQuery	Get DSP capability information, such as CODEC and detectors support, etc.
adamDspConfigTone	Configure the tone template to a DSP.
Channel	
adamChanQuery	Get a channel configuration.
adamChanConfig	Change a channel configuration, such as its detectors' active state and Tx/Rx Gain.
adamChanPlayTone	Generate tone(s) on a channel (to interface).
adamChanStopTone	Stop a tone generation.
adamChanPlayCid	Manually generate caller ID signal on a channel.
adamChanPlayType2Cid	Generate Type-II caller ID on a channel.
adamChanDumpPcm	Enable raw Tx/Rx PCM data dump to a network peer.
Stream	
adamStrmQuery	Get the configuration of a stream and its active state.
adamStrmConfig	Change the configuration of a stream.
adamStrmStart	Start the streaming process of a channel.
adamStrmStop	Stop the streaming process of a channel.
adamStrmSendDtmfr	Manually send DTMF relay packet to the network peer.
adamStrmPlayTone	Generate tone(s) to the network peer.
adamStrmStopTone	Stop the tone generation to the network peer.
Interface	
adamInfcQuery	Get interface configuration and line/hook state.
adamInfcConfigLine	Change the line state of an FXS interface, limited to certain states only.
adamInfcConfigHook	Change the hook state of an FXO interface.
adamInfcConfigRing	Change the ring configuration of an FXS interface.
adamInfcRing	Set FXS interface to RING state.
adamInfcStopRing	Stop FXS ringing.

Event List:

EVENT	Description
EVENT_CODE_INVALID	Return value in NON-BLOCKING mode when no valid event available.
EVENT_CODE_TONE	Notify application of a tone detection event. Including DTMF, Modem (FAX), Call Progress Tone (CPT).
EVENT_CODE_CID	Notify application of receiving CID signal and

EVENT	Description
	CID context.
EVENT_CODE_LINE	Notify application of line state change.
EVENT_CODE_HOOK	Notify application of hook state change.
EVENT_CODE_JB_UPDATE	Update jitter buffer statistic information.
EVENT_CODE_NON_RTP_RECVD	Notify application of receiving unidentified packet in RTP port and packet context, could be used for sending/receiving STUN packet for RTP port open; Not used now.
EVENT_CODE_RTCP_SEND	Notify application of RTCP sending event and RTCP context; Not used now.
EVENT_CODE_RTCP_RECVD	Notify application of RTCP receiving event and RTCP context; Not used now.
EVENT_CODE_STREAM_UPDATE	Update stream statistic information; Not used now.
EVENT_CODE_TIMER	Generate notification to application based on DSP ticks and user configured interval; Not used now.
EVENT_CODE_PERFORMANCE	To update DSP benchmark information, such as DSP uptime and average MHz consumption; Not used now.
EVENT_CODE_ERROR	To report DSP error; Not used now.

2.4 Glossary

ADAM Adaptive DSP Access Manager

Cadence A combination of signal on and off for certain time is called a cadence.

Caller ID (CID) A telephony signal standard to indicate subscriber (caller) identification, usually telephone number, and other information, such as user name, calling date and time. ** There is Type-1 caller ID which is known as the on-hook caller ID. The caller ID is transmitted during the telephone ringing; There is also Type-2 caller ID which is known as the “call-waiting” caller ID or the off-hook caller ID. The caller ID is transmitted during a call-waiting request (only certain countries provide Type-2 caller ID service).

CPT Call progress tone. Telephony signals used to indicate the state of service. i.e. Dial-tone indicates a line is ready for dialing out. Busy-tone indicates a line is occupied and cannot reach its destination.

Channel A DSP process path connecting the PCM I/O from a physical audio hardware to a network CODEC I/O.

CNG Comfort Noise Generation. By incorporating with VAD and silence compression and generate artificial background noise to save bandwidth and improve talking experience.

CODEC Coded/Encoded, usually implies a process of conversion between raw data and compressed (coded) data.

DAA Data Access Arrangement. A hardware component emulate a POTS phone to provide FXO function.

DSP	Digital Signal Processor
DTMF	Dual-Tone Multi-Frequency, a telephone standard to indicate (signaling) digits.
DTMF Relay	A RFC standard (RFC2833 obsolete by RFC4733) to transmit DTMF information in RTP payload instead of in-band audio to provide reliable DTMF transmission.
Echo Cancellation (Echo Canceller)	A process to remove echo.
EVA	Enhanced VoIP Architecture.
FXO	Foreign Exchange Office, a telephony endpoint (Telephone) or device used to signal Central Office (CO) its request or response of a phone call.
FXS	Foreign Exchange Station, a telephony endpoint or device at Central Office (CO) side to provide signal and power for FXO.
Interface	An interface is an abstract of a physical audio hardware.
OP Code	EVCOM operation code, a short conversion of EVCOM command.
P-time / P-rate	Packetization time (rate) used to negotiate and indicate the length (ms) of the audio in each packet payload.
SAS	Subscriber Alert Signal. A signal to alert the user (telephone) a call is waiting, may be followed with type-2 caller ID.
Silence Compression	A method to save bandwidth consumption by transmit silence indication packet (SID) instead of full RTP payload when user is not talking.
SLIC	Subscriber Line Interface Circuit. A hardware component emulate CO service to provide FXS function.
Stream	Stream is a path or process to disassembly sequential coded data (i.e. audo), transmit over network, and reassembly the coded data on the far-end to restore the original information.
VAD	Voice Activity Detection. A method to assess the audio level to determine if a user is talking.

3 ADAM Management

3.1 adamInit

Prototype:
exCode_e adamInit(void);

Return Values:
EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
None	

Description:
Initialize ADAM to get all DSP handles.

Example Code:

```
printf("Initializing ADAM ... \n");

if (EXEC_SUCCESS != adamInit()) {
        printf("ADAM initialization failed!!\n");
}
```

3.2 adamExit

Prototype:
exCode_e adamExit(void);

Return Values:
EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
None	

Description:
Terminate ADAM and release any allocated resources.

Example Code:

```
printf("Shutdown ADAM ... \n");

if (EXEC_SUCCESS != adamExit()) {
        printf("ADAM shutdown failed!!\n");
}
```

3.3 adamQuery

Prototype:
exCode_e adamQuery(adamConfig_t *pAdamConf);

Return Values:
EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
pAdamConf	Pointer to an adamConfig_t instance to receive the ADAM configuration.

Description:
Query ADAM information, such as version, number of DSP support, etc.

Example Code:

```
if (EXEC_SUCCESS == adamQuery(&adConf)) {
        printf("ADAM Version: %s\n", adConf.version);
        printf("Number of DSP: %d\n", adConf.dspNum);
        printf("Number of Interface: %d\n",  adConf.infcNum);
}
```

3.4 adamPollEvent

Prototype:

exCode_e adamPollEvent(blockMode_e mode, event_t *pEvent);

Return Values:
EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
Mode	Event polling mode: NON-BLOCKING – the function will return immediately regardless if a valid event presented. Return EXEC_SUCCESS when there is a valid event and EXEC_FAIL when no valid event received. BLOCKING – the function will not return until a valid event is received. In BLOCKING mode, the return valid is always EXEC_SUCCESS.
pEvent	Pointer to an event_t instance to receive the event data.

Description:
Polling event from DSP.

Example Code:
(See *Chapter 8 - ADAM EVENT Processing*)

4 DSP Control

4.1 adamDspInvoke

Prototype:
exCode_e adamDspInvoke(dspId_e dsp);

Return Values:
EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
dsp	DSP ID.

Description:
Invoke DSP will initialize the DSP and start DSP process.

Example Code:

```
printf("Invoking DSP ...\n\n");

if (EXEC_SUCCESS != adamDspInvoke(DSP_VIKING)) {
        printf("Error: DSP initialization failed!\n");
}
```

4.2 adamDspRevoke

Prototype:
exCode_e adamDspRevoke(dspId_e dsp);

Return Values:
EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
dsp	DSP ID.

Description:
Revoke DSP will shutdown the DSP process.

**NOTE: Depending on the DSP capability, once DSP is revoked, it might not be able to re-invoke again unless system reboot or other external process executed.

Example Code:

```
printf("Revoking DSP ...\n\n");
if (EXEC_SUCCESS != adamDspRevoke(DSP_VIKING)) {
        printf("Error: DSP shutdown failed!\n");;
}
```

4.3 adamDspQuery

Prototype:
exCode_e adamDspQuery(dspId_e dsp, activeState_e *dspActive, dspFeature_t *pFeature);

Return Values:
EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
dsp	DSP ID.
dspActive	DSP active state
pFeature	Pointer to a dspFeature_t instance to receive the DSP supported feature information.

Description:
Query the DSP capability information.

Example Code:

```
activeState_e active;
dspFeature_t feature;

if (EXEC_SUCCESS != adamDspQuery(DSP_VIKING, &active, &feature)) {
        printf("Error: Cannot get DSP status, application will quit now!\n\n");
        exit(-1);
}

printf("DSP features: \n");
printf("Active status: %s\n", (active));
printf("DSP ID: (%d)\n", feature.dspId);
printf("Number of Channel: %d\n", feature.numOfChan);
printf("Stream per Channel: %d\n", feature.strmsPerChan);
```

4.4 adamDspConfigTone

Prototype:
exCode_e adamDspConfigTone(dspId_e dsp, uint16 toneId, tone_t *pTone);

Return Values:
EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
Dsp	DSP ID.
toneId	Tone ID, ** 0~29, but tone ID [0] is reserved for silence which cannot be configured.
pTone	Pointer to a tone_t instance which contains the tone configuration information.

Description:
Configure a tone (generation) template in the DSP.

Example Code:

```
tone_t mTone;
int toneId;

toneId = 1;
mTone.toneType = TONE_REGULAR;

mTone.regular.toneFreq[0] = 350; /* Hz */
mTone.regular.tonePwr[0] =  -18 * 2; /* -18db */
mTone.regular.toneFreq[1] = 440; /* Hz */
mTone.regular.tonePwr[1] =  -18 * 2; /* -18db */

mTone.makeTime[0] = 500;
mTone.breakTime[0] = 1000;
mTone.repeat[0] = 5;

if (EXEC_SUCCESS !=  adamDspConfigTone(DSP_VIKING, toneId, &mTone)) {
        printf("Error: adamDspConfigTone failed!\n");
        return;
}
```

5 Channel Control

5.1 adamChanQuery

Prototype:
exCode_e adamChanQuery(dspId_e dsp, chanId_e ch, chanConfig_t *pChanConf);

Return Values:
EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
dsp	DSP ID.
ch	Channel ID
pChanConf	Pointer to a chanConfig_t instance to receive the channel configuration information.

Description:
Query the configuration of a channel

Example Code:

```
chanConfig_t config;
chanId_e          ch = 0;

if (EXEC_SUCCESS != adamChanQuery(DSP_VIKING, ch, &config)) {
          printf("Execution failed! Cannot retrieve channel configuration.\n\n");
          return;
}

printf("Channel (%d) configuration:\n", ch);
printf("Enabled Detectors:\n");
printf("Detect (DTMF_TONE)=%d\n", (config.detectMask & DETECT_TONE_DTMF));
printf("Detect (FAX/MODEM_TONE)=%d\n", (config.detectMask & DETECT_TONE_MODEM));
printf("Detect (CALL_PROGRESS_TONE)=%d\n", (config.detectMask & DETECT_TONE_CPT));
printf("Detect (Caller_ID)=%d\n", (config.detectMask & DETECT_CID));
printf("EC Control =%d\n", (config.ecEnable));
printf("Tx Gain: %ddb\n", config.ampTx/2);
printf("Rx Gain: %ddb\n", config.ampRx/2);
```

5.2 adamChanConfig

Prototype:
exCode_e adamChanConfig(dspId_e dsp, chanId_e ch, chanConfig_t *pChanConf);

Return Values:
EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
dsp	DSP ID.
ch	Channel ID
pChanConf	Pointer to a chanConfig_t instance which contains the configuration information.

Description:
Change the configuration of a channel.

Example Code:

```
chanConfig_t config;
chanId_e  ch = 0;

config.ampTx = (-3) * 2; /* -3db */
config.ampRx = 3 * 2;  /* +3db */
if (EC_ON) {
        config.ecEnable = CONTROL_ENABLE;  /* Enable Echo Cancellation */
}
else {
        config.ecEnable = CONTROL_DISABLE;  /* Disable Echo Cancellation */
}

if (DTMF_DETECT_ON) {
        config.detectMask |= DETECT_TONE_DTMF;  /* Enable DTMF Detection */
}
else {
        config.detectMask &= ~(DETECT_TONE_DTMF); /* Disable DTMF Detection */
}

if (MODEM_DETECT_ON) {
        config.detectMask |= DETECT_TONE_MODEM; /* Enable FAX/Modem Detection */
}
else {
        config.detectMask &= ~(DETECT_TONE_MODEM); /* Disable FAX/Modem Detection */
}

if (CPT_DETECT_ON) {
        config.detectMask |= DETECT_TONE_CPT; /* Enable Call Progress Tone Detection */
}
else {
        config.detectMask &= ~(DETECT_TONE_CPT); /* Disable Call Progress Tone Detection */
}

if (CID_DETECT_ON) {
        config.detectMask |= DETECT_CID; /* Enable Caller ID Detection, only for FXO interface */
}
else {
        config.detectMask &= ~(DETECT_CID); /* Disable Caller ID Detection, only for FXO interface */
}

if (EXEC_SUCCESS !=  adamChanConfig(DSP_VIKING, ch, &config)) {
        printf("Error: adamChanConfig failed!\n");
        return;
}
```

5.3 adamChanPlayTone

Prototype:
exCode_e adamChanPlayTone(dspId_e dsp, chanId_e ch, toneSeq_t *pToneSeq, uint32 repeat);

Return Values:
EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
dsp	DSP ID.
ch	Channel ID
pToneSeq	Pointer to a toneSeq_t instance which contains a sequence of tones and number of tones.
Repeat	Times to repeat the tone sequence generation.

Description:
Generate tone(s) on a channel.

Example Code:

```
chanId_e ch=0;
toneSeq_t toneList;
const int sz = 3;
int mToneId[sz] = {3, 2, 1};
int rpt = 2;

toneList.numOfTone = sz;
toneList.toneIdSeq = (uint8*)mTone;

if (EXEC_SUCCESS != adamChanPlayTone(DSP_VIKING, ch, &toneList,rpt)) {
        printf("Error: adamChanPlayTone failed! \n");
        return;
}
```

5.4 adamChanStopTone

Prototype:

exCode_e adamChanStopTone(dspId_e dsp, chanId_e ch);

Return Values:
EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
dsp	DSP ID.
ch	Channel ID

Description:
Stop tone generation on a channel.

Example Code:

```
if (EXEC_SUCCESS != adamChanStopTone(DSP_VIKING, 0)) {
        printf("Error: adamChanStopTone failed! \n");
        return;
}

/* Another alternative is to play tone[0] (Silence) to stop a tone play */
toneSeq_t toneList = {
        . numOfTone = 1,
        . toneIdSeq = {0}
};

if (EXEC_SUCCESS != adamChanPlayTone(DSP_VIKING, CH0, &toneList, 1)) {
        printf("Error: adamChanPlayTone failed! \n");
        return;
}
```

5.5 adamChanPlayCid

exCode_e adamChanPlayCid(dspId_e dsp, chanId_e ch, cid_t *pCid);

Return Values:
EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT

DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
dsp	DSP ID.
ch	Channel ID
pCid	Pointer of a cid_t instance which contains caller ID information.

Description:
Manually generate a caller ID on the channel (to interface) without any leading signal or interface control (ring).

** NOTE: This API is provided for advance user who would like to do manually CID transmission and control the SLIC operation on his/her own. To correctly transmit CID manually, the SLIC must also be configure properly prior to call this API.

Example Code:

```
chanId_e ch = 0;
cid_t cid = {
        .number = “88888888”,
        .name = “”,
        .dateTime = “”
};

if (EXEC_SUCCESS != adamChanPlayCid(DSP_VIKING, ch, &cid)) {
        printf("Error: adamChanPlayCid failed! \n");
        return;
}
```

5.6 adamChanPlayType2Cid

exCode_e adamChanPlayType2Cid(dspId_e dsp, chanId_e ch, cid_t *pCid);

Return Values:
EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description

dsp	DSP ID.
ch	Channel ID
pCid	Pointer of a cid_t instance which contains caller ID information.

Description:
Generate a Type-II (off-hook) caller ID on the channel (to interface).

Example Code:

```
chanId_e ch = 0;
cid_t cid = {
        .number = “88888888”,
        .name = “”,
        .dateTime = “”
};

if (EXEC_SUCCESS != adamChanPlayType2Cid(DSP_VIKING, ch, &cid)) {
        printf("Error: adamChanPlayType2Cid failed! \n");
        return;
}
```

5.7 adamChanDumpPcm

Prototype:
exCode_e adamChanDumpPcm(dspId_e dsp, chanId_e ch, netAddr_t *pDstAddr);

Return Values:
EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
dsp	DSP ID.
ch	Channel ID
pDstAddr	Pointer to a netAddr_t instance which contains the information of destination endpoint to be receive the dump data. ** Configure IP address as: 0.0.0.0 to disable the dump.

Description:

Enable channel PCM dump process for debugging. When PCM dump enabled, the PCM Tx/Rx raw data of a channel will be sent to the designated network address and port in RTP format. User may capture these packet with sniffer tool, i.e. Wireshark, extract the payload and restore the audio.

** NOTE: To disable a dump process, configure the destination IP address to 0x0 (0.0.0.0).

Example Code:

```
chanId_e ch;
netAddr_t mAddr = {
        .ver = IPV4,
        .addrV4 = inet_addr("192.168.1.200");
};

if (EXEC_SUCCESS != adamChanDumpPcm(DSP_VIKING, ch, &mAddr)) {
        printf("Error: adamChanDumpPcm failed! \n");
}
```

6 Stream Control

6.1 adamStrmQuery

Prototype:
exCode_e adamStrmQuery(dspId_e dsp, chanId_e ch, strmId_e strm, activeState_e *pStrmActive, strmConfig_t *pStrmConf);

Return Values:
EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
dsp	DSP ID.
ch	Channel ID
strm	Stream ID
pStrmActive	Pointer to an activeState_e instance to receive stream active state.
pStrmConf	Pointer to a strmConfig_t instance to receive the stream configuration.

Description:
Query the configuration of a stream.

Example Code:

```
chanId_e ch = 0;
strmId_e st = 0;
strmConfig_t config;
activeState_e active;

if (EXEC_SUCCESS != adamStrmQuery(DSP_VIKING, ch, st, &active, &config)) {
        printf("Execution failed! Cannot retrieve stream configuration.\n\n");
}

printf("Channel %d -> Stream %d Configuration:\n", ch, st);
printf("Stream state: %d\n", active);
printf("Source address: 0x%x:%d\n", config.session.srcAddr.addrV4, config.session.srcAddr.port);
printf("Destination address: 0x%x:%d\n", config.session.dstAddr.addrV4, config.session.dstAddr.port);
printf("Codec: %d\n",  config.strmAttr.payloadSelect);
printf("Ptime: %d\n", config.strmAttr.ptimeSelect);
printf("Silence compression: %d\n", config.strmAttr.silenceComp);
printf("DTMF Relay: %d\n", config.strmAttr.dtmfRelay);
printf("Stream direction: %d\n", config.strmAttr.direction);
```

6.2 adamStrmConfig

Prototype:

exCode_e adamStrmConfig(dspId_e dsp, chanId_e ch, strmId_e strm, strmConfig_t *pStrmConf);

Return Values:

EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
dsp	DSP ID.
ch	Channel ID
strm	Stream ID
pStrmConf	Pointer to a strmConfig_t instance which contains the stream configuration information.

Description:

Change the configuration of a stream.

Example Code:

```
chanId_e ch = 0;
strmId_e st = 0;
strmConfig_t config;

config.session.srcAddr.addrV4 = inet_addr(“192.168.1.1”);
config.session.srcAddr.port = 5000;
config.session.dstAddr.addrV4 = inet_addr(“192.168.1.100”);
config.session.dstAddr.port = 5000;
config.strmAttr.payloadSelect = CODEC_G711A;
config.strmAttr.ptimeSelect = PTIME_20MS;
config.strmAttr.silenceComp = CONTROL_ENABLE;
config.strmAttr.dtmfRelay = CONTROL_ENABLE;
config.strmAttr.direction = STRM_SENDRECV;

if (EXEC_SUCCESS != adamStrmConfig(DSP_VIKING, ch, st, &config)) {
        printf("Execution failed! Cannot set stream configuration.\n\n");}
}
```

6.3 adamStrmStart

Prototype:

exCode_e adamStrmStart(dspId_e dsp, chanId_e ch, strmId_e strm);

Return Values:

EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
dsp	DSP ID.
ch	Channel ID
strm	Stream ID

Description:

Start streaming process of a channel.

Example Code:

```
if (EXEC_SUCCESS != adamStrmStart(DSP_VIKING, CH0, STRM0)) {
        printf("Execution failed! Cannot start stream .\n\n");
}
```

6.4 adamStrmStop

Prototype:
exCode_e adamStrmStop(dspId_e dsp, chanId_e ch, strmId_e strm);

Return Values:
EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
dsp	DSP ID.
ch	Channel ID
strm	Stream ID

Description:
Stop streaming process of a channel.

Example Code:

```
if (EXEC_SUCCESS != adamStrmStop(DSP_VIKING, CH0, STRM0)) {
        printf("Execution failed! Cannot stop stream .\n\n");
}
```

6.5 adamStrmSendDtmfr

Prototype:
exCode_e adamStrmSendDtmfr(dspId_e dsp, chanId_e ch, strmId_e strm, dtmf_e dtmf, uint32 dur);

Return Values:
EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
dsp	DSP ID.

ch	Channel ID
strm	Stream ID
dtmf	DTMF digit: 0~9, *, #, A, B, C, D
dur	Duration (ms).

Description:
Manually generate DTMF relay packet (RFC2833/4733) to a stream.

Example Code:

```
if (EXEC_SUCCESS != adamStrmSendDtmfr(DSP_VIKING, CH0, STRM0, DTMF_1, 1000)) {
        printf("Execution failed! Cannot send stream dtmfr .\n\n");
}
```

6.6 adamStrmPlayTone

Prototype:
exCode_e adamStrmPlayTone(dspId_e dsp, chanId_e ch, strmId_e strm, toneSeq_t *pToneSeq, uint32 repeat);

Return Values:
EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
dsp	DSP ID.
ch	Channel ID
strm	Stream ID
pToneSeq	Pointer to a toneSeq_t instance which contains a sequence of tones and number of tones.
Repeat	Times to repeat the tone sequence generation.

Description:
Generate tone(s) to a stream.

Example Code:

```
chanId_e ch=0;
toneSeq_t toneList;
const int sz = 3;
int mToneId[sz] = {3, 2, 1};
int rpt = 2;

toneList.numOfTone = sz;
toneList.toneIdSeq = (uint8*)mTone;

if (EXEC_SUCCESS != adamStrmPlayTone(DSP_VIKING, ch, st, &toneList, rpt)) {
        printf("Execution failed! Cannot play stream tone .\n\n");
}
```

6.7 adamStrmStopTone

Prototype:
exCode_e adamStrmStopTone(dspId_e dsp, chanId_e ch, strmId_e strm);

Return Values:
EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
dsp	DSP ID.
ch	Channel ID.
strm	Stream ID.

Description:
Stop tone generation to a stream.

Example Code:

```
if (EXEC_SUCCESS != adamStrmStopTone(DSP_VIKING, CH0, STRM0)) {
        printf("Error: adamStrmStopTone failed! \n");
        return;
}

/* Another alternative is to play tone[0] (Silence) to stop a tone play */
toneSeq_t toneList = {
        . numOfTone = 1,
        . toneIdSeq = {0}
};

if (EXEC_SUCCESS != adamStrmPlayTone(DSP_VIKING, CH0, STRM0, &toneList, 1)) {
        printf("Error: adamStrmPlayTone failed! \n");
        return;
}
```

7 Interface Control

7.1 adamInfcQuery

Prototype:
exCode_e adamInfcQuery(infcId_e infc, infcConfig_t *pInfcConf);

Return Values:
EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
Infc	Interface ID.
pInfcConf	Pointer of a infcConfig_t instance to receive the interface configuration.

Description:
Query the interface configuration.

Example Code:

```
infcId_e infc = 0;
infcConfig_t infcConf;

if (EXEC_SUCCESS != adamInfcQuery(infc, &infcConf)) {
        printf("Error: adamInfcQuery failed! \n");
        return;
}

printf("Interface type: %d\n", infcConf.type);
printf("Line State: %d\n", infcConf.lineState);
printf("Hook State: %d\n", infcConf.hookState);
```

7.2 adamInfcConfigLine

exCode_e adamInfcConfigLine(infcId_e infc, lineState_e state);

Return Values:
EXEC_SUCCESS

EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
Infc	Interface ID.
State	Line state such as power down (LINE_DOWN), polarity reverse (LINE_ACTIVE_REV), ring (LINE_RING), etc. ** LINE_BUSY is a passive state that should only be triggered by the phone.

Description:
Change interface line state.

Example Code:

```
if (EXEC_SUCCESS != adamInfcConfigLine(INFC0, LINE_ACTIVE_FWD)) {
        printf("Error: adamInfcConfigLine failed! \n");
}
```

7.3 adamInfcConfigHook

Prototype:
exCode_e adamInfcConfigHook(infcId_e infc, hookState_e state);

Return Values:
EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
Infc	Interface ID.
state	Hook state such as on-hook (HOOK_RELEASE), off-hook (HOOK_SEIZE), flash (HOOK_FLASH), etc.

Description:
Change interface hook state. Only works for FXO interface.

Example Code:

```
if (EXEC_SUCCESS != adamInfcConfigHook(INFC2, HOOK_FLASH)) {
        printf("Error: adamInfcConfigHook failed! \n");
}
```

7.4 adamInfcConfigRing

Prototype:

exCode_e adamInfcConfigRing(infcId_e infc, ringProfile_t *pRingProf);

Return Values:

EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
Infc	Interface ID.
pRingProf	Pointer of a ringProfile_t instance which contains ring configuration including cadence, duration, caller ID, and caller ID generation timing.

Description:

Change interface ring configuration.

Example Code:

```
infcConfig_t infcConf;

infcConf.ring.dur = 5000; /* ms */
infcConf.ring.cidAt = 1; /*Ring break after first ring */
infcConf.ring.cad[0].onTime = 500; /* ms */
infcConf.ring.cad[0].offTime = 1500; /* ms */
infcConf.ring.cad[1].onTime = 0; /* ms */
infcConf.ring.cad[1].offTime = 0; /* ms */
infcConf.ring.cad[2].onTime = 0; /* ms */
infcConf.ring.cad[2].offTime = 0; /* ms */

if (EXEC_SUCCESS != adamInfcConfigRing(INFC0, &(infcConf.ring))) {
        printf("Error: adamInfcConfigRing failed! \n");
        return;
}
```

7.5 adamInfcRing

Prototype:

exCode_e adamInfcRing(infcId_e infc, uint32 dur, cid_t *pCid);

Return Values:

EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
Infc	Interface ID.
dur	Ring duration (ms)
pCid	Pointer of a cid_t instance which contains caller ID information. Pass (NULL) if no caller ID to be presented.

Description:

Start ringing on an interface.

Example Code:

```
if(EXEC_SUCCESS != adamInfcRing(INFC0, 4000, NULL)){
        printf("Error: adamInfcRing failed! \n");
}
```

7.6 adamInfcStopRing

Prototype:

exCode_e adamInfcStopRing(infcId_e infc);

Return Values:

EXEC_SUCCESS
EXEC_FAIL
INVALID_PARAM
FUNC_UNSUPPORT
DEVICE_BUSY
UNKNOWN_ERROR

Argurments:

Name	Description
Infc	Interface ID.

Description:

Stop ringing on an interface.

Example Code:

```
if(EXEC_SUCCESS != adamInfcStopRing(INFC0)){
        printf("Error: adamInfcStopRing failed! \n");
}

/* Another alternative is call adamInfcRing with dur=0 */
if(EXEC_SUCCESS != adamInfcRing(INFC0, 0, NULL)){
        printf("Error: adamInfcRing failed! \n");
}
```

8 ADAM EVENT Processing

Event is used to notify user something happened, such as a signal has been detected which match the configured patterns, or user off-hook the phone. Developer should handle the event properly and do corresponding process for each event.

Example code of event handling process:

```
event_t mEvent;

while(1) {
        usleep(10000);
        memset(&mEvent, 0, sizeof(event_t));
        if (EXEC_SUCCESS == adamPollEvent(BLOCKING, &mEvent)) {

                printf("\n[T:%010u] %s: %s", (unsigned int)mEvent.dspTick, \
                        etosEdge(mEvent.edge), etosEvent(mEvent.evtCode));

                switch(mEvent.evtCode) {
                case EVENT_CODE_HOOK:
                        switch(mEvent.context.hook.status) {
                        case HOOK_SEIZE:
                                printf("\nInterface (%d) off-hooked.\n", mEvent.infcId);
                                break;
                        case HOOK_RELEASE:
                                printf("\nInterface (%d) on-hooked.\n", mEvent.infcId);
                                break;
                        case HOOK_FLASH:
                                printf("\nInterface (%d) hook-flashed.\n", mEvent.infcId);
                                break;
                        default:
                                break;
                        }
                        break;
                case EVENT_CODE_TONE:
                        printf("\nChannel (%d) tone[%s] detected.\n", \
                                mEvent.chanId,   etosTone(mEvent.context.tone.code));
                        break;
                default:
                        break;
                }
        }
}
```

9 Appendix: Constant and Enumeration

9.1 Constant

Constant	Value
MAX8	(0xff)
MAX16	(0xffff)
MAX32	(0xffffffff)
MAX_CID_CHAR_LEN	(32)
MAX_CADENCE	(3)
MAX_TONE_FREQ	(4)
MAX_PACKET_SZ	(1024)
MAX_GAIN_AMP	(40)
MIN_GAIN_AMP	(-40)
MAX_CODEC_NUM	(CODEC_T38+1)
MASK_CODEC_G711A	(1 << CODEC_G711A)
MASK_CODEC_G711U	(1 << CODEC_G711U)
MASK_CODEC_G723	(1 << CODEC_G723)
MASK_CODEC_G722	(1 << CODEC_G722)
MASK_CODEC_G726	(1 << CODEC_G726)
MASK_CODEC_G729	(1 << CODEC_G729)
MASK_CODEC_SILCOMP	(1 << CODEC_SILCOMP)
MASK_CODEC_DTMFR	(1 << CODEC_DTMFR)
MASK_CODEC_T38	(1 << CODEC_T38)
DETECT_TONE_DTMF	(1 << 0)
DETECT_TONE_MODEM	(1 << 1)
DETECT_TONE_CPT	(1 << 2)
DETECT_CID	(1 << 3)
DETECT_DTMFR	(1 << 4)

9.2 activeState_e

```
typedef enum {
        STATE_INACTIVE,
        STATE_ACTIVE
} activeState_e;
```

9.3 blockMode_e

```
typedef enum {
        BLOCKING,
        NON_BLOCKING
} blockMode_e;
```

9.4 chanId_e

```
typedef enum {
        CH0,
        CH1,
        CH2,
        CH3,
        CH4,
        CH5,
        CH6,
        CH7
} chanId_e;
```

9.5 codec_e

```
typedef enum {
        CODEC_G711A,
        CODEC_G711U,
        CODEC_G722,
        CODEC_G723,
        CODEC_G726,
        CODEC_G729,
        CODEC_SILCOMP,
        CODEC_DTMFR,
        CODEC_T38,
        CODEC_INVALID
} codec_e;
```

9.6 dspId_e

```
typedef enum {
        DSP_VIKING
} dspId_e;
```

9.7 dtmf_e

```
typedef enum {
        DTMF_0,
        DTMF_1,
        DTMF_2,
        DTMF_3,
        DTMF_4,
        DTMF_5,
        DTMF_6,
```

```
        DTMF_7,
        DTMF_8,
        DTMF_9,
        DTMF_STAR,
        DTMF_POUND,
        DTMF_A,
        DTMF_B,
        DTMF_C,
        DTMF_D,
} dtmf_e;
```

9.8 ecTail_e

```
typedef enum {
        TAIL_16MS,
        TAIL_32MS,
        TAIL_48MS,
        TAIL_64MS,
        TAIL_128MS
} ecTail_e;
```

9.9 enableControl_e

```
typedef enum {
        CONTROL_DISABLE,
        CONTROL_ENABLE
} enableControl_e;
```

9.10 evaBool_e

```
typedef enum {
        EVA_FALSE,
        EVA_TRUE
} evaBool_e;
```

9.11 eventCode_e

```
typedef enum {
        EVENT_CODE_INVALID,
        EVENT_CODE_TONE,
        EVENT_CODE_CID,
        EVENT_CODE_LINE,
        EVENT_CODE_HOOK,
        EVENT_CODE_JB_UPDATE,
        EVENT_CODE_NON_RTP_RECVD,
```

```
        EVENT_CODE_RTCP_SEND,
        EVENT_CODE_RTCP_RECVD,
        EVENT_CODE_STREAM_UPDATE,
        EVENT_CODE_TIMER,
        EVENT_CODE_PERFORMANCE,
        EVENT_CODE_ERROR
} eventCode_e;
```

9.12 eventEdge_e

```
typedef enum {
        EDGE_ONCE,
        EDGE_BEGIN,
        EDGE_END
} eventEdge_e;
```

9.13 exCode_e

```
typedef enum {
        EXEC_SUCCESS = 1,
        EXEC_FAIL = -1,
        INVALID_PARAM = -2,
        FUNC_UNSUPPORT = -3,
        DEVICE_BUSY = -4,
        UNKNOWN_ERROR = -5
} exCode_e;
```

9.14 hookState_e

```
typedef enum {
        HOOK_FLASH,
        HOOK_RELEASE,
        HOOK_SEIZE,
        HOOK_PULSE1,
        HOOK_PULSE2,
        HOOK_PULSE3,
        HOOK_PULSE4,
        HOOK_PULSE5,
        HOOK_PULSE6,
        HOOK_PULSE7,
        HOOK_PULSE8,
        HOOK_PULSE9,
        HOOK_PULSE10,
        HOOK_PULSE11,
        HOOK_PULSE12,
```

```
        HOOK_PULSE13,
        HOOK_PULSE14,
        HOOK_PULSE15,
        HOOK_PULSE16,
        HOOK_PULSE17,
        HOOK_PULSE18,
        HOOK_PULSE19,
        HOOK_PULSE20,
        HOOK_ERROR
} hookState_e;
```

9.15 infcId_e

```
typedef enum {
        INFC0,
        INFC1,
        INFC2,
        INFC3,
        INFC4,
        INFC5,
        INFC6,
        INFC7
} infcId_e;
```

9.16 infcType_e

```
typedef enum {
        INFC_FXS,
        INFC_FXO,
        INFC_AUDIO,
        INFC_OTHER
} infcType_e;
```

9.17 ipVer_e

```
typedef enum {
        IPV4,
        IPV6
} ipVer_e;
```

9.18 lineState_e

```
typedef enum {
        LINE_DOWN,
        LINE_ACTIVE_FWD,
```

```
        LINE_ACTIVE_REV,
        LINE_RING,
        LINE_RING_PAUSE,
        LINE_BUSY,
        LINE_SLEEP,
        LINE_ERROR
} lineState_e;
```

9.19 polDir_e

```
typedef enum {
        POL_FWD,
        POL_REV
} polDir_e;
```

9.20 pTime_e

```
typedef enum {
        PTIME_10MS,
        PTIME_20MS,
        PTIME_30MS,
        PTIME_40MS,
        PTIME_50MS,
        PTIME_60MS
} pTime_e;
```

9.21 strmDir_e

```
typedef enum {
        STRM_INACTIVE,
        STRM_SENDONLY,
        STRM_RECVONLY,
        STRM_SENDRECV
} strmDir_e;
```

9.22 strmId_e

```
typedef enum {
        STRM0,
        STRM1,
        STRM2,
        STRM3
} strmId_e;
```

9.23 toneCode_e

```
typedef enum {
        TONE_DTMF_1 = 1,
        TONE_DTMF_2,
        TONE_DTMF_3,
        TONE_DTMF_4,
        TONE_DTMF_5,
        TONE_DTMF_6,
        TONE_DTMF_7,
        TONE_DTMF_8,
        TONE_DTMF_9,
        TONE_DTMF_0,
        TONE_DTMF_STAR,
        TONE_DTMF_POUND,
        TONE_DTMF_A,
        TONE_DTMF_B,
        TONE_DTMF_C,
        TONE_DTMF_D,
        TONE_DIAL,
        TONE_BUSY,
        TONE_REORDER,
        TONE_HAWLER,
        TONE_RINGBACK,
        TONE_SAS,
        TONE_SIT,
        TONE_CNG,
        TONE_CED,
        TONE_ANS,
        TONE_ANSAM,
        TONE_V21PREAMBLE
} toneCode_e;
```

9.24 toneType_e

```
typedef enum {
        TONE_REGULAR,
        TONE_MODULATE
} toneType_e;
```

10 Appendix: Data Structure

10.1 cadence_t

```
typedef struct {
        uint16 onTime;
        uint16 offTime;
} cadence_t;
```

Description:
Cadence holds the time information a signal on-off duration.

Attribute	Type	Valid Value Range	Description
onTime	uint16	0 ~ 65535(ms)	Time of signal on.
offTime	uint16	0 ~ 65535(ms)	Time of signal off.

10.2 chanConfig_t

```
typedef struct {
        uint16              detectMask;
        int8                ampTx;
        int8                ampRx;
        enableControl_e     ecEnable;
}chanConfig_t;
```

Description:
Channel configuration holds the configuration information per channel, including the signal detector controller on/off, Tx/RX gain, and echo canceller on/off.

Attribute	Type	Valid Value Range	Description
detectMask	uint16	DETECT_TONE_DTMF DETECT_TONE_MODEM DETECT_TONE_CPT DETECT_CID DETECT_DTMFR	Bit mask configuration to enable/disable DSP detectors.
ampTx	int8	-40 ~ 40 (0.5db)	Adjust Tx (output raw PCM) gain to the interface within +/-20db range, step by 0.5db.
ampRx	int8	-40 ~ 40 (0.5db)	Adjust Rx (input raw PCM) gain from the interface within +/-20db range, step by 0.5db.
ecEnable	enableControl_e	CONTROL_DISABLE CONTROL_ENABLE	Enable or disable echo cancellation.

10.3 cid_t

```
typedef struct {
        char number[MAX_CID_CHAR_LEN];
        char name[MAX_CID_CHAR_LEN];
        char dateTime[MAX_CID_CHAR_LEN];
}cid_t;
```

Description:

Caller ID holds caller ID information such as number, user name, and date-time.

Attribute	Type	Valid Value Range	Description
Number	char[]		Caller ID display number.
Name	char[]		Caller ID display name.
dateTime	char[]		Caller ID display date and time.

** NOTE: For some telephones do not support display name or date-time, they might not be able to show number when name and/or dateTime field is presented.

10.4 dspFeature_t

```
typedef struct {
        uint16  dspId;
        uint8   numOfChan;
        uint8   strmsPerChan;
        uint32  codecSupport;
        uint32  ptimeSupport;
        uint8   rtpRedundancy;
        uint8   maxEcTailLength;
}dspFeature_t;
```

Description:

DSP feature is used to acquire DSP capability information.

Attribute	Type	Valid Value Range	Description
dspId	uint16	Read-Only	Provide DSP ID information.
numOfChan	uint8	Read-Only	Number of channel supports on the DSP.
strmsPerChan	uint8	Read-Only	Number of streams supports on each channel.
codecSupport	uint32	Read-Only	CODECs supports on the DSP. Use MASK_CODEC_X to check.
ptimeSupport	uint32	Read-Only	PTime supports on the DSP.
rtpRedundancy	uint8	Read-Only	Check if DSP supports RTP redundancy (RFC2198).
maxEcTailLength	uint8	Read-Only	Maximum echo cancellation tail length

			that DSP supports.

10.5 eventContext_u

```
typedef union {
        struct{
                lineState_e     status;
                int             ringCount;
        }line; /* EVENT_CODE_LINE */

        struct{
                hookState_e     status;
                int             pulseCount;
        }hook; /*EVENT_CODE_HOOK*/

        struct{
                uint32  dspExecTimes;
                uint32  averageMhz;
        }performanceIdx; /*EVENT_CODE_PERFORMANCE*/

        struct{
                int8    number[MAX_CID_CHAR_LEN];
                int8    name[MAX_CID_CHAR_LEN];
                int8    dateTime[MAX_CID_CHAR_LEN];
        }cidData;/*EVENT_CODE_CID_DETECTED*/

        struct{
                toneCode_e      code;
        }tone; /*EVENT_CODE_TONE_DETECTED*/

        struct{
                uint8   streamId;
                uint32  total;
                uint32  drop;
                uint32  plc;
                uint32  jbSize;
                uint32  avgJitter;
        }jb; /*EVENT_CODE_JB_UPDATE*/

        struct{
                netAddr_t       srcAddr;
                netAddr_t       dstAddr;
                uint8           payload[MAX_PACKET_SZ];
        }packet; /*EVENT_CODE_NON_RTP_RECVD*/
}eventContext_u;
```

Description:

Event context provide the detail information/data for particular events.

Attribute	Type	Valid Value Range	Description
line.status	lineState_e	LINE_DOWN LINE_ACTIVE_FWD LINE_ACTIVE_REV LINE_RING LINE_RING_PAUSE LINE_BUSY LINE_SLEEP LINE_ERROR	Interface line state.
line.ringCount	int		Ring times counter for RING event.
hook.status	hookState_e	HOOK_FLASH HOOK_RELEASE HOOK_SEIZE HOOK_PULSE1 HOOK_PULSE2 HOOK_PULSE3 HOOK_PULSE4 HOOK_PULSE5 HOOK_PULSE6 HOOK_PULSE7 HOOK_PULSE8 HOOK_PULSE9 HOOK_PULSE10 HOOK_PULSE11 HOOK_PULSE12 HOOK_PULSE13 HOOK_PULSE14 HOOK_PULSE15 HOOK_PULSE16 HOOK_PULSE17 HOOK_PULSE18 HOOK_PULSE19 HOOK_PULSE20 HOOK_ERROR	Interface hook state.
hook.pulseCount	int		Pulse time counter for PULSE event. ** Not used now.
performanceIdx.dspExecTimes	uint32		Not used now.
performanceIdx.averageMhz	uint32		Not used now.
cidData.number	int8		CID display number information.
cidData.name	int8		CID display name

Attribute	Type	Valid Value Range	Description
			information.
cidData.dateTime	int8		CID date-time information.
tone.code	toneCode_e		Tone code information.
jb.streamId	uint8		Stream ID information.
jb.total	uint32		Number of total packets.
jb.drop	uint32		Number of dropped packets.
jb.plc	uint32		Number of compensated packets.
jb.jbSize	uint32		Current jitter buffer size.
jb.avgJitter	uint32		Average jitter.
packet.srcAddr	netAddr_t		Not used now.
packet.dstAddr	netAddr_t		Not used now.
packet.payload	uint8[]		Not used now.

10.6 event_t

```
typedef struct {
        eventEdge_e         edge;
        eventCode_e         evtCode;
        uint32              dspTick;
        infcId_e            infcId;
        chanId_e            chanId;
        eventContext_u      context;
} event_t;
```

Description:

Event report provide event category, event time, interface/channel ID, and event context information.

Attribute	Type	Valid Value Range	Description
edge	eventEdge_e	EDGE_ONCE EDGE_BEGIN EDGE_END	Edge information of an event report. For some events that will last for a while, such as tone or ring, etc., the event reports once at the beginning with EDGE_BEGIN and again at the end with EDGE_END. User may use dspTick of both events to get the duration of the event last. For other events represented a state change, such as on-hook, off-hook,

Attribute	Type	Valid Value Range	Description
			EDGE_ONCE is used.
evtCode	eventCode_e	EVENT_CODE_INVALID EVENT_CODE_TONE EVENT_CODE_CID EVENT_CODE_LINE EVENT_CODE_HOOK EVENT_CODE_JB_UPDATE EVENT_CODE_NON_RTP_RECVD EVENT_CODE_RTCP_SEND EVENT_CODE_RTCP_RECVD EVENT_CODE_STREAM_UPDATE EVENT_CODE_TIMER EVENT_CODE_PERFORMANCE EVENT_CODE_ERROR	Event message categorization information.
dspTick	uint32	0x0~0xFFFFFFFF(ms)	DSP (or CPU) tick to indicate the time information of an event.
infcId	infcId_e		Interface ID, presented when an event is interface related.
chanId	chanId_e		Channel ID, presented when an event is channel related.
Context	eventContext_u		Event context contains detail information of the event if applicable.

10.7 infcConfig_t

```
typedef struct {
        infcType_e      type;
        lineState_e     lineState;
        hookState_e     hookState;
        polDir_e        pol;

        ringProfile_t   ring;
}infcConfig_t;
```

Description:

Interface configuration provide the line, hook, and/or ring configuration information of an interface.

Attribute	Type	Valid Value Range	Description
Type	infcType_e	INFC_FXS INFC_FXO INFC_AUDIO	Read-only, device type of the interface.

		INFC_OTHER	
lineState	lineState_e	LINE_DOWN LINE_ACTIVE_FWD LINE_ACTIVE_REV LINE_RING LINE_RING_PAUSE LINE_BUSY LINE_SLEEP LINE_ERROR	Interface line state. Read-Writable for FXS interface and Read-only for FXO interface.
hookState	hookState_e	HOOK_FLASH HOOK_RELEASE HOOK_SEIZE HOOK_PULSE1 HOOK_PULSE2 HOOK_PULSE3 HOOK_PULSE4 HOOK_PULSE5 HOOK_PULSE6 HOOK_PULSE7 HOOK_PULSE8 HOOK_PULSE9 HOOK_PULSE10 HOOK_PULSE11 HOOK_PULSE12 HOOK_PULSE13 HOOK_PULSE14 HOOK_PULSE15 HOOK_PULSE16 HOOK_PULSE17 HOOK_PULSE18 HOOK_PULSE19 HOOK_PULSE20 HOOK_ERROR	Interface hook state. Read-Writable for FXO interface and Read-only for FXS interface. ** HOOK_PULSEXX is not used now.
Pol	polDir_e	POL_FWD POL_REV	Line power feed polarity direction.
Ring	ringProfile_t		Ring configuration per interface. Only for FXS.

10.8 netAddr_t

```
typedef struct {
        uint32          addrV4;
        uint16          addrV6[8];
        ipVer_e         ver;
        uint16          port;
}netAddr_t;
```

Description:
Network address provide the IP address and data port information.

Attribute	Type	Valid Value Range	Description
addrV4	uint32		IPv4 address
addV6	uint16[]		IPv6 address
Ver	ipVer_e	IPV4 IPV6	Network address type
Port	uint16	0 ~ 65535	RTP port

10.9 ringProfile_t

```
typedef struct {
        cadence_t       cad[MAX_CADENCE];
        uint32          dur;
        cid_t           cid;
        uint8           cidAt;
} ringProfile_t;
```

Description:
Ring profile provide the configuration of a ring.

Attribute	Type	Valid Value Range	Description
Cad	cadence_t[]		Ring cadences.
Dur	uint32	0x0~0xFFFFFFFF(ms)	Ring duration.
cid	cid_t		Caller ID information.
cidAt	uint8	0~255	N-th ring-breaks for CID transmission.

10.10 session_t

```
typedef struct {
        ipVer_e         ver;
        netAddr_t       srcAddr;
        netAddr_t       dstAddr;
        uint8           encrypt; /*T/F*/ /*reserved for user to select the encypt type*/
        uint32          dur; /*session time*/ /*reserved for update the session duration*/
}session_t;
```

Description:
Session holds the source and destination network address and other session configuration information.

Attribute	Type	Valid Value Range	Description

Attribute	Type	Valid Value Range	Description
Ver	ipVer_e	IPV4 IPV6	IP version of the session.
srcAddr	netAddr_t		Source address of the session.
dstAddr	netAddr_t		Destination address of the session.
Encrypt	uint8		Not used now.
Dur	uint32		Not used now.

10.11 strmAttr_t

```
typedef struct {
        codec_e         payloadSelect;
        pTime_e         ptimeSelect;
        enableControl_e dtmfRelay;
        enableControl_e silenceComp;
        strmDir_e       direction;
        uint32          jbUpdateTime;
}strmAttr_t;
```

Description:

Stream attribute holds the attribute configurations of a stream.

Attribute	Type	Valid Value Range	Description
payloadSelect	codec_e	CODEC_G711A CODEC_G711U CODEC_G722 CODEC_G723 CODEC_G726 CODEC_G729 CODEC_SILCOMP CODEC_DTMFR CODEC_T38	CODEC used for streaming.
ptimeSelect	pTime_e	PTIME_10MS PTIME_20MS PTIME_30MS PTIME_40MS PTIME_50MS PTIME_60MS	Stream P-time (P-rate) configuration.
dtmfRelay	enableControl_e	CONTROL_DISABLE CONTROL_ENABLE	Enable/disable DTMF relay (RFC2833/4733)
silenceComp	enableControl_e	CONTROL_DISABLE CONTROL_ENABLE	Enable/disable silence compression (CN).
Direction	strmDir_e	STRM_INACTIVE STRM_SENDONLY STRM_RECVONLY	Stream transmission direction.

Attribute	Type	Valid Value Range	Description
		STRM_SENDRECV	
jbUpdateTime	uint32	0x0 ~ 0xFFFFFFFF(ms)	Not used now.

10.12 strmConfig_t

```
typedef struct {
        session_t           session;
        strmAttr_t          strmAttr;
        uint8               payloadType[MAX_CODEC_NUM];
}strmConfig_t;
```

Description:

Stream configuration holds the session, stream attribute, and payload type number information.

Attribute	Type	Valid Value Range	Description
Session	session_t		Stream session information.
strmAttr	strmAttr_t		Stream attribute configuration.
payloadType	uint8[]	0 ~ 127	Payload type number for each CODEC.

10.13 tone_t

```
typedef struct {
        toneType_e toneType;

   struct{
                uint16   toneFreq[MAX_TONE_FREQ];
                int16    tonePwr[MAX_TONE_FREQ];
        }regular;

        struct{
                int16    baseFreq;
                int16    modFreq;
                int16    modPwr;
                int16    modDepth;
        }modulate;

        int16 makeTime[MAX_CADENCE];
        int16 breakTime[MAX_CADENCE];
        int16 repeat[MAX_CADENCE];
}tone_t;
```

Description:

Tone is used to configure the frequency, power, cadence, repeat, etc., for tone generation.

Attribute	Type	Valid Value Range	Description
toneType	toneType_e	TONE_REGULAR TONE_MODULATE	Type of Tone
regular.toneFreq	uint16[]	0 ~ 4000(Hz)	Up to 4 tone frequency can be configured for generation.
regular.tonePwr	int16[]	-40 ~ 0(db)	Tone power for each tone frequency.
modulate.baseFreq	int16	0 ~ 4000(Hz)	Base frequency for amplitude modulation.
modulate.modFreq	int16	0 ~ 4000(Hz)	Amplitude modulation frequency.
modulate.modPwr	int16	-40 ~ 0(db)	Modulation power.
modulate.modDepth	int16	0 ~ 65535	Modulation depth.
makeTime	int16[]	0 ~ 65535(ms)	Up to 3 cadence configuration for tone. Time of signal on for each cadence.
breakTime	int16[]	0 ~ 65535(ms)	Up to 3 cadence configuration for tone. Time of signal off for each cadence.
Repeat	int16[]	0 ~ 65535(time)	Up to 3 cadence configuration for tone. Time of repeat for each cadence.

10.14 toneSeq_t

```
typedef struct {
        uint8 *toneIdSeq;
        uint8 numOfTone;
}toneSeq_t;
```

Description:

Tone sequence holds a series of tone to be generated.

Attribute	Type	Valid Value Range	Description
toneIdSeq	uint8*		Array of Tone ID to be played in sequence.
numOfTone	uint8	0 ~ 255	Number of Tone in the array.

11 Appendix: Default Call Progress Tone Profile

Call Progress Tone (CPT) profile configures the signal patterns to detect whether a signal contains a signal match the configured patterns and report a detection event. It is used on FXO interface to understand the line status by listening to distinctive tones.

Tone	Frequency	Cadence
DAIL	350Hz@-21db 440Hz@-21db	Cadence[0] = 1000(ms)/on, 0(ms)/off,
BUSY	480Hz@-21db 620Hz@-21db	Cadence[0] = 500(ms)/on, 500(ms)/off
RING_BACK	440Hz@-18db 480Hz@-18db	Cadence[0] = 1000(ms)/on, 3000(ms)/off
REORDER	480Hz@-18db 620Hz@-18db	Cadence[0] = 250(ms)/on, 250(ms)/off
HOWLER	1800Hz@0db 2500Hz@0db	Cadence[0] = 100(ms)/on, 100(ms)/off
SAS	400Hz@-15db	Cadence[0] = 150(ms)/on, 0(ms)/off
CNG	1300Hz@-15db	Cadence[0] = 500(ms)/on, 0(ms)/off
CED	2100Hz@-15db	Cadence[0] = 500(ms)/on, 0(ms)/off