# Contents

## 1.0 Introduction
- General ......................................................................................................................... 7
- Scope ................................................................................................................................. 7
- Audience .......................................................................................................................... 7
- Acronyms .......................................................................................................................... 8

## 2.0 Architectural Overview .............................................................................................. 8

## 3.0 Media Processing Resource Components .................................................................... 9

- 3.1 Network Endpoint Resource Component .................................................................. 10
  - Media Processing Functions ......................................................................................... 10
  - Parameters ..................................................................................................................... 11
  - Events ........................................................................................................................... 12
- 3.2 Decoder Resource Component .................................................................................. 13
  - Media Processing Functions ......................................................................................... 13
  - Resource-Specific Control Messages ........................................................................... 13
  - Parameters ..................................................................................................................... 13
  - Events ........................................................................................................................... 14
- 3.3 Encoder Resource Component .................................................................................. 14
  - Media Processing Functions ......................................................................................... 14
  - Resource-Specific Control Messages ........................................................................... 14
  - Parameters ..................................................................................................................... 15
  - Events ........................................................................................................................... 15
- 3.4 Tone Generation Resource Component ................................................................... 15
  - Media Processing Functions ......................................................................................... 16
  - Resource-Specific Control Messages ........................................................................... 16
  - Parameters ..................................................................................................................... 16
  - Events ........................................................................................................................... 16
- 3.5 Tone Detection Resource Component ..................................................................... 16
  - Media Processing Functions ......................................................................................... 16
  - Resource-Specific Control Messages ........................................................................... 17
  - Parameters ..................................................................................................................... 17
  - Events ........................................................................................................................... 17
- 3.6 Audio Player Resource Component ........................................................................ 18
  - Media Processing Functions ......................................................................................... 18
  - Resource-Specific Control Messages ........................................................................... 18
  - Parameters ..................................................................................................................... 18
  - Events ........................................................................................................................... 18
- 3.7 Audio Mixer Resource Component ........................................................................ 18
  - Media Processing Functions ......................................................................................... 18
  - Resource-Specific Control Messages ........................................................................... 18
  - Parameters ..................................................................................................................... 19
  - Events ........................................................................................................................... 19
- 3.8 Message Agent Resource Component .................................................................... 19
  - Media Processing Functions ......................................................................................... 19
  - Resource-Specific Control Messages ........................................................................... 19
  - Parameters ..................................................................................................................... 19
  - Events ........................................................................................................................... 19

## 4.0 Message Format and Delivery Mechanism .................................................................. 19

- 4.1 Message Functions .................................................................................................... 20
- 4.2 Message Header Format ............................................................................................. 21
- 4.3 Message Type List ....................................................................................................... 21
5.0 Common Control Message

5.1 Reset Message
5.2 Start Message
5.3 Stop Message
5.4 Ping Message
5.5 Set Parameter Message
5.6 Set Multiple-Parameter Message
5.7 Get Parameter Message
5.8 Get Parameter Acknowledge Message
5.9 Get All Parameters Message
5.10 Get All Parameters Acknowledge Message
5.11 General Acknowledge Message
5.12 Error Message
5.13 Event Message

6.0 Resource-Specific Control Messages

6.1 CODEC Start Message
6.2 CODEC Stop Acknowledgement Message
6.3 Tone Generator Play Message
6.4 Tone Generator Play FSK Message
6.5 Tone Generator Play Completed Message
6.6 Tone Detector Receive Digit Message
6.7 Tone Detector Receive Completed Message
6.8 Tone Detector Receive FSK Message
6.9 Tone Detector FSK Receive Completed Message
6.10 Player Start Message
6.11 Player Play Completed Message
6.12 Get Jitter Buffer Statistics Message
6.13 Complete Message of Getting Jitter Buffer Statistics
6.14 Get ERLE Statistics Message
6.15 Complete Message of Getting ERLE statistics
6.16 Get MFPP Statistics Message
6.17 Complete Message of Getting MFPP statistics

7.0 Packet Data Interface

7.1 Packet Formats
7.2 Packet Delivery Mechanism

8.0 Configuration and Initialization

8.1 System Configuration with HSS Interface
8.1.1 Description
8.2 Adding Tones to Tone Generator
8.2.1 Description
8.3 Change the DTMF Tone Parameters
8.3.1 Description
8.4 Adding Tones to Tone Detector
8.4.1 Description
8.5 Amplitude Check in Tone Detection
8.5.1 Description
8.6 Getting DSP Resource Configuration and Routing Information
8.6.1 Description

9.0 Complementary Functions

9.1 Direct Parameter Access
9.2 Direct Plug-in Parameter Access
9.3 Flash Hook Detection
9.4 Cache Prompt Registration ................................. 48
9.5 Get Version Number ........................................... 49
9.6 External PCM Interface Synchronization ............ 49

10.0 Constant Data .................................................... 49
10.1 Error Codes ......................................................... 49
10.2 Event Codes ......................................................... 50
10.3 Tone IDs .............................................................. 50
10.4 Other Constants .................................................. 53

A : HSS Driver APIs ................................................. 55
B : SLIC Driver APIs ................................................. 55

Figures
1  Architecture of Intel® Infrastructure DSP Solution ......................... 9
2  Resource Component Identifiers ........................................ 10
## Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Revision</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>January 2007</td>
<td>001</td>
<td>Initial release,</td>
</tr>
<tr>
<td>June 2007</td>
<td>002</td>
<td>Added support for Intel® IXP43X product line of network processors.</td>
</tr>
</tbody>
</table>


1.0 **Introduction**

The Intel® Infrastructure DSP Solution Version 1.1 is a software module that provides basic voice and signal processing functionality for voice-over-Internet-protocol (VoIP) on the Intel® IXP4XX product line of network processors (that is, the Intel® IXP42X product line of network processors, the Intel® IXP43X product line of network processors and the Intel® IXP46X product line of network processors). It can be viewed as a completed media processing layer with control and data interfaces as its API.

This document defines the API specifications.

*Note:* The Intel® Infrastructure DSP Solution Version 1.1 is referred to as the “DSP software” throughout this document.

1.1 **General**

The DSP software is a software module for media processing, targeted for next generation IADs such as Consumer Premises Equipment (CPE), specifically, to perform audio encoding/decoding, echo cancellation, tone processing and jitter control, and other functionalities, as required in any IP media gateway or real-time media streaming functionalities.

This document is intended to describe the control and data interfaces for a third party developer to incorporate the module into a media gateway. It provides sufficient details of the interfaces so that the user can fully configure and control the operations and services.

It also describes the data interface, message format, message and data delivery mechanisms.

1.2 **Scope**

The interface of the DSP software is a set of functions, macros, message and packet formats that determines how the applications access the media processing resource components.

1.3 **Audience**

This document is intended for the following audiences:

- Firmware engineers who are responsible for the development of DSP Resources
- Third party software engineers who are building a gateway or server application
- System architects and engineers
- Project development managers
1.4 **Acronyms**

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AEC</td>
<td>Acoustic Echo Canceller (Note: AEC component not supported)</td>
</tr>
<tr>
<td>AGC</td>
<td>Automatic Gain Control for voice data towards IP network</td>
</tr>
<tr>
<td>ALC</td>
<td>Automatic Level Control</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CPE</td>
<td>Consumer Premise Equipment</td>
</tr>
<tr>
<td>EC</td>
<td>Echo Cancellation</td>
</tr>
<tr>
<td>FSK</td>
<td>Frequency Shift Keying</td>
</tr>
<tr>
<td>FXO</td>
<td>Foreign Exchange Office</td>
</tr>
<tr>
<td>FXS</td>
<td>Foreign Exchange Subscriber</td>
</tr>
<tr>
<td>IAD</td>
<td>Integrated Access Device</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>ISR</td>
<td>Interrupt Service Routine</td>
</tr>
<tr>
<td>MPR</td>
<td>Media Processing Resource</td>
</tr>
<tr>
<td>NLP</td>
<td>Non-linear Processing (for EC)</td>
</tr>
<tr>
<td>PCM</td>
<td>Pulse Code Modulation</td>
</tr>
<tr>
<td>RTP</td>
<td>Real-time Transport Protocol</td>
</tr>
<tr>
<td>SLIC</td>
<td>Subscriber Line Interface Circuit</td>
</tr>
<tr>
<td>SP</td>
<td>Signal Processing</td>
</tr>
<tr>
<td>TD</td>
<td>Tone Detector</td>
</tr>
<tr>
<td>TDM</td>
<td>Time Division Multiplexing</td>
</tr>
<tr>
<td>TG</td>
<td>Tone Generator</td>
</tr>
<tr>
<td>USCI</td>
<td>Unified Speech codec Interface</td>
</tr>
<tr>
<td>VAD</td>
<td>Voice Activity Detection</td>
</tr>
</tbody>
</table>

2.0 **Architectural Overview**

The DSP software is implemented as an independent module having its own tasks and runtime environment. The software architecture comprises the following two-layer hierarchy:

1. A control layer that provides the control interface and control logic, and
2. A data processing layer where the media data streams are processed by appropriate algorithms.

**Figure 1** shows the architecture of the module. In this architecture, a group of Media Processing Resource (MPR) components form a channel for full duplex media processing. They are the addressable entities that can be controlled individually by the applications.

Software developers have the flexibility to use Intel provided Media processing algorithmic modules or plug-in external algorithmic modules. These modules are static libraries and can be plugged into the framework during the build process. The Plug-in interface is the interface between the framework and the plug-in modules. The plug-in interface supports the Unified Speech Codec Interface (USCI).
As shown in Figure 2, the addressable control entities of the DSP software are Media Processing Resource (MPR) components. There are seven resource components, working together to provide all the media processing needed by a gateway. Each resource component has a unique identifier as shown in Figure 2. In the following, we refer to each media processing entity either as a resource or a resource component.

Note: T.38 MPR and AEC component in Network Endpoint MPR are not supported in the Intel® Infrastructure DSP Solution Version 1.1.
Each resource contains a particular set of algorithms to perform a specific set of media-
processing functions. For example, the Network Endpoint resource comprises echo
cancellation, high pass filter and PCM A-law or µ-law conversion algorithms to perform
TDM front-end processing. Thus, each resource has a unique set of parameters
associated with the particular set of algorithms it contains.

Note: Acoustic Echo Canceller (AEC) component in Network Endpoint MPR is not supported in
Intel® Infrastructure DSP Solution Version 1.1. All references to Echo Canceller (EC) in
this document refer to Line Echo Canceller.

The control information for these resource components is communicated through the
messages defined in this document. Some messages are common to all the resources
while others are unique only to a particular resource.

The following sections describe each resource in terms of their identifiers, media
processing functions, parameters, and control messages. The resource parameters can
be read or modified by the messages or direct function calls. Some of the parameters
can only be set through the messages because they can only be updated by the
internal control task.

3.1 Network Endpoint Resource Component

Resource Type: XMPR_NET

3.1.1 Media Processing Functions

- A-law or µ-law compression and decompression
- High pass Filter
- Echo Cancellation (EC)
- Supplementary functions (timer and flash hook detection)

Resource-Specific Control Messages: None

Note: Acoustic Echo Canceller (AEC) component in Network Endpoint MPR is not supported in
Intel® Infrastructure DSP Solution Version 1.1. All the references of Echo Canceller in
this document refer to Line Echo Canceller.
### 3.1.2 Parameters

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description, Values</th>
<th>Attr.</th>
<th>Direct Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>XPARMID_RES_STATE</td>
<td>Current state (0: idle, 1: active)</td>
<td>R</td>
<td>N</td>
</tr>
<tr>
<td>XPARMID_NET_LP_STREAM</td>
<td>The L-Port stream ID. Default: the stream assigned to the IP termination’s T-Port of the same channel if exist, otherwise -1.</td>
<td>R/W</td>
<td>N</td>
</tr>
<tr>
<td>XPARMID_NET_LAW</td>
<td>PCM data format on HSS TDM bus. XPARM_NET_ALAW or XPARM_NET_MULAW. Default: XPARM_NET_MULAW</td>
<td>R/W</td>
<td>N</td>
</tr>
<tr>
<td>XPARMID_NET_ECENABLE</td>
<td>EC enabling flag, XPARM_ON or XPARM_OFF. Default: XPARM_ON</td>
<td>R/W</td>
<td>Y</td>
</tr>
<tr>
<td>XPARMID_NET_ECTAIL</td>
<td>EC tail length (2, 4, 6, 8, ... in 1 ms unit, Max 128 in narrowband mode and 64 in wideband mode). Default: 6. The resource must be reset after setting the parameter.</td>
<td>R/W</td>
<td>N</td>
</tr>
<tr>
<td>XPARMID_NET_ECNLP</td>
<td>EC NLP and suppress flag, XPARM_OFF, XPARM_EC_NLP_ON. (Note: XPARM_EC_NLP_SUP_ON Option is not supported in this version.) Default: XPARM_OFF</td>
<td>R/W</td>
<td>N</td>
</tr>
<tr>
<td>XPARMID_NET_ECFREEZE</td>
<td>EC freezing flag, XPARM_ON (freeze) or XPARM_OFF (adaptive). Typically, freeze is used only in debug situations. Default: XPARM_OFF</td>
<td>R/W</td>
<td>N</td>
</tr>
<tr>
<td>XPARMID_NET_DELAYCOMP</td>
<td>EC delay compensation (0 ~ 240 in 0.125-ms units). Default: 20 (or 2.5 ms delay compensation)</td>
<td>R/W</td>
<td>Y</td>
</tr>
<tr>
<td>XPARMID_NET_FLASH_HK</td>
<td>The window of flash hook detection (in 10-ms units) Default: 100</td>
<td>R/W</td>
<td>Y</td>
</tr>
<tr>
<td>XPARMID_NET_TIMER</td>
<td>Timer counter (in 10 ms unit). This timer can be used for timing that is synchronized to the TDM clock. Range: 0 to 65535 Default: 0</td>
<td>R/W</td>
<td>Y</td>
</tr>
<tr>
<td>XPARMID_NET_GAIN_RX</td>
<td>Input gain of HSS interface (+15 ~ -40 in 1-dB units) Default: 0</td>
<td>R/W</td>
<td>N</td>
</tr>
<tr>
<td>XPARMID_NET_GAIN_TX</td>
<td>Output gain of HSS interface (+15 ~ -40 in 1-dB units) Default: 0</td>
<td>R/W</td>
<td>N</td>
</tr>
<tr>
<td>XPARMID_NET_HSS_BYPASS</td>
<td>TDM short bypass flag, XPARM_ON or XPARM_OFF. The low latency connection made within NPE between the corresponding time slots if enabled. Do not enable it in wideband mode. Default: XPARM_OFF</td>
<td>R/W</td>
<td>N</td>
</tr>
<tr>
<td>XPARMID_NET_AECENABLE</td>
<td>AEC component is not supported.</td>
<td>not supported</td>
<td>not supported</td>
</tr>
<tr>
<td>XPARMID_NET_AECALG</td>
<td>AEC component is not supported.</td>
<td>not supported</td>
<td>not supported</td>
</tr>
<tr>
<td>XPARMID_NET_AECTAIL</td>
<td>AEC component is not supported.</td>
<td>not supported</td>
<td>not supported</td>
</tr>
<tr>
<td>XPARMID_NET_AECNLP</td>
<td>AEC component is not supported.</td>
<td>not supported</td>
<td>not supported</td>
</tr>
<tr>
<td>XPARMID_NET_AECFREEZE</td>
<td>AEC component is not supported.</td>
<td>not supported</td>
<td>not supported</td>
</tr>
<tr>
<td>XPARMID_NET_AECHOWL</td>
<td>AEC component is not supported.</td>
<td>not supported</td>
<td>not supported</td>
</tr>
<tr>
<td>XPARMID_NET_AECTD</td>
<td>AEC component is not supported.</td>
<td>not supported</td>
<td>not supported</td>
</tr>
</tbody>
</table>
### 3.1.3 Events

- **XEVT_NET_HOOK_STATE** — Hook state change detected.
- **XEVT_NET_TIMER** — Timer expired.
- **XEVT_NET_SIN_PHASEREV_YES** — Event reported on detection of phase reversal in 2100 Hz tone on Tx path.
- **XEVT_NET_RIN_PHASEREV_YES** — Event reported on detection of phase reversal in 2100 Hz tone on Rx path.

Detection of phase reversal on either Tx or Rx path triggers the detection of continuous silence on both paths.

- **XEVT_NET_SIN_ECNABLED** — Event reported on detection of continuous silence on Tx path for duration of X ms.
- **XEVT_NET_RIN_ECNABLED** — Event reported on detection of continuous silence on Rx path for duration of X ms.

This is applicable for continuous silence at the end of a fax session. Detection of either **XEVT_NET_SIN_PHASEREV_YES** or **XEVT_NET_RIN_PHASEREV_YES** is required prior to the detection of silence event. Silence duration X is configurable using **XPARMID_NET_ECTDSILENCETIME** API.
3.2 Decoder Resource Component

Resource Type: XMPR_DEC

3.2.1 Media Processing Functions

- Decoding
- Automatic level control and/or volume control
- Comfort noise generation
- Jitter compensation

3.2.2 Resource-Specific Control Messages

- XMSG_CODER_START (inbound)
- XMSG_CODER_STOP_ACK (outbound)

3.2.3 Parameters

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description and Values</th>
<th>Attr.</th>
<th>Direct Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>XPARMID_RES_STATE</td>
<td>Current state (0: idle, 1: active)</td>
<td>R</td>
<td>N</td>
</tr>
<tr>
<td>XPARMID_DEC_VOL</td>
<td>Decoder volume adjustment; +15 ~ -40 in 1-dB units. Default: 0 (Set to -99 to mute)</td>
<td>R/W</td>
<td>N</td>
</tr>
<tr>
<td>XPARMID_DEC_ALC</td>
<td>ALC enable flag. XPARM_ON or XPARM_OFF. Default: XPARM_ON</td>
<td>R/W</td>
<td>N</td>
</tr>
<tr>
<td>XPARMID_DEC_CNG</td>
<td>CNG enable flag. XPARM_ON or XPARM_OFF. Default: XPARM_OFF</td>
<td>R/W</td>
<td>Y</td>
</tr>
</tbody>
</table>
| XPARMID_DEC_CTYPE     | Coder type. Currently supported types are XCODER_TYPE_G711MU_10MS, XCODER_TYPE_G711A_10MS,
                          |                                          | R/W   | N            |
| XPARMID_DEC_EVT_PKT    | Report bad and lost packet, caused by the jitter buffer unable to provide packets to the decoder. XPARM_ON or XPARM_OFF. Default: XPARM_OFF | R/W   | Y            |
| XPARMID_DEC_EVT_PKTCHNG | Report RTP payload type change. XPARM_ON or XPARM_OFF. Default: XPARM_OFF            | R/W   | Y            |
| XPARMID_DEC_AUTOSW    | Auto-Switch mask bits. This specifies which coder types are allowed to be auto-switched based on input RTP payload type. Default: XPARM_DEC_AUTOSW_ALL | R/W   | Y            |
| XPARMID_DEC_JB_MAXDLY | Jitter buffer maximum delay (10 ~ 500 in 1-ms units). Default: 200.                  | R/W   | N            |
| XPARMID_DEC_JB_PLR    | Jitter buffer packet loss rate in 0.1% units. Default: 1                              | R/W   | N            |
### 3.2.4 Events

- **XEVT_LOST_PACKET** – Bad or lost packet.
- **XEVT_DEC_PACKET_CHNG** – RTP payload type changed.

### 3.3 Encoder Resource Component

**Resource Type:** XPARR_ENC

#### 3.3.1 Media Processing Functions

- Encoding
- Automatic Gain Control
- Voice Activity Detection

#### 3.3.2 Resource-Specific Control Messages

- **XMSG_CODER_START** *(inbound)*
- **XMSG_CODER_STOP_ACK** *(outbound)*

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description and Values</th>
<th>Attr.</th>
<th>Direct Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>XPARMID_DEC_JB_MODE</td>
<td>Jitter Buffer Mode. 0 indicates fixed delays using static Jitter Buffer, 1 indicates dynamic adjustment using the histogram algorithm, 2 indicates dynamic adjustment using the RFC3550 method. Default: 1.</td>
<td>R/W</td>
<td>N</td>
</tr>
<tr>
<td>XPARMID_DEC_G726_40_RTP_PLD</td>
<td>RTP payload type for G.726 40-Kbps coder. The payload type is negotiated and set by the call stack. The range of values is 96 to 127. Default: 96.</td>
<td>R/W</td>
<td>Y</td>
</tr>
<tr>
<td>XPARMID_DEC_G726_32_RTP_PLD</td>
<td>RTP payload type for G.726 32-Kbps coder. The payload type is negotiated and set by the call stack. The range of values is 96 to 127. Default: 97.</td>
<td>R/W</td>
<td>Y</td>
</tr>
<tr>
<td>XPARMID_DEC_G726_24_RTP_PLD</td>
<td>RTP payload type for G.726 24-Kbps coder. The payload type is negotiated and set by the call stack. The range of values is 96 to 127. Default: 98.</td>
<td>R/W</td>
<td>Y</td>
</tr>
<tr>
<td>XPARMID_DEC_G726_16_RTP_PLD</td>
<td>RTP payload type for G.726 16-Kbps coder. The payload type is negotiated and set by the call stack. The range of values is 96 to 127. Default: 99.</td>
<td>R/W</td>
<td>Y</td>
</tr>
</tbody>
</table>
### 3.3.3 Parameters

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description and values</th>
<th>Attr.</th>
<th>Direct Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>XPARM_ID_RES_STATE</td>
<td>Current state (0: idle, 1: active)</td>
<td>R</td>
<td>N</td>
</tr>
<tr>
<td>XPARM_ID_ENC_LP_STREAM</td>
<td>L-Port stream ID. Default: the stream assigned to the TDM termination’s T-Port of the same channel if exist, otherwise -1.</td>
<td>R/W</td>
<td>N</td>
</tr>
<tr>
<td>XPARM_ID_ENC_AGC</td>
<td>AGC enable flag. XPARM_ON or XPARM_OFF. Default: XPARM_OFF</td>
<td>R/W</td>
<td>N</td>
</tr>
<tr>
<td>XPARM_ID_ENC_VAD</td>
<td>VAD enable flag. XPARM_ON or XPARM_OFF. Default: XPARM_OFF</td>
<td>R/W</td>
<td>N</td>
</tr>
<tr>
<td>XPARM_ID_ENC_MFPP</td>
<td>Number of frames per packet. Supported range is 1 to 6 frames for G.711 and G.722, 1 to 8 frames for G.723, 1 to 9 frames for G.726 40 Kbps, 1 to 12 frames for G.726 32 Kbps, 1 to 16 frames for G.726 24 Kbps, and 1 to 24 frames for G.729 and G.726 16 Kbps. Default: 1.</td>
<td>R/W</td>
<td>N</td>
</tr>
<tr>
<td>XPARM_ID_ENC_EVT_PKT</td>
<td>Enable packet lost event. XPARM_ON or XPARM_OFF. Default: XPARM_OFF</td>
<td>R/W</td>
<td>Y</td>
</tr>
<tr>
<td>XPARM_ID_ENC_G726_40_RTP_PLD</td>
<td>RTP payload type for G.726 40-Kbps coder. The payload type is negotiated and set by the call stack. The range of values is 96 to 127. Default: 96.</td>
<td>R/W</td>
<td>Y</td>
</tr>
<tr>
<td>XPARM_ID_ENC_G726_32_RTP_PLD</td>
<td>RTP payload type for G.726 32-Kbps coder. The payload type is negotiated and set by the call stack. The range of values is 96 to 127. Default: 97.</td>
<td>R/W</td>
<td>Y</td>
</tr>
<tr>
<td>XPARM_ID_ENC_G726_24_RTP_PLD</td>
<td>RTP payload type for G.726 24-Kbps coder. The payload type is negotiated and set by the call stack. The range of values is 96 to 127. Default: 98.</td>
<td>R/W</td>
<td>Y</td>
</tr>
<tr>
<td>XPARM_ID_ENC_G726_16_RTP_PLD</td>
<td>RTP payload type for G.726 16-Kbps coder. The payload type is negotiated and set by the call stack. The range of values is 96 to 127. Default: 99.</td>
<td>R/W</td>
<td>Y</td>
</tr>
<tr>
<td>XPARM_ID_ENC_VOL</td>
<td>Encoder gain adjustment, +15 ~ - 40 in 1-dB units. Default: 0 (Set to -99 to mute)</td>
<td>R/W</td>
<td>N</td>
</tr>
</tbody>
</table>

### 3.3.4 Events

- XPARM_ID_EVT_PACKET — Bad packet.
- XPARM_ID_EVT_PACKET_CHANGE — Received RTP payload type changed.

### 3.4 Tone Generation Resource Component

**Resource Type:** XPAR_TNGEN
3.4.1 Media Processing Functions

• Generating multiple frequency tone signals
• Generating call progress tones

3.4.2 Resource-Specific Control Messages

• XMSG_TG_PLAY (inbound)
• XMSG_TG_PLAY_FSK (inbound)
• XMSG_TG_PLAY_CMPLT (outbound)

3.4.3 Parameters

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description and values</th>
<th>Attr.</th>
<th>Direct Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>XPARMID_RES_STATE</td>
<td>Current state (0: idle, 1: active)</td>
<td>R</td>
<td>N</td>
</tr>
<tr>
<td>XPARMID_TNGEN_VOL</td>
<td>Tone Generator’s volume adjustment, +15 ~ -20 in 1-dB units. Default: 0</td>
<td>R/W</td>
<td>N</td>
</tr>
<tr>
<td>XPARMID_TNGEN_FSK_MOD</td>
<td>FSK modulator mode. XPARM_TNGEN_FSK_V23 or XPARM_TNGEN_FSK_B202. Default: XPARM_TNGEN_FSK_B202 if country code set to COUNTRY_CODE_US or COUNTRY_CODE_PRC, otherwise XPARM_TNGEN_FSK_V23</td>
<td>R/W</td>
<td>Y</td>
</tr>
<tr>
<td>XPARMID_TNGEN_FSK_CS</td>
<td>CS bit length of FSK modulator (in bit unit). Default: 300 if country code set to COUNTRY_CODE_US or COUNTRY_CODE_PRC, otherwise 0.</td>
<td>R/W</td>
<td>Y</td>
</tr>
<tr>
<td>XPARMID_TNGEN_FSK_MARK</td>
<td>Mark bit length of FSK modulator (in bit unit). Default: 180 if country code set to COUNTRY_CODE_US or COUNTRY_CODE_PRC, otherwise 100.</td>
<td>R/W</td>
<td>Y</td>
</tr>
<tr>
<td>XPARMID_TNGEN_FSK_RATE</td>
<td>FSK modulator baud rate (XPARM_TNGEN_FSK_R1200, XPARM_TNGEN_FSK_R600, XPARM_TNGEN_FSK_R300, XPARM_TNGEN_FSK_R150 or XPARM_TNGEN_FSK_R75). Default: XPARM_TNGEN_FSK_R1200, for example, 1200 bps</td>
<td>R/W</td>
<td>N</td>
</tr>
<tr>
<td>XPARMID_TNGEN_FSK_POSTMK</td>
<td>Postmark bit length of FSK modulator (in bit unit). Default: 72.</td>
<td>R/W</td>
<td>Y</td>
</tr>
<tr>
<td>XPARMID_TNGEN_RFC2833</td>
<td>RFC2833 enable flag. XPARM_ON or XPARM_OFF. Default: XPARM_ON</td>
<td>R/W</td>
<td>N</td>
</tr>
</tbody>
</table>

3.4.4 Events

None.

3.5 Tone Detection Resource Component

Resource Type: XPAR_TNDET

3.5.1 Media Processing Functions

• Receiving DTMF digits
• Detecting individual tone event
• Receive FSK
3.5.2 Resource-Specific Control Messages

- XMSG_TD_RCV (inbound)
- XMSG_TD_RCV_FSK (inbound)
- XMSG_TD_RCV_CMPLT (outbound)
- XMSG_TD_RCV_FSK_CMPLT (outbound)

3.5.3 Parameters

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description and values</th>
<th>Attr.</th>
<th>Direct Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>XPARMID_RES_STATE</td>
<td>Current state (0: idle, 1: active)</td>
<td>R</td>
<td>N</td>
</tr>
<tr>
<td>XPARMID_TD_LP_STREAM</td>
<td>L-Port stream ID. Default: the stream assigned to the TDM termination's T-Port of the same channel if exist, otherwise -1.</td>
<td>R/W</td>
<td>N</td>
</tr>
<tr>
<td>XPARMID_TD_TC</td>
<td>Tone Clamping enable flag. XPARM_ON or XPARM_OFF. Default: XPARM_OFF</td>
<td>R/W</td>
<td>Y</td>
</tr>
<tr>
<td>XPARMID_TD_TC_FRAMES</td>
<td>Tone Clamping buffer size. 0 ~ 3 in 10 ms unit. Default: 3</td>
<td>R/W</td>
<td>N</td>
</tr>
<tr>
<td>XPARMID_TD_RPT_EVENTS</td>
<td>Tone event enable flag. XPARM_OFF, XPARM_TD_RPT_TONE_ON, XPARM_TD_RPT_TONE_OFF or XPARM_TD_RPT_TONE_OFF. Default: XPARM_OFF</td>
<td>R/W</td>
<td>Y</td>
</tr>
<tr>
<td>XPARMID_TD_RFC2833E_ENABLE</td>
<td>RFC2833 event enable flag. XPARM_ON or XPARM_OFF. Default: XPARM_OFF</td>
<td>R/W</td>
<td>Y</td>
</tr>
<tr>
<td>XPARMID_TD_RFC2833E_UPDATERATE</td>
<td>RFC 2833 packet rate in 10-ms units, for example, the period between the packets generated when a tone event is detected. Default: 5</td>
<td>R/W</td>
<td>N</td>
</tr>
<tr>
<td>XPARMID_TD_RFC2833E_NUMEOE</td>
<td>Redundancy of end-of-event packet. Range 0-255. Default: 3</td>
<td>R/W</td>
<td>Y</td>
</tr>
<tr>
<td>XPARMID_TD_RFC2833E_NUMBOE</td>
<td>Redundancy of begin-of-event packet. Range 0-255. Default: 0</td>
<td>R/W</td>
<td>Y</td>
</tr>
<tr>
<td>XPARMID_TD_RFC2833E_AUDIOSUPPRESS</td>
<td>Flag of audio encoding suppression when event detected. XPARM_ON or XPARM_OFF. Default: XPARM_ON</td>
<td>R/W</td>
<td>N</td>
</tr>
<tr>
<td>XPARMID_TD_RFC2833E_PAYLOADTYPE</td>
<td>RFC 2833 Payload type, Range is in the RTP dynamic payload type range of 96 to 127. Default: 0x65.</td>
<td>R/W</td>
<td>Y</td>
</tr>
<tr>
<td>XPARMID_TD_FSK_CS</td>
<td>Minimum CS-bit length required by FSK receiver. Default: 200 if country code set to COUNTRY_CODE_US or COUNTRY_CODE_PRC, otherwise 0.</td>
<td>R</td>
<td>Y</td>
</tr>
<tr>
<td>XPARMID_TD_FSK_MARK</td>
<td>Minimum mark-bit length required by FSK receiver. Default: 100 if country code set to COUNTRY_CODE_US or COUNTRY_CODE_PRC, otherwise 60.</td>
<td>R</td>
<td>Y</td>
</tr>
<tr>
<td>XPARMID_TD_FSK_STOP</td>
<td>Extra stop bits allowed between data. Default: 20</td>
<td>R/W</td>
<td>Y</td>
</tr>
<tr>
<td>XPARMID_TD_FSK_RATE</td>
<td>Baud rate of FSK receiver. (Reserved for future, currently only support 1,200 bps rate)</td>
<td>R/W</td>
<td>Y</td>
</tr>
<tr>
<td>XPARMID_TD_FSK_ADAPT_THRESHOLD</td>
<td>FSK detector with Adaptive or Non-Adaptive threshold. Non-Adaptive:0 (default), Adaptive: 1</td>
<td>R</td>
<td>Y</td>
</tr>
<tr>
<td>XPARMID_TD_FSK_RX_MIN_BYTES</td>
<td>Minimum Number of data bytes to receive for FSK detection (0: default, 0~15)</td>
<td>R/W</td>
<td>Y</td>
</tr>
</tbody>
</table>
3.5.4 Events

- XEV_T_CODE_TD_TONEON – tone on event for an individual tone
- XEV_T_CODE_TD_TONEOFF – tone off event for an individual tone

Event data1 gives the tone ID and data2 gives the time stamp in 10-ms units.

3.6 Audio Player Resource Component

Resource Type: XMPR_PLY

3.6.1 Media Processing Functions

- Play back recorded audio data.

3.6.2 Resource-Specific Control Messages

- XMSG_PLY_START (inbound)
- XMSG_PLY_CMPLT (outbound)

3.6.3 Parameters

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description and values</th>
<th>Attr.</th>
<th>Direct Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>XPARMID_RES_STATE</td>
<td>Current state (0: idle, 1: active)</td>
<td>R</td>
<td>N</td>
</tr>
<tr>
<td>XPARMID_PLY_VOL</td>
<td>Volume adjustment (+15 ~ –30 in 1dB unit) Default: 0</td>
<td>R/W</td>
<td>N</td>
</tr>
</tbody>
</table>

3.6.4 Events

None.

3.7 Audio Mixer Resource Component

Resource Type: XMPR_MIX

3.7.1 Media Processing Functions


3.7.2 Resource-Specific Control Messages

None.
3.7.3 Parameters

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description and values</th>
<th>Attr.</th>
<th>Direct Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>XPARMID_RES_STATE</td>
<td>Current state (0: idle, 1: active)</td>
<td>R</td>
<td>N</td>
</tr>
<tr>
<td>XPARMID_MIX_LP_STREAM</td>
<td>The first L-Port stream ID. Default: -1</td>
<td>R/W</td>
<td>N</td>
</tr>
<tr>
<td>XPARMID_MIX_LP_STREAM+1</td>
<td>The 2nd L-Port stream ID. Default: -1</td>
<td>R/W</td>
<td>N</td>
</tr>
<tr>
<td>XPARMID_MIX_LP_STREAM+n-1</td>
<td>The nth L-Port stream ID. Default: -1</td>
<td>R/W</td>
<td>N</td>
</tr>
</tbody>
</table>

3.7.4 Events

None.

3.8 Message Agent Resource Component

Resource Type: XPAR_MA

3.8.1 Media Processing Functions

- No media processing function.
- Converting the user-defined messages and executing the control accordingly (enabling trace, that is, XPARM_MA_DEBUG = XPARM_ON, will give debug prints showing control messages being decoded from a user-defined message.)

3.8.2 Resource-Specific Control Messages

None.

3.8.3 Parameters

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description and values</th>
<th>Attr.</th>
<th>Direct Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>XPARMID_RES_STATE</td>
<td>Current state (0: idle, 1: active)</td>
<td>R</td>
<td>N</td>
</tr>
<tr>
<td>XPARMID_MA_DEBUG</td>
<td>Enable trace during processing user’s messages. XPARM_ON or XPARM_OFF Default: XPARM_OFF</td>
<td>R/W</td>
<td>Y</td>
</tr>
</tbody>
</table>

3.8.4 Events

None.

4.0 Message Format and Delivery Mechanism

There are two message queues (in-bound and out-bound) for the user application to send control messages and to receive response and event messages, respectively. The message queues are created from pre-allocated memory buffers in consideration of maximum message size and total number of messages. The entire message header and content are copied to/from the buffers in the message queue during message
transmitting and receiving. The memory used for messaging is not shared between the message sender and the receiver. For additional description of message queues, refer to the *Intel® Infrastructure DSP Solution Version 1.1 Programmer’s Guide*.

**Note:** Acoustic Echo Canceller (AEC) component in Network Endpoint MPR and T.38 MPR are not supported in Intel® Infrastructure DSP Solution Version 1.1. Error message will be returned if control messages sent have reference to AEC component or T.38 MPR.

### 4.1 Message Functions

Three functions are provided to send and receive messages.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*<em>XStatus_t xMsgSend (void <em>pMsgBuf);</em></em></td>
<td>Sends a control message to the in-bound message queue</td>
</tr>
<tr>
<td><strong>Input</strong></td>
<td>pMsgBuf – Pointer to the message buffer.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td>None</td>
</tr>
</tbody>
</table>
| **Return** | • XSUCC – If successful  
• XERROR – If errors |
| **Caution** | Message buffer requires 4-byte alignment. |
| **Note** | Message buffer can be used for any other purpose after sending. |

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*<em>XStatus_t xMsgReceive (void <em>pMsgBuf, UINT16 channel, int timeout);</em></em></td>
<td>Receives acknowledgement or event from the outbound message queue.</td>
</tr>
</tbody>
</table>
| **Input** | • pMsgBuf – Pointer to the message buffer  
• channel – Channel number. Set to number 0. (Reserved for future extension)  
• timeout – Waiting flag  
  – XWAIT_NONE – If return immediately  
  – XWAIT_FOREVER – If never time out (no other values are valid.) |
| **Output** | None |
| **Return** | • XSUCC – If successful  
• XERROR – If errors |
| **Caution** | Message buffer requires 4-byte alignment. The receiving buffer must fit the maximum message size. Cannot be called from ISR. |

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*<em>XStatus_t xMsgWrite (void <em>pMsgBuf);</em></em></td>
<td>Posts a message (e.g. a user-defined external event message) to the outbound queue so that it can be retrieved by XMsgReceive().</td>
</tr>
<tr>
<td><strong>Input</strong></td>
<td>pMsgBuf – Pointer to the message buffer.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td>None</td>
</tr>
</tbody>
</table>
| **Return** | • XSUCC – If successful  
• XERROR – If errors |
| **Caution** | Message buffer requires 4-byte alignment. |
| **Note** | The message buffer can be used for any other purpose, after posting. |
4.2 Message Header Format

```c
typedef struct{
    UINT32 transactionId; /* used by apps to track the message */
    UINT16 instance; /* instance ID (1-0xffffffff), 0:reserved */
    UINT8 resource; /* MPR resource type */
    UINT8 reserved; /* reserved for future */
    UINT16 size; /* total size in bytes */
    UINT8 type; /* message type */
    UINT8 attribute; /* attribute, reserved for future */
} XMsgHdr_t, *XMsgRef_t;
```

Caution: Message content must follow the header in contiguous memory.

Macros

```c
#define XMSG_MAKE_HEAD(pMsg, trans, res, inst, sz, typ, attr) 
    (((XMsgRef_t)(pMsg))->transactionId = trans;\ 
    (((XMsgRef_t)(pMsg))->instance = inst;\ 
    (((XMsgRef_t)(pMsg))->resource = res;\ 
    (((XMsgRef_t)(pMsg))->reserved = 0;\ 
    (((XMsgRef_t)(pMsg))->size = sz;\ 
    (((XMsgRef_t)(pMsg))->type = typ;\ 
    (((XMsgRef_t)(pMsg))->attribute = attr;
```

4.3 Message Type List

All message types are pre-defined as:
Typedef enum{
  XMSG_BEGIN =0, /* Begin list */
  XMSG_RESET, /* reset a resource */
  XMSG_START, /* start media processing a SP resource */
  XMSG_STOP, /* stop a current action on a SP resource */
  XMSG_PING, /* ping a SP resource */
  XMSG_SET_PARM, /* set a parameter on a SP resource */
  XMSG_SET_MPARMS, /* set multiple parameters on a SP resource */
  XMSG_GET_PARM, /* get a parameter from a SP resource */
  XMSG_GET_PARM_ACK, /* acknowledgement to get parameter message */
  XMSG_GET_ALLPARMS, /* get all parameters from a SP resource */
  XMSG_GET_ALLPARMS_ACK, /* acknowledgement to get all parameter message */
  XMSG_ACK, /* general acknowledgement message */
  XMSG_ERROR, /* error message from SP resource */
  XMSG_EVENT, /* event message from SP resource */
  XMSG_CODER_START, /* start a codec resource */
  XMSG_CODER_STOP_ACK, /* acknowledgement to stop message */
  XMSG_TG_PLAY, /* play a digit string on a TG instance */
  XMSG_TG_PLAY_FSK, /* play FSK modulated data */
  XMSG_TG_PLAY_CMPLT, /* play-completed message from a TG instance */
  XMSG_TD_RCV, /* receive a digit string on a TD instance */
  XMSG_TD_RCV_CMPLT, /* receive-completed message from a channel */
  XMSG_TD_RCV_FSK, /* receive a FSK signal on a TD instance */
  XMSG_TD_RCV_FSK_CMPLT, /* receive-completed message from TD instance */
  XMSG_PLY_START, /* start playing audio on a Player instance */
  XMSG_PLY_CMPLT, /* play-completed message from Player */
  XMSG_GET_JBSTAT, /* get jitter buffer statistics from Dec */
  XMSG_GET_JBSTAT_CMPLT, /* response to the get-JB-statistics msg */
  XMSG_GET_ERLESTAT, /* get ERLE statistics from net instance */
  XMSG_GET_ERLESTAT_CMPLT, /* response to the get ERLE statistics message */
  XMSG_GET_MFPPSTAT, /* get MFPP statistics from encoder instance */
  XMSG_GET_MFPPSTAT_CMPLT, /* response to the get MFPP statistics message */
  XMSG_T38_START, /* start T.38 resource */
  XMSG_T38_CMPLT, /* T.38 session complete message */
  /*------------------Definition for internal messages ------------------*/
  XMSG_STATE_FWD, /* state forward message */
  XMSG_END_OF_LIST /* end of list */
} XMsgType_t;
5.0 Common Control Message

This section defines the control messages that can be applied to all the resources.

Note: Acoustic Echo Canceller (AEC) component in Network Endpoint MPR and T.38 MPR are not supported in Intel® Infrastructure DSP Solution Version 1.1. Error messages will be returned if control messages sent have reference to AEC component or T.38 MPR.

5.1 Reset Message

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_RESET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Inbound</td>
</tr>
<tr>
<td>Description</td>
<td>Stops the current action and resets the resource to idle state.</td>
</tr>
</tbody>
</table>

Format:
```c
typedef struct{
    XMsgHdr_t head; /* message header */
} XMsgReset_t;
```

Macro:
```c
#define XMSG_MAKE_RESET(pMsg, trans, res, inst) 
{ 
    XMSG_MAKE_HEAD(pMsg, trans, res, inst, sizeof(XMsgReset_t),
    XMSG_RESET, 0)
}
```

Response:
- General acknowledgement message (XMSG_ACK)
- Error message (XMSG_ERROR) if error.

Caution: Any intermediate results are discarded.

5.2 Start Message

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_Start</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Inbound</td>
</tr>
<tr>
<td>Description</td>
<td>Generic start message. Starts the media-processing functions on a resource.</td>
</tr>
</tbody>
</table>

Format:
```c
typedef struct{
    XMsgHdr_t head; /* message header */
} XMsgStart_t;
```

Macro:
```c
#define XMSG_MAKE_START(pMsg, trans, res, inst) 
{ 
    XMSG_MAKE_HEAD(pMsg, trans, res, inst, sizeof(XMsgStart_t),
    XMSG_START, 0)
}
```

Response:
- General acknowledgement message (XMSG_ACK)
- Error message (XMSG_ERROR) if error.

Caution: This message is not applicable to Tone Generator and Player resources.
### 5.3 Stop Message

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_STOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Inbound</td>
</tr>
<tr>
<td>Description</td>
<td>Stops the current action.</td>
</tr>
</tbody>
</table>

**Format**

```c
typedef struct{
    XMsgHdr_t head; /* message header */
} XMsgStop_t;
```

**Macro**

```c
#define XMSG_MAKE_STOP(pMsg, trans, res, inst)
{|
    XMSG_MAKE_HEAD(pMsg, trans, res, inst, sizeof(XMsgStop_t),
    XMSG_STOP, 0)\n}
```

**Response**

Resource returns the processing results or states, if any, depending on the resources and current actions.

### 5.4 Ping Message

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_PING</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Inbound</td>
</tr>
<tr>
<td>Description</td>
<td>Verifies if the resource is alive.</td>
</tr>
</tbody>
</table>

**Format**

```c
typedef struct{
    XMsgHdr_t head; /* message header */
} XMsgPing_t;
```

**Macro**

```c
#define XMSG_MAKE_PING(pMsg, trans, res, inst) |
{|
    XMSG_MAKE_HEAD(pMsg, trans, res, inst, sizeof(XMsgPing_t),
    XMSG_PING, 0)\n}
```

**Response**

- General acknowledgement message (XMSG_ACK)
- Error message (XMSG_ERROR) if error.
## 5.5 Set Parameter Message

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_SET_PARM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Inbound</td>
</tr>
<tr>
<td>Description</td>
<td>Sets a parameter to a resource.</td>
</tr>
</tbody>
</table>
| Format                | typedef struct{
  XMsgHdr_t head; /* message header */
  UINT16 parmId; /* parameter id */
  UINT16 value; /* parameter value */
} XMsgSetParm_t; |
| Macro                 | #define XMSG_MAKE_SET_PARM(pMsg, trans, res, inst, id, val) \ |
|                      | { \ |
|                      |   XMSG_MAKE_HEAD(pMsg, trans, res, inst, |
|                      |   sizeof(XMsgSetParm_t),
|                      |   XMSG_SET_PARM, 0)\ |
|                      |   ((XMsgSetParm_t *)(pMsg))->parmId= id;\ |
|                      |   ((XMsgSetParm_t *)(pMsg))->value= val;\ |
} |
| Response              | • General acknowledgement message (XMSG_ACK) |
|                       | • Error message (XMSG_ERROR) if error. |

## 5.6 Set Multiple-Parameter Message

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_SET_MPARMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Inbound</td>
</tr>
<tr>
<td>Description</td>
<td>Set multiple parameters to a resource</td>
</tr>
</tbody>
</table>
| Format                | typedef struct{
  XMsgHdr_t head; /* message header */
  UINT16 numParms; /* number of parameters */
  UINT16 parmIDs[XMAX_PARMS]; /* parameter id */
  UINT16 values[XMAX_PARMS]; /* parameter value */
} XMsgSetxParms_t; |
| Macro                 | #define XMSG_MAKE_SET_MPARMS(pMsg, trans, res, inst, num) \ |
|                      | { \ |
|                      |   XMSG_MAKE_HEAD(pMsg, trans, res, inst, |
|                      |   sizeof(XMsgSetxParms_t),
|                      |   XMSG_SET_MPARMS, 0)\ |
|                      |   ((XMsgSetxParms_t *)(pMsg))->numParms = num; \ |
|                      | } |
|                      | #define XMSG_FIELD_SET_MPARMS(pMsg, pIDs, pVals) \ |
|                      | { \ |
|                      |   pIDs =((XMsgSetxParms_t *)(pMsg))->parmIDs;\ |
|                      |   pVals = ( (XMsgSetxParms_t *)(pMsg))->values;\ |
} |
| Response              | • General acknowledgement message (XMSG_ACK) |
|                       | • Error message (XMSG_ERROR) if error. |
## 5.7 Get Parameter Message

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_GET_PARM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Inbound</td>
</tr>
<tr>
<td>Description</td>
<td>Gets a parameter from a resource.</td>
</tr>
</tbody>
</table>
| Format        | ```
typedef struct{
    XMsgHdr_t head; /* message header */
    UINT16 parmId; /* parameter id */
} XMsgGetParm_t;
```
| Macro         | ```
#define XMSG_MAKE_GET_PARM(pMsg, trans, res, inst, id) \
{\n    XMSG_MAKE_HEAD(pMsg, trans, res, inst, sizeof(XMsgGetParm_t),\n    XMSG_GET_PARM, 0)\n    ((XMsgGetParm_t *)(pMsg))->parmId= id;\n}
```
| Response      | • Specific acknowledgement message (XMSG_GET_PARM_ACK)  
• Error message (XMSG_ERROR) if error. |

## 5.8 Get Parameter Acknowledge Message

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_GET_PARM_ACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Outbound</td>
</tr>
<tr>
<td>Description</td>
<td>Resource returns the parameter enquired.</td>
</tr>
</tbody>
</table>
| Format        | ```
typedef struct{
    XMsgHdr_t head; /* message header */
    UINT16 parmId; /* parameter id */
    UINT16 value; /* parameter value */
} XMsgGetParmAck_t;
```
| Macro         | ```
#define XMSG_FIELD_GET_PARM_ACK(pMsg, id, val)\n{\n    id = ((XMsgGetParmAck_t *)(pMsg))->parmId;\n    val = ((XMsgGetParmAck_t *)(pMsg))->value;\n}
```
## 5.9 Get All Parameters Message

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_GET_ALLPARMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Inbound</td>
</tr>
<tr>
<td>Description</td>
<td>Gets all parameters from a resource.</td>
</tr>
</tbody>
</table>
| Format        | `typedef struct{      
                  XMsgHdr_t head;  /* message header */
                } XMsgGetAllParms_t;` |
| Macro         | `#define XMSG_MAKE_GET_ALLPARMS(pMsg, trans, res, inst) \  
               {                                             
                  XMSG_MAKE_HEAD(pMsg, trans, res, inst, \  
                    sizeof(XMsgGetAllParms_t), \  
                    XMSG_GET_ALLPARMS, 0)\  
               }` |
| Response      | Specific acknowledgement message (XMSG_GET_ALLPARMS_ACK) |

## 5.10 Get All Parameters Acknowledge Message

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_GET_ALLPARMS_ACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Outbound</td>
</tr>
<tr>
<td>Description</td>
<td>Resource returns the parameter inquired.</td>
</tr>
</tbody>
</table>
| Format        | `typedef struct{      
                  XMsgHdr_t head;  /* message header */
                  UINT16 numParms;  /* number of parameters */
                  UINT16 parmIDs[XMAX_PARMS_GET];  /* array of parameter IDs */
                  UINT16 values[XMAX_PARMS_GET];  /* array of parameter values */
                } XMsgGetAllParmsAck_t;` |
| Macro         | `#define XMSG_FIELD_GET_ALLPARMS_ACK(pMsg, num, pIDs, pVals)\  
               {                                                   
                  num  = ((XMsgGetAllParmsAck_t *)(pMsg))->numParms;\  
                  pIDs = ((XMsgGetAllParmsAck_t *)(pMsg))->parmIDs;\  
                  pVals = ((XMsgGetAllParmsAck_t *)(pMsg))->values;\  
               }` |

## 5.11 General Acknowledge Message

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_ACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Outbound</td>
</tr>
<tr>
<td>Description</td>
<td>Resource indicates the control message has been processed successfully.</td>
</tr>
</tbody>
</table>
| Format        | `typedef struct{      
                  XMsgHdr_t head;  /* message header */
                } XMsgAck_t;` |
### 5.12 Error Message

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_ERROR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Outbound</td>
</tr>
<tr>
<td>Description</td>
<td>Resource reports an error condition. (See constant data section for error codes.)</td>
</tr>
</tbody>
</table>

**Format**

```c
typedef struct{
    XMsgHdr_t head; /* message header */
    UINT32 code; /* error code */
    UINT32 data1; /* error data1 */
    UINT32 data2; /* error data2 */
} XMsgError_t;
```

**Macro**

```c
#define XMSG_FIELD_ERROR(pMsg, c, d1, d2)
{
    c = ((XMsgError_t *)(pMsg))->code;
    d1 = ((XMsgError_t *)(pMsg))->data1;
    d2 = ((XMsgError_t *)(pMsg))->data2;
}
```

### 5.13 Event Message

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_EVENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Outbound</td>
</tr>
<tr>
<td>Description</td>
<td>Resource reports an event condition. (See constant data section for error codes.)</td>
</tr>
</tbody>
</table>

**Format**

```c
typedef struct{
    XMsgHdr_t head; /* message header */
    UINT32 code; /* event code */
    UINT32 data1; /* event data1 */
    UINT32 data2; /* event data2 */
} XMsgEvent_t;
```

**Macro**

```c
#define XMSG_FIELD_EVENT(pMsg, c, d1, d2)
{
    c = ((XMsgEvent_t *)(pMsg))->code;
    d1 = ((XMsgEvent_t *)(pMsg))->data1;
    d2 = ((XMsgEvent_t *)(pMsg))->data2;
}
```
# 6.0 Resource-Specific Control Messages

This section defines the resource-specific messages.

## 6.1 CODEC Start Message

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_CODER_START</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Inbound</td>
</tr>
<tr>
<td>Description</td>
<td>Starts a decoder or encoder.</td>
</tr>
</tbody>
</table>

```
typedef struct{
    XMsgHdr_t head;    /* message header */
    UINT16 codecType; /* codec type */
    UINT16 frmsPerPkt; /* number of frames per packet */
} XMsgCoderStart_t;
```

### Macro
```
#define XMSG_MAKE_CODER_START(pMsg, trans, res, inst, cType, fpp)\
    {\
        XMSG_MAKE_HEAD(pMsg, trans, res, inst, sizeof(XMsgCoderStart_t),
                        XMSG_CODER_START, 0)\
        ((XMsgCoderStart_t *)(pMsg))->codecType = cType;\
        ((XMsgCoderStart_t *)(pMsg))->frmsPerPkt = fpp;\
    }
```

**Response**
- General acknowledgement message (XMSG_ACK)
- Error message (XMSG_ERROR) if error.

## 6.2 CODEC Stop Acknowledgement Message

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_CODER_STOP_ACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Outbound</td>
</tr>
<tr>
<td>Description</td>
<td>Decoder or encoder resource acknowledges the XMSG_STOP message</td>
</tr>
</tbody>
</table>

```
typedef struct{
    XMsgHdr_t head;    /* message header */
    UINT32 numFrames; /* total number of frames processed */
    UINT32 numBadFrames; /* number of bad frames */
} XMsgCoderStopAck_t;
```

### Macro
```
#define XMSG_FIELD_EVENT(pMsg, num, numBad)\
    {\
        num = ((XMsgCoderStopAck_t *)(pMsg))->numFrames;\
        numBad = ((XMsgCoderStopAck_t *)(pMsg))->numBadFrames;\
    }
```
### 6.3 Tone Generator Play Message

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_TG_PLAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Inbound</td>
</tr>
<tr>
<td>Description</td>
<td>Requires Tone Generator to play a tone string. (Tone IDs are listed in the constant data section.)</td>
</tr>
</tbody>
</table>
| Format        | typedef struct{  
|               |     XMsgHdr_t head; /* message header */  
|               |     UINT8 numTones; /* number of tones to play */  
|               |     UINT8 toneId[XMAX_TONEBUFSIZE]; /* tone ID string */  
|               | } XMsgTGPPlay_t; |
| Macro         | #define XMSG_MAKE_TG_PLAY(pMsg, trans, inst, num)\  
|               |     {  
|               |     XMSG_MAKE_HEAD(pMsg, trans, XMPR_TNGEN, inst,  
|               |         sizeof(XMsgTGPPlay_t),\  
|               |         XMSG_TG_PLAY, 0)\  
|               |         ((XMsgTGPPlay_t *)(pMsg))->numTones = num;\  
|               |     }  
|               | #define XMSG_FIELD_TG_PLAY(pMsg, pToneID) \  
|               |     {  
|               |     pToneID= ((XMsgTGPPlay_t *)(pMsg))->toneId;\  
|               |     } |
| Response      | • Tone Generator Play-Completed message (XMSG_TG_PLAY_CMPLT)  
|               | • Error message (XMSG_ERROR) if error. |

### 6.4 Tone Generator Play FSK Message

<table>
<thead>
<tr>
<th>Type</th>
<th>MSG_TG_PLAY_FSK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Inbound</td>
</tr>
<tr>
<td>Description</td>
<td>Require Tone Generator to play a FSK modulated data</td>
</tr>
</tbody>
</table>
| Format        | typedef struct{  
|               |     XMsgHdr_t head; /* message header */  
|               |     UINT8 numBytes; /* number of bytes to play */  
|               |     INT8 data[XMAX_FSKDATASIZE]; /* data string */  
|               | } XMsgTGPPlayFSK_t; |
| Macro         | #define XMSG_MAKE_TG_PLAY_FSK(pMsg, trans, inst, num)\  
|               |     {  
|               |     XMSG_MAKE_HEAD(pMsg, trans, XMPR_TNGEN, inst,  
|               |         sizeof(XMsgTGPPlayFSK_t),\  
|               |         XMSG_TG_PLAY_FSK, 0)\  
|               |         ((XMsgTGPPlayFSK_t *)(pMsg))->numBytes = num;\  
|               |     }  
|               | #define XMSG_FIELD_TG_PLAY_FSK(pMsg, pData) \  
|               |     {  
|               |     pData= ((XMsgTGPPlayFSK_t *)(pMsg))->data;\  
|               |     } |
| Response      | • Tone Generator Play-Completed message (XMSG_TG_PLAY_CMPLT)  
|               | • Error message (XMSG_ERROR) if error. |
## 6.5 Tone Generator Play Completed Message

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_TG_PLAY_CMPLT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Outbound</td>
</tr>
<tr>
<td>Description</td>
<td>Tone Generator indicates the completion of playing tones.</td>
</tr>
</tbody>
</table>

### Format

typedef struct{
    XMsgHdr_t head; /* message header */
    UINT16 reason; /* the reason of completion: */
    UINT8 numTones; /* number of tones played. 0 if FSK data */
} XMsgTGPlayCmplt_t;

### Macro

#define XMSG_FIELD_TG_PLAY_CMPLT(pMsg, rsn, num)\   
{\    reason= ((XMsgTGPlayCmplt_t *)(pMsg))->reason;\    num = ((XMsgTGPlayCmplt_t *)(pMsg))->numTones;\ }

## 6.6 Tone Detector Receive Digit Message

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_TD_RCV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Inbound</td>
</tr>
<tr>
<td>Description</td>
<td>Require Tone Detector to receive a tone string.</td>
</tr>
</tbody>
</table>

### Format

typedef struct{
    XMsgHdr_t head; /* message header */
    UINT16 totalTimeout; /* total time out (in 10 ms unit)*/
    UINT16 firstDigitTimeout; /* first digit time out (10 ms uint)*/
    UINT16 interDigitTimeout; /* inter digit time out (10 ms unit)*/
    UINT16 termDigit; /* OR'd terminate digit bits */
    UINT8 numDigits; /* number of digits to receive */
} XMsgTDRcv_t;

### Macro

#define XMSG_MAKE_TD_RCV(pMsg, trans, inst, num, term, tm, fstTm, intTm)\  
{\    XMSG_MAKE_HEAD(pMsg, trans, XMPR_TNDET, inst, \        sizeof(XMsgTDRcv_t), XMSG_TD_RCV, 0)\    ((XMsgTDRcv_t *)(pMsg))->numDigits = num;\    ((XMsgTDRcv_t *)(pMsg))->termDigit = term;\    ((XMsgTDRcv_t *)(pMsg))->totalTimeout = tm;\    ((XMsgTDRcv_t *)(pMsg))->firstDigitTimeout = fstTm;\    ((XMsgTDRcv_t *)(pMsg))->interDigitTimeout = intTm;\ }

### Response
- Tone detector receives completed message (XMSG_TD_RCV_CMPLT)
- Error message (XMSG_ERROR) if error.
### 6.7 Tone Detector Receive Completed Message

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_TD_RCV_CMPLT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Outbound</td>
</tr>
<tr>
<td>Description</td>
<td>Tone detector indicates the completion of receiving DTMF tones.</td>
</tr>
</tbody>
</table>

```c
typedef struct{
    XMsgHdr_t head;  /* message header */
    UINT16 reason;   /* the reason of completion */
    UINT8 numDigits; /* number of tones received */
    UINT8 digits[XMAX_DIGITBUFSIZE]; /* received tone IDs */
} XMsgTDRcvCmplt_t;
```

where the reason may be:

- #define XMSG_STOP_REASON_EOD  2
- #define XMSG_STOP_REASON_TERM 3
- #define XMSG_STOP_REASON_TIMEOUT 4

### 6.8 Tone Detector Receive FSK Message

<table>
<thead>
<tr>
<th>Type</th>
<th>MSG_TD_RCV_FSK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Inbound</td>
</tr>
<tr>
<td>Description</td>
<td>Require Tone Detector to receive FSK data</td>
</tr>
</tbody>
</table>

```c
typedef struct{
    XMsgHdr_t head;  /* message header */
    UINT16 timeout;  /* total time out (in 10 ms unit) */
} XMsgTDRcvFSK_t;
```

#### Macro

```c
#define XMSG_MAKE_TD_RCV_FSK(pMsg, trans, inst, tmout)\
{\n    XMSG_MAKE_HEAD(pMsg, trans, XMPR_TNDET, inst,\
        sizeof(XMsgTDRcvFSK_t), XMSG_TD_RCV_FSK, 0)\n    ((XMsgTDRcvFSK_t *)(pMsg))->timeout = tmout;\n}
```

#### Response
- Tone Detector FSK receive-completed message (XMSG_TD_RCV_FSK_CMPLT)
- Error message (XMSG_ERROR) if error.
### 6.9 Tone Detector FSK Receive Completed Message

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_TD_RCV_FSK_CMPLT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Outbound</td>
</tr>
<tr>
<td>Description</td>
<td>Tone Detector indicates the completion of receiving FSK data</td>
</tr>
</tbody>
</table>

#### Format

```c
typedef struct{
    XMsgHdr_t  head;               /* message header */
    UINT16    reason;               /* the reason of completion */
    UINT8    numBytes;             /* number of bytes received */
    UINT8   data[XMAX_FSKDATASIZE];/* received data */
} XMsgTDRcvFskCmplt_t;
```

Where the reason may be:
- `#define XMSG_STOP_REASON_EOD        2`
- `#define XMSG_STOP_REASON_TIMEOUT    4`

#### Macro

```c
#define XMSG_FIELD_TD_RCV_FSK_CMPLT(pMsg, rsn, num, pBuf)\
{\
    rsn = ((XMsgTDRcvFskCmplt_t *)(pMsg))->reason;\
    num = ((XMsgTDRcvFskCmplt_t *)(pMsg))->numBytes;\
    pBuf= ((XMsgTDRcvFskCmplt_t *)(pMsg))->data;\
}
```
### 6.10 Player Start Message

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_PLY_START</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Inbound</td>
</tr>
<tr>
<td>Description</td>
<td>Start Player to play back pre-recorded audio data</td>
</tr>
</tbody>
</table>

**Format**

```c
typedef struct{
    XMsgHdr_t       head;           /* message header */
    XPlyMediaDesc_t mediaSeg[XMAX_PLY_SEG];  /* media segments to play */
    UINT8           numSeg;         /* number of segments */
} XMsgPlyStart_t;
```

where the media segment data structure is defined as

```c
typedef struct{
    INT32 offset;           /* offset in byte where player starts */
    INT32 length;           /* length to play (in 10ms unit), 0 means playing till end of this segment*/
    XMediaHandle_t  handle; /* media storage handle */
    INT16 next;             /* the relative index of next segment followed,XPLY_MEDIA_SEG_EOP means end-of-play at this segment */
} XPlyMediaDesc_t;
```

**Macro**

```c
#define XMSG_MAKE_PLY_START(pMsg, trans, inst, num)
{    
    XMSG_MAKE_HEAD(pMsg, trans, XMPR_PLY, inst, sizeof(XMsgPlyStart_t), XMSG_PLY_START, 0)
    ((XMsgPlyStart_t *)(pMsg))->numSeg = num;
}
define XMSG_FIELD_PLY_START(pMsg, pMedia)  
{   
pMedia   = ((XMsgPlyStart_t *)(pMsg))->mediaSeg;
}
```

**Response**

- Player play-completed message (XMSG_PLY_CMPLT)
- Error message (XMSG_ERROR) if error.

### 6.11 Player Play Completed Message

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_PLY_CMPLT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Outbound</td>
</tr>
<tr>
<td>Description</td>
<td>Player indicates the completion of playing audio data.</td>
</tr>
</tbody>
</table>

**Format**

```c
typedef struct{
    XMsgHdr_t       head;           /* message header */
    UINT16      reason;         /* the reason of completion */
} XMsgPlyCmplt_t;
```

**Macro**

```c
#define XMSG_STOP_REASON_USER    1
#define XMSG_STOP_REASON_EOD     2
```

```c
#define XMSG_FIELD_PLY_CMPLT(pMsg, rsn)  
{   
    rsn = ((XMsgPlyCmplt_t *)(pMsg))->reason;
}
```
## 6.12 Get Jitter Buffer Statistics Message

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_GET_JBSTAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Inbound</td>
</tr>
<tr>
<td>Description</td>
<td>Get the jitter buffer statistics from a Decoder instance.</td>
</tr>
</tbody>
</table>

**Format**

```c
typedef struct{
    XMsgHdr_t head; /* message header */
    UINT16 reset; /* reset flag, 1: reset statistics after retrieve the information */
} XMsgGetJBStat_t;
```

**Macro**

```c
#define XMSG_MAKE_GET_JBSTAT(pMsg, trans, inst, clr)
{
    XMSG_MAKE_HEAD(pMsg, trans, XMPR_DEC, inst, sizeof(XMsgGetJBStat_t), XMSG_GET_JBSTAT, 0)
    ((XMsgGetJBStat_t *)(pMsg))->reset = clr;
}
```

**Response**

- Complete message of getting jitter buffer statistics (XMSG_GET_JBSTAT_CMPLT)
- Error message (XMSG_ERROR) if error.

## 6.13 Complete Message of Getting Jitter Buffer Statistics

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_GET_JBSTAT_CMPLT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Outbound</td>
</tr>
<tr>
<td>Description</td>
<td>Response to the message of getting the jitter buffer statistics.</td>
</tr>
</tbody>
</table>

**Format**

```c
typedef struct{
    XMsgHdr_t head; /* message header */
    XJBStatistics_t stat; /* jitter buffer statistics */
} XMsgGetJBStatCmplt_t;
```

where the `XMsgGetJBStatCmplt_t` data structure of jitter buffer statistics is defined as

```c
typedef struct{
    UINT32 rcvdPackets; /* total packets received */
    UINT32 lostPackets; /* lost packets */
    UINT32 badFrames; /* decoder bad frames */
    UINT32 rcvdTonePackets; /* RFC2833 packets received */
    UINT32 maxJitter; /* maximum jitter in millisecs */
    UINT32 minJitter; /* minimum jitter in millisecs */
    UINT32 meanJitter; /* mean jitter in millisecs */
    UINT32 devJitter; /* standard deviation jitter in millisecs */
    UINT16 jbMaxDelay; /* JB maximum delay in millisecs */
    UINT16 jbAbsMaxDelay; /* JB absolute max delay in millisecs */
    UINT16 jbNominalDelay; /* JB nominal delay in millisecs */
    UINT8 discardRate; /* Discard Rate in number */
} XJBStatistics_t;
```

**Macro**

```c
#define XMSG_FIELD_GET_JBSTAT_CMPLT(pMsg, pStat)
{
    pStat = &(((XMsgGetJBStatCmplt_t *)(pMsg))->stat);
}
```
### 6.14 Get ERLE Statistics Message

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_GET_ERLESTAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Inbound</td>
</tr>
<tr>
<td>Description</td>
<td>Get the ERLE extended report statistics for ERLE from a NET instance.</td>
</tr>
</tbody>
</table>
| Format                | ```
typedef struct{
    XMsgHdr_t head;  /* message header */
} XMsgGetERLEStat_t;
``` |
| Macro                 | ```
#define XMSG_MAKE_GET_ERLESTAT (pMsg, trans, inst)\ 
    {
        XMSG_MAKE_HEAD (pMsg, trans, XMPR_NET, inst, \
        sizeof(XMsgGetERLEStat_t), XMSG_GET_ERLESTAT, 0)\ 
    }
``` |
| Response              | Complete message of getting extended report statistics on ERLE. (XMSG_GET_ERLESTAT_CMPLT) |

### 6.15 Complete Message of Getting ERLE statistics

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_GET_ERLE_STAT_CMPLT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Outbound</td>
</tr>
<tr>
<td>Description</td>
<td>Response to the message of getting the statistics about ERLE.</td>
</tr>
</tbody>
</table>
| Format                | ```
typedef struct{
    XMsgHdr_t head;  /* message header */
    XERLEStatistics_t stat;  /* ERLE- extended report statistics */
} XMsgGetERLEStatCmplt_t;
``` where the XERLEStatistics_t data structure is defined as below. ```
typedef struct {
    UINT8 erle;  /* echo return loss enhancement in dB*/
} XERLEStatistics_t;
``` |
| Macro                 | ```
#define XMSG_FIELD_GET_ERLESTAT_CMPLT (pMsg, pStat)\ 
    {\n        pStat = &(((XMsgGetERLEStatCmplt_t *)(pMsg))->stat);\n    }
``` |
### 6.16 Get MFPP Statistics Message

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_GET_MFPPSTAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Inbound</td>
</tr>
<tr>
<td>Description</td>
<td>Get the MFPP statistics for MFPP from a ENCODER instance.</td>
</tr>
</tbody>
</table>
| Format          | typedef struct{
|                 | XMsgHdr_t           head;   /* message header */
|                 | } XMsgGetMFPPStat_t; |
| Macro           | #define XMSG_MAKE_GET_MFPPSTAT (pMsg, trans, inst)\ |
|                 | {\                        |
|                 |   XMSG_MAKE_HEAD (pMsg, trans, XMPR_ENC, inst, \|
|                 |   sizeof (XMsgGetMFPPStat_t), XMSG_GET_MFPPSTAT, 0)\|
|                 | }                         |
| Response        | Complete message of getting extended report statistics on MFPP. |

### 6.17 Complete Message of Getting MFPP statistics

<table>
<thead>
<tr>
<th>Type</th>
<th>XMSG_GET_MFPP_STAT_CMPLT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Outbound</td>
</tr>
<tr>
<td>Description</td>
<td>Response to the message of getting the statistics about MFPP.</td>
</tr>
</tbody>
</table>
| Format          | typedef struct{
|                 |   XMsgHdr_t           head;   /* message header */
|                 |   XMFPStatistics_t   stat;   /* MFPP- extended report statistics */
|                 | } XMsgGetMFPPStatCmplt_t; |
| Macro           | #define XMSG_FIELD_GET_MFPPSTAT_CMPLT (pMsg, pStat)\ |
|                 | {\                        |
|                 |   pStat = &(((XMsgGetMFPPStatCmplt_t *)(pMsg))->stat); \|
|                 | }                         |

### 7.0 Packet Data Interface

The packet data interface is a protocol for the DSP software to exchange the encoded data packets with IP stack. This interface is defined as a packet format and has two callback functions – one is provided by the DSP software and another is provided by the user (IP stack).
### 7.1 Packet Formats

The ingress packet from the IP stack to the DSP software has an 8-byte header as shown below:

```
+----------------+-----------------+-----------------+-----------------+----------------+-----------------+-----------------+-----------------+-----------------+
| 31  | 24  | 23  | 16  | 15  | 12  | 11  | 0   | Bit Position |
+----------------+-----------------+-----------------+-----------------+----------------+-----------------+-----------------+-----------------+-----------------+
| Channel ID     | Payload Type    | Media Type      | Payload Length  | Remote Time Stamp|                | Payload         |                |                |
|                |                |                 |                 |                |                |                |                |                |
+----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+
```

Similarly, the egress packet from the DSP software to the IP stack has an 8-byte header as shown below:

```
+----------------+-----------------+-----------------+-----------------+----------------+-----------------+-----------------+-----------------+
| 31  | 24  | 23  | 16  | 15  | 12  | 11  | 0   | Bit Position |
+----------------+-----------------+-----------------+-----------------+----------------+-----------------+-----------------+-----------------+-----------------+
| Channel ID     | Payload Type    | Media Type      | Payload Length  | Local Time Stamp|                | Payload         |                |                |
|                |                |                 |                 |                |                |                |                |                |
+----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+
```

The fields of the packet header and the payload are described as:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Time Stamp</td>
<td>Packet arrival time as measured by a local clock.</td>
</tr>
<tr>
<td>Remote Time Stamp</td>
<td>Packet data sampling time measured by a remote clock.</td>
</tr>
<tr>
<td>Payload Length</td>
<td>Payload length in bytes.</td>
</tr>
<tr>
<td>Media</td>
<td>4-bit media type field is defined as:</td>
</tr>
<tr>
<td></td>
<td>- 0x01  Audio</td>
</tr>
<tr>
<td></td>
<td>- 0x02  Tone (RFC2833 event type)</td>
</tr>
<tr>
<td></td>
<td>- 0x04  Tone (RFC2833 tone type)</td>
</tr>
<tr>
<td></td>
<td>- 0x08  T.38 UDP (T.38 MPR is not supported)</td>
</tr>
<tr>
<td></td>
<td>- 0x09  T.38 TCP (T.38 MPR is not supported)</td>
</tr>
<tr>
<td>M</td>
<td>Marker bit for the RTP packet. This bit set indicates the first speech packet after a silence period or the first packet of a RFC-2833 tone event, otherwise 0.</td>
</tr>
<tr>
<td>Payload type</td>
<td>RTP payload type as defined in RFC1890.</td>
</tr>
<tr>
<td>Payload</td>
<td>Encoded audio data or RFC2833, tone-event information.</td>
</tr>
</tbody>
</table>

The corresponding data structure is defined as:
typedef struct{
    UINT8    channelID;   /* channel ID */
    UINT8    payloadType; /* bit[0-6]payloadtype, bit[7] SID mark bit*/
    unsigned int mediaType:4; /* media type */
    unsigned int payloadLen:12; /* payload length */
    UINT32    timeStamp;   /* local or remote time stamp */
} __attribute__ ((packed)) XPacketHeader_t;

On the ingress side, the header information Remote Time Stamp, Payload Type and Marker bit are directly copied from a RTP packet. On the egress, the header information is filled by the DSP software except for the Payload Type of RFC 2833 event packets. The RTP processing module is responsible to determine the payload type if media type indicates a RFC 2833 tone-event packet.

7.2 Packet Delivery Mechanism

The packets are transferred between the DSP software and IP stack via the callback functions. The packet delivery module calls the callback function and passes the packet each time a packet is produced. The rules of using the callback function to deliver the packets include:

- The packet receiver registers a callback function with the packet deliverer.
- The packet deliverer is responsible to prepare the memory for the packet.
- The packet receiver has to copy the data to its internal buffer immediately in the callback function because the deliverer may reuse the same memory for the next packet (for example, the packet data may not be valid any more after the callback function returns).
- The packet receiver may perform some data processing in the callback function provided the execution of such processing is predictable (for example, the processing must be guaranteed to complete within a certain short period of time).

The following function is provided by the DSP software to receive the packets from the IP stack.

<table>
<thead>
<tr>
<th>XStatus_t xPacketReceive (UNIT16 channel, XPacket_t *buffer);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
</tr>
<tr>
<td>Input</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Output</td>
</tr>
<tr>
<td>Return</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

IP stack has to build the data packets from the IP packets it received and deliver them to DSP software by calling this function.

In egress direction, IP stack must provide a function to receive egress data packets. The DSP software calls the function each time a packet is generated. That function must be registered during initialization as described in Section 8.0.
8.0 Configuration and Initialization

The DSP software is configurable at initialization time, allowing the user to specify the HSS parameters, the number of resource instances to be created and the country-specific features. The user-supplied call back functions are also registered at that time.

8.1 System Configuration with HSS Interface

8.1.1 Description

This function performs the following procedures:

- Initialize and start HSS port
- Create TDM termination channels (for example, Network Endpoint resource instance) and link them to the HSS time slots sequentially. Error occurs if not enough time slots are enabled for all the TDM channels
- Create the IP terminations (for example, Decoder, Encoder, Tone Generator and Tone Detector resources)
- Create media service resources (for example, Player and Mixer)
- Enable country-specific call progress tones and set country-specific default parameters to the resources
- Register user-supplied call back functions

The configuration information in this function is defined as:

<table>
<thead>
<tr>
<th>Prototype</th>
<th>void xDspSysInit(XDSPSysConfig_t *pSysConfig);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>pSysConfig - System configuration information</td>
</tr>
<tr>
<td>Output</td>
<td>None</td>
</tr>
<tr>
<td>Return</td>
<td>None</td>
</tr>
</tbody>
</table>
The `pChanTsMap` field is an array that specifies how the instances of Network Endpoint are linked with the time slots of HSS. Each element of the array is defined as:

```c
typedef struct{
    int slotSample1; /* time slot of the 1st sample */
    int slotSample2; /* time slot of the 2nd sample, set to XCHAN_TDM_SLOT_NULL if narrowband */
} XDSPChanTdmSlots_t;
```

Assuming there are two channels – one wideband and one narrowband -- and the time slot locations for the channels in a 32-slot frame are shown as:

```
1 2 3 ... 17 18 ... 31
MSB LSB µ-law ...
```

The array that describes such configuration is given as:

```
1st WB sample 2nd WB sample
```

where:

```c
typedef XStatus_t (*XPktRcvFxn_t)(UINT16 channel, void *pPacket);
typedef int (*XMsgAgentDec_t)(XMsgRef_t pUsrMsg, XMsgRef_t pNativeMsg, int sequenceNo);
typedef void (*XMsgAgentEnc_t)(XMsgRef_t pUsrReply, XMsgRef_t pNativeReply, int sequenceNo, UINT8 usrMsgType);
```
If the `pChanTsMap` field is given a NULL pointer, all the instances of Network Endpoint will be configured to the narrowband mode and are linked to the active time slots sequentially.

**Warning:** This function must be called after downloading HSS NPE. An assertion occurs if any fatal errors happen (such as, memory exhausted) during the initialization. If the numbers of resources to be created are not specified correctly, the default ones are applied, which can be retrieved by the `xDspGetResConfig()` function.

### 8.2 Adding Tones to Tone Generator

#### Description

This function adds a new tone which can be played by the Tone Generator resources. Each new tone can contain one or more segments which is defined as:

```c
XDSPChanTdmSlots_t chanTdmSlotMap[3] =
{
    {1, 17}, /* channel 1 – WB, time slot 1 and 17 */
    {3, XCHAN_TDM_SLOT_NULL} /* channel 2 – NB, time slot 3 */
};
```

#### Prototype

```c
XStatus_t xBuildToneTG(UINT16 toneId, UINT16 numSegs, XTGToneSeg_t *pToneSegs, UINT32 *pErrCode);
```

#### Input

- `toneId` — Tone TD, must be in the range of 16 ~ 255
- `numSegs` — Number of segments of the tone
- `pToneSegs` — Array of tone segment definition

#### Output

- `pErrCode` — Error code if errors

#### Return

- `XSUCC` if successful
- Otherwise `XERROR`
**Warning:** New tone definition must be added during the initialization after `xDspSysInit()` or `xDspInit2()`. The pre-defined country-specific call progress tone is overwritten if a new tone is added with the same tone ID.

### 8.3 Change the DTMF Tone Parameters

**Prototype**

```c
typedef struct {
    UINT16  repCount;     /* Number of repetitions of the segment. 0 means to repeat forever */
    UINT16  segType;      /* signal type (single or dual frequency wave or AM wave) */
    UINT32  durationOn;   /* active duration in 1-ms unit. tone on duration in milli seconds */
    UINT32  durationOff;  /* silence duration in 1-ms unit. tone off duration in milli seconds*/
    INT16   freqA;        /* 1st frequency if single or dual frequency wave, the modulated carry frequency if AM wave, in 1Hz unit*/
    INT16   freqB;        /* 2nd frequency if dual frequency wave or the modulating frequency if AM wave, ignored if single frequency wave */
    INT16   ampA;         /* amplitude of frequency A above, (0~ - 45 in 1dBm unit) */
    INT16   ampB;         /* amplitude of frequency B if dual frequency wave, or modulation rate if AM wave (0~100 in 1% unit), ignored if single frequency wave */
    UINT16  mode;         /* mode, overwrite or mix over the Decoder output */
    INT16   nextSeg;      /* the index of next segment relative to the current segment. e.g., 1 means to go to the following segment, 0 means repeat the current segment, -2 means go back to previous 2 segments, XTG_LASTSEG means end-of-tone */
} XTGToneSeg_t;
```

**Input**

- `toneOn` - tone on duration in ms. Range 1 ~ FFFFFFF
- `toneOff` - tone off duration in ms. Range 1 ~ FFFFFFF
- `ampdBm` - total tone level in dBm, must be in 0 ~ -45 range

**Return**

- `XSUCC` if successful
- `XERROR` otherwise

### 8.3.1 Description

The DTMF tone generation has the default parameters of 100 ms tone-on and tone-off duration and -3dBm level. This function allows the users to change the default parameters.
8.4 Adding Tones to Tone Detector

8.4.1 Description

This function adds a criterion for the Tone Detector to detect a new tone. The criterion specifies the qualification ranges to a set of parameters defined as:

```c
/* segment data for tone detection template. */
typedef struct {
    UINT16   type;/* tone type (single or dual frequency tone) */
    UINT16   criteria;/* loose, medium or tight, use medium for normal case, use loose to get higher detection probability in poor SNR, use tight to get lower false detection probability in good SNR */
    UINT16   freqLowA;/* low bound of the 1st frequency in Hz */
    UINT16   freqHighA;/* high bound of the 1st frequency in Hz */
    UINT16   freqLowB;/* low bound of the 2nd frequency in Hz */
    UINT16   freqHighB;/* high bound of the 2nd frequency in Hz */
    INT16    ampLowA;/* low level of the 1st frequency in dBm */
    INT16    ampHighA;/* high level of the 1st frequency in dBm. If both low and high are set to 0, the default full range is applied */
    INT16    ampLowB;/* low level of the 2nd frequency in dBm */
    INT16    ampHighB;/* high level of the 2nd frequency in dBm. If both low and high are set to 0, the default full range is applied */
    UINT8    attributes;/* attribute (report the tone on, tone off or both on/off) */
} XTDToneInfo_t;
```

**Warning:** New tone detection criterion must be added during the initialization before xDspSysInit() or xDspInit2().

Prototype

```c
Status_t xBuildToneTD(UINT8 toneId, XTDToneInfo_t *pToneInfo, UINT32 *pErrCode);
```

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>toneId</td>
<td>Tone ID, must be in the range of 16 ~ 255</td>
</tr>
<tr>
<td>pToneInfo</td>
<td>Tone detection criterion information</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pErrCode</td>
<td>Error code if errors</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSUCC</td>
<td>if successful</td>
</tr>
<tr>
<td>Otherwise</td>
<td>XERROR</td>
</tr>
</tbody>
</table>
8.5   Amplitude Check in Tone Detection

8.5.1   Description

The Tone Detector is able to detect the pre-defined DTMF tones and fax tones in the full amplitude level range of +3 ~ -45 dBm. The applications can use this function to set a specific amplitude range. Only the signals within this amplitude range can detected as the DTMF or fax tones.

Warning: New tone detection criterion must be added during the initialization before xDspSysInit() or xDspInit2().

8.6   Getting DSP Resource Configuration and Routing Information

8.6.1   Description

The user’s applications can call this function any time after xDspSysInit() to obtain the DSP resource configuration and the stream IDs assigned to the T-Ports of each type of the resources. The data structure XDSPResConfig_t is defined as:

Prototype

| XStatus_t xSetAmplitudeRangeTD(int category, |
| int ampMinF0, int ampMaxF0, |
| int ampMinF1, int ampMaxF1) |

Input

| Category - Tone category to specify DTMF tones |
| ampMinF0 - Minimum amplitude of the low frequency, +3 ~ -45 in dBm |
| ampMaxF0 - Maximum amplitude of the low frequency, +3 ~ -45 in dBm |
| ampMinF1 - Minimum amplitude of the high frequency, +3 ~ -45 in dBm |
| ampMaxF1 - Maximum amplitude of the high frequency, +3 ~ -45 in dBm |

Output  None

Return  XSUCC if successful, otherwise XERROR

Prototype void xDspGetResConfig(XDSPResConfig_t *pCfgInfo)

Input  pCfgInfo - Pointer to DSP configuration data structure

Output  The resource configuration and the assignment of the routing streams

Return  None
The stream ID information is used for the application to connect the T-Ports and L-Ports of the resources. For a detailed explanation of T-Ports and L-Ports, refer to the Intel® Infrastructure DSP Solution Version 1.1 Programmer's Guide.

## 9.0 Complementary Functions

### 9.1 Direct Parameter Access

The user's applications can bypass the message and directly access the DSP parameters. This allows quicker access without having to send a message and receive a response. All parameters can be directly read but only some of them can be directly modified. The functions to access the parameters are:

```c
typedef struct{
    int numChTDM; /* number of TDM termination channels */
    int numChIP; /* number of IP termination channels */
    int numPlayers; /* number of player instances */
    int numMixers; /* number of Audio Mixers */
    int numPortsPerMixer; /* number of ports per mixer */
    int numStreams; /* number of total streams in the router */
    int streamBaseTDM; /* T-Port stream ID of the first TDM termination channel */
    int streamBaseIP; /* T-Port stream ID of the first IP termination channel */
    int streamBasePly; /* T-Port stream ID 1st port of the 1st Player instance */
    int streamBaseMix; /* T-Port stream ID of the first mixer port */
    int countryCode; /* country code */
} XDSPResConfig_t;
```

The prototype for accessing these parameters is:

```c
XStatus_t xDspParmRead(UINT8 res, UINT16 inst, UINT16 parmId, UINT16 *pParmVal);
```

**Input**

- `res` - DSP resource ID
- `inst` - Instance ID of the resource
- `parmId` - Parameter ID
- `pParmVal` - Pointer to the variable that receives the returned parameter value

**Output**

Parameter value

**Return**

- XSUCC if successful
- Otherwise XERROR

**Description**

This function retrieves the specified parameter value.
9.2 Direct Plug-in Parameter Access

These APIs are very similar to direct access APIs, as described in Section 9.1. These APIs provide an opportunity to directly access third-party plug-in component parameters. This helps applications to take advantage of additional capabilities supported by third-party plug-ins in addition to predefined component parameters.

**9.2.1 xDspParmWrite**

Prototype:

```c
XStatus_t xDspParmWrite(UINT8 res, UINT16 inst, UINT16 parmId, UINT16 parmVal, UINT32 transId);
```

**Input**

- `res` - DSP resource ID
- `inst` - Instance ID of the resource
- `parmId` - Parameter ID
- `parmVal` - Parameter value to be set
- `transId` - Transaction ID

**Output**

None

**Return**

- `XSUCC` if successful
- Otherwise `XERROR`

**Description**

This function sets the value of the specified parameter.

**9.2.2 xDspPlugInParmRead**

Prototype:

```c
XStatus_t xDspPlugInParmRead (UINT8 res, UINT16 inst, UINT16 algType, UINT16 parmId, UINT16 *pParmVal);
```

**Input**

- `res` - DSP resource ID
- `inst` - Instance ID of the resource
- `algType` - Algorithm type, e.g. `XCODER_TYPE_G729A`
- `parmId` - It is an index of mode item defined in USCI interface, where index value must be < `nModes` [Ref: USCI interface document for mode structures].
- `pParmVal` - Pointer to the variable that receives the returned parameter value

**Output**

Parameter value

**Return**

- `XSUCC` if successful
- Otherwise `XERROR`

**Description**

This function retrieves the specified parameter value for the ID of plug-in. Also note that `parmId` is not the same as that defined in direct access read/write APIs.

**9.2.3 xDspPlugInParmWrite**

Prototype:

```c
XStatus_t xDspPlugInParmWrite (UINT8 res, UINT16 inst, UINT16 algType, UINT16 parmId, UINT16 parmVal);
```

**Input**

- `res` - DSP resource ID
- `inst` - Instance ID of the resource
- `algType` - Algorithm type, e.g. `XCODER_TYPE_G729A`
- `parmId` - It is an index of mode item defined in USCI interface, where index value must be < `nModes` [Ref: USCI interface document for mode structures].
- `parmVal` - Parameter value to be set

**Output**

None

**Return**

- `XSUCC` if successful
- Otherwise `XERROR`

**Description**

This function sets the value of the specified parameter. Also note that `parmId` is not the same as that defined in direct access read/write APIs.
9.3 Flash Hook Detection

The hook states are defined as:

```c
typedef enum{
    XHOOK_STATE_ON = 0,
    XHOOK_STATE_OFF,
    XHOOK_STATE_FLASH
} XHookState_t;
```

### Prototype

```
Status_t xFlashHookDetect(UINT16 channel,
                          XHookState_t hookState,
                          XUINT32 transId);
```

### Input
- `channel` - Channel number starting from 1
- `hookState` - Hook state, `XHOOK_STATE_ON` or `XHOOK_STATE_OFF`
- `transId` - Transaction ID

### Output
None

### Return
- `XSUCC` if successful
- Otherwise `XERROR`

### Description
This function is called by the SLIC driver to report the hook state changes via the event message. If an on-hook transition is followed by an off-hook one within the time specified by the `XPARMID_NET_FLASH_HK` parameter, a flash hook event is reported.

9.4 Cache Prompt Registration

### Prototype

```
XMediaHandle_t xDspRegCachePrompt(XCachePromptDesc_t *pDesc);
```

### Input
- `pDesc` - The pointer to structure `XCachePromptDesc_t`

### Output
None

### Return
- `XMediaHandle` — Returns `XMEDIA_HANDLE_NULL` in the error case.

### Description
This function is called to register a cached prompt for playing at a later time. `XCachePromptDesc_t` describes the data required to register a cached prompt.

```c
typedef struct{
    UINT8    *pBuffer; /* Pointer to the play buffer. */
    INT32    size;    /* The size of play buffer. */
    XCoderType_t type; /* The type of data in play buffer. */
    // The valid types are
    XCODER_TYPE_G711MU_10MS,
    XCODER_TYPE_G711A_10MS and
    XCODER_TYPE_G729A */
} XCachePromptDesc_t;
```
9.5 Get Version Number

Prototype: char * xDspGetVersion(void);

Input: None

Output: None

Return: Pointer to the version string.

Description: This function returns a 8-digit version string in ASCII format hard coded in each release uniquely. The first 2 digits give the major version number, the 4 digits in the middle give the minor number and the last 2 digits give the build number.

9.6 External PCM Interface Synchronization

Prototype: void xDspPcmSync(int rxOffset, int txOffset)

Input:
- rxOffset - the current access index of the external PCM device in Rx direction
- txOffset - the current access index of the external PCM device in Tx direction

Output: None

Return: None

Description: The external device must call this function every frame period (10ms) when it passes the frame boundary in the data transfer buffers.

10.0 Constant Data

This section lists the definitions for constant data such as error codes and event codes.

10.1 Error Codes

Errors are reported via XMSG_ERROR message with an error code and two error data. The common error codes are defined as:

```c
#define XERR_SYSTEM 0x0001/* system error */
#define XERR_HSSIF 0x0002/* HSS interface error */
#define XERR_MEMORY 0x0003/* memory error */
#define XERR_INVALID_RES_ID 0x0011/* invalid resource id */
#define XERR_INVALID_CHAN_ID 0x0012/* invalid channel id */
#define XERR_INVALID_PARM_ID 0x0013/* invalid parameter id */
#define XERR_INVALID_STREAM_ID 0x0014/* invalid stream id */
#define XERR_PARM_READONLY 0x0015/* read only parameter */
#define XERR_PARM_SET_FAIL 0x0016/* cannot set parameter */
#define XERR_PARM_GET_FAIL 0x0017/* cannot get parameter */
#define XERR_UNEXPECTED_MSG 0x0018/* unexpected message */
#define XERR_UNSUPPORTED_MSG 0x0019/* unsupported message */
#define XERR_ALGORITHM 0x0041/* algorithm related error */
#define XERR_OTHERS 0x00ff/* other errors */
```

The resource-specific error codes are defined as:
#define XERR_INVALID_CODE_TYPE 0x401/* invalid codec type */
#define XERR_INVALID_FPP 0x402/* invalid # frms per pkt */
#define XERR_TG_INVALID_TONE_ID 0x403/* invalid tone ID */
#define XERR_TG_INVALID_TID_NUM 0x404/* too many tone IDs */
#define XERR_TG_INVALID_DATA_NUM 0x405/* too many FSK data */
#define XERR_TD_INVALID_DIGIT_NUM 0x406/* too many digits */
#define XERRRESOURCE_BUSY 0x407/* resource is busy */
#define XERRRESOURCE_IDLE 0x408/* resource is idle */
#define XERR_MA_DEEP_RECURSIVE 0x409/* deep recursive msg decoder*/
#define XERR_MA_MSG_DECORDER 0x40a/* message decoding fail */

## 10.2 Event Codes

Events are reported via `XMSG_EVENT` message with an event code and two event data. The resource specific event codes are defined as:

#define XEVT_CODE_TD_TONEON 0x101/* tone-on event */
#define XEVT_CODE_TD_TONEOFF 0x102/* tone-off event */
#define XEVT_LOST_PACKET 0x103/* lost packet */
#define XEVT_DEC_PACKET_CHNG 0x104/* RTP payload type changed */
#define XEVT_NET_HOOK_STATE 0x105/* hook state change detected */
#define XEVT_NET_TIMER 0x106/* timer expired */
#define XEVT_NET_SIN_PHASEREV_YES 0x107/* 2100hz tone with phase reversal every 450ms detected on Tx path */
#define XEVT_NET_RIN_PHASEREV_YES 0x108/* 2100hz tone with phase reversal every 450ms detected on Rx path */
#define XEVT_NET_SIN_ECENABLED 0x109/* silence detected on Tx path, indication to enable EC */
#define XEVT_NET_RIN_ECENABLED 0x10a/* silence detected on Rx path, indication to enable EC */
#define XEVT_NET_T30PREAMBDET_SIN 0x10b/* T30 preamble detected on Tx path */
#define XEVT_NET_T30PREAMBDET_RIN 0x10c/* T30 preamble detected on Rx path */

## 10.3 Tone IDs

The DTMF tone IDs used by the Tone Generator and Detector are defined as:
Fax-tone IDs reported by the Tone Detector for fax bypass applications. Not supported by the Tone Generator.

```
#define RFC_TID_DTMF_0 0
#define RFC_TID_DTMF_1 1
#define RFC_TID_DTMF_2 2
#define RFC_TID_DTMF_3 3
#define RFC_TID_DTMF_4 4
#define RFC_TID_DTMF_5 5
#define RFC_TID_DTMF_6 6
#define RFC_TID_DTMF_7 7
#define RFC_TID_DTMF_8 8
#define RFC_TID_DTMF_9 9
#define RFC_TID_DTMF_STAR 10
#define RFC_TID_DTMF_POUND 11
#define RFC_TID_DTMF_A 12
#define RFC_TID_DTMF_B 13
#define RFC_TID_DTMF_C 14
#define RFC_TID_DTMF_D 15
```

The general call-progress tone IDs used by the Tone Generator are defined as:

```
#define RFC_TID_FAX_CED 32
#define RFC_TID_FAX_CNG 36
#define RFC_TID_FAX_V21 40
```
Currently only the following specific call progress tones are supported for tone generation:

- China (People’s Republic of China)
- Japan
- United States

Japan country code and pre-defined call progress tones are as follows:

```c
#define COUNTRY_CODE_JP 81 /* country code for Japan */
#define NTT_TID_DT RFC_TID_DIAL /* dial tone */
#define NTT_TID_RBT RFC_TID_RING /* ring back tone */
#define NTT_TID_BT RFC_TID_BUSY /* busy tone */
#define NTT_TID_PDT RFC_TID_PBX_DIAL /* private dial tone */
#define NTT_TID_SDT RFC_TID_2ND_DIAL /* 2nd dial tone */
#define NTT_TID_CPT RFC_TID_POS_IND /* acceptance tone */
#define NTT_TID_HST RFC_TID_HOLD /* hold service tone */
#define NTT_TID_IIT RFC_TID_CALL_WT /* incoming id tone */
#define NTT_TID_SIIT 110 /* special incoming id tone */
#define NTT_TID_HOW RFC_TID_OFFHK_WARN /* howler tone */
```

United States country code and pre-defined call progress tones are as follows:
China country code and pre-defined call progress tones are as follows:

```c
#define COUNTRY_CODE_US 1  /**< US country code */
#define US_TID_DIAL RFC_TID_DIAL  /**< dial tone */
#define US_TID_RING RFC_TID_RING  /**< ring back tone */
#define US_TID_BUSY RFC_TID_BUSY  /**< busy tone */
#define US_TID_RC.Dial RFC_TID_SP.Dial  /**< recall dial tone */
#define US_TID_PBX.Dial RFC_TID_PBX.Dial  /**< PBX dial tone */
#define US_TID_CONGESTION RFC_TID_CONGESTION  /**< congestion tone */
#define US_TID_CALL.WT RFC_TID_CALL.WT  /**< call waiting tone */
#define US_TID_WARN.OPER 110  /**< operator intervening tone */
```

```c
#define COUNTRY_CODE_PRC 86  /**< China country code */
#define PRC_TID_DIAL RFC_TID_DIAL  /**< dial tone */
#define PRC_TID_RING RFC_TID_RING  /**< ring back tone */
#define PRC_TID_BUSY RFC_TID_BUSY  /**< busy tone */
#define PRC_TID_SP.Dial RFC_TID_SP.Dial  /**< special dial tone */
#define PRC_TID_CONGESTION RFC_TID_CONGESTION  /**< congestion tone */
#define PRC_TID_UNAVAILABLE RFC_TID_UNAVAILABLE  /**< unavailable tone */
#define PRC_TID_TOLL RFC_TID_COMFORT  /**< long distance tone */
#define PRC_TID_QUEUE RFC_TID_QUEUE  /**< queue tone */
#define PRC_TID_CALL.WT RFC_TID_CALL.WT  /**< call waiting tone */
#define PRC_TID_THR.PARTY RFC_TID_THR.PARTY  /**< 3 party remind tone */
#define PRC_TID_CONFIRMATION RFC_TID_CONFIRMATION  /**< confirmation tone */
#define PRC_TID_OFFHK.Warn RFC_TID_OFFHK_WARN  /**< howler tone */
```

10.4 Other Constants

The coder types used in the `XPARMID_DEC_CTYPE` and `XPARMID_ENC_CTYPE` parameters and the `XMSG_CODER_START` message are defined as:

```c
typedef enum{
    XCORDER_TYPE_PASSTHRU = 0,
    XCORDER_TYPE_G711MU_10MS,
    XCORDER_TYPE_G711A_10MS,
    XCORDER_TYPE_G729A,
    XCORDER_TYPE_G723,
    XCORDER_TYPE_G722,
    XCORDER_TYPE_G726_40,
    XCORDER_TYPE_G726_32,
    XCORDER_TYPE_G726_24,
    XCORDER_TYPE_G726_16,
    XCORDER_TYPE_G729 = 17,
    XCORDER_TYPE_UNDEF = -1
} XCoderType_t;
```

Mask bits used to specify the coder type subset in Decoder auto-switch parameter are defined as:
The mask bits used to specify the termination digits in the `XMSG_TD_RCV` message are defined as:

```c
#define XPARM_DEC_AUTOSW_OFF 0x0000
#define XPARM_DEC_AUTOSW_G711MU 0x0001
#define XPARM_DEC_AUTOSW_G711A 0x0002
#define XPARM_DEC_AUTOSW_G729A 0x0004
#define XPARM_DEC_AUTOSW_G723 0x0008
#define XPARM_DEC_AUTOSW_G722 0x0010
#define XPARM_DEC_AUTOSW_G726_40 0x0020
#define XPARM_DEC_AUTOSW_G726_32 0x0040
#define XPARM_DEC_AUTOSW_G726_24 0x0080
#define XPARM_DEC_AUTOSW_G726_16 0x0100
#define XPARM_DEC_AUTOSW_ALL 0xffff
```

The stop-reasons in the `XMSG_TG_PLAY_CMPLT`, `XMSG_TD_RCV_CMPLT`, `XMSG_TD_RCV_FSK_CMPLT`, and `XMSG_PLY_CMPLT` messages are defined as:

```c
#define XPARM_DEC_AUTOSW_OFF 0x0000
#define XPARM_DEC_AUTOSW_G711MU 0x0001
#define XPARM_DEC_AUTOSW_G711A 0x0002
#define XPARM_DEC_AUTOSW_G729A 0x0004
#define XPARM_DEC_AUTOSW_G723 0x0008
#define XPARM_DEC_AUTOSW_G722 0x0010
#define XPARM_DEC_AUTOSW_G726_40 0x0020
#define XPARM_DEC_AUTOSW_G726_32 0x0040
#define XPARM_DEC_AUTOSW_G726_24 0x0080
#define XPARM_DEC_AUTOSW_G726_16 0x0100
#define XPARM_DEC_AUTOSW_ALL 0xffff
```

```c
#define XMSG_STOP_REASON_USER 1/* stopped by XMSG_STOP message */
#define XMSG_STOP_REASON_EOD 2/* end of data */
#define XMSG_STOP_REASON_TERM 3/* stopped by the terminate digits */
#define XMSG_STOP_REASON_TIMEOUT 4/* time out */
#define XMSG_STOP_REASON_ERROR 5/* general error */
```
Appendix A : HSS Driver APIs

The following APIs are provided by the HSS driver.

<table>
<thead>
<tr>
<th>API</th>
<th>Functions</th>
</tr>
</thead>
</table>
| IxHssDriverHssPortInit(hss_port_config) | • Downloads NPE microcode image corresponding to npe_image_id provided in hss_port_config structure and initializes NPE A  
• Initializes the HSS device  
• Configures HSS port for the channelized service  
• Connects the configured port to the HSS driver |
| IxHssDriverNpeBCInit(npe_b_image_id, npe_c_image_id) | This API downloads the NPE images to NPE B and C and starts the corresponding NPE. For example, IxHssDriverNpeBCInit (IX_NPEDL_NPEIMAGE_NPEB_ETH, IX_NPEDL_NPEIMAGE_NPEC_ETH) downloads NPE Ethernet image to NPE B and C and starts NPE B and C devices. |
| IxHssDriverNpeCInit( UINT32 npeC_image_id) | This API downloads the NPE image to NPE C and starts the NPE. For example, IxHssDriverNpeCInit (IX_NPEDL_NPEIMAGE_NPEC_ETH) downloads NPE Ethernet image to NPE C and starts NPE C device. |

HSS port configuration information has to be provided using the structure:

```c
typedef struct {
    IxHssAccHssPort hssport_Id;
    IxHssAccTdmSlotUsage
        hssTdmMap[IX_HSSACC_TSLOTS_PER_HSS_PORT];
    IxHssAccConfigParams hssportTx_Rx_config;
    UINT32 hssNpeImageid;
}Hss_config;
```

Appendix B : SLIC Driver APIs

SLIC driver module comprises the following parts:
1. A core, which is a legacy of earlier DSP software releases.
2. A user space adaptor (IxSlicModuleSymbols.c) defined in kernel space.

The user space adaptor part offers only a subset of the core driver functions sufficient for the Codelet applications and you are free to extend it (IxSlicModuleSymbols.c) to meet your specific needs.

The following APIs are available for accessing the functionalities of SLIC. The user commands get translated to the kernel space through the "ioctl" call, and the prototype is as follows:

```c
ioctl (struct *inode, struct file, unsigned int cmd, unsigned long arg)
```
The status from the SLIC after execution of a command is intimated to users space application through copy_to_user() call.

The interrupt driven events such as off-hook and on-hook status of SLIC are handled by using the callback functions. The basic mechanism followed in handling the call back function is as below:

1. Define a kernel callback function WaitForSlic eventCallback with respect to the SLIC events. So, when on-hook and off-hook transition happens in SLIC, the state and POTS line information will be available inside this function as this function is already registered inside the kernel space.

2. Define another function, called slicdummyCallback in kernel space, which will get the information of SLIC events and finally the events are reported to user space by using copy_to_user function.

### ixFXSStartRinging(int slot, intboard, int chip)

**Description**
This function allows clients to apply a ringing signal on a particular FXS chip.

**Input**
- slot - The index number of possible slots supported. (Note: Only HSS Slot 0 is validated and supported.)
  - 0: HSS Slot 0
  - 1: HSS Slot 1.
- board - The number of boards supported. Intel® IXDP465 Development Platform has a mezzanine interface that supports 8 voice modules. The DSP software is validated with one voice module. This parameter is ignored and should set as 0 for Intel® IXP435 Multi-Service Residential Gateway Reference Platform and Intel® IXDP425 Network Gateway Development Platform.
  - 0: board1
  - 1: board2
  - 2: board3
  - 3: board4
  - 4: board5
  - 5: board6
  - 6: board7
  - 7: board8
- chip - The following FXS chips are supported by the board.
  - 0-1: FXS1-FXS2 for IXP435 reference platform, 0-3: FXS1-FXS4 for IXDP465 development platform

**Return**
- zero - if the function executes successfully
- non zero - if the function does not execute successfully
### ixFXSStopRinging(int slot, int board, int chip)

**Description**
This function allows clients to send a "stop ringing signal" to a particular FXS chip.

**Input**
- slot - The index number of possible slots supported. (Note: Only HSS Slot 0 is validated and supported.)
  - 0: HSS Slot 0
  - 1: HSS Slot 1.
- board - The number of boards supported. IXDP465 development platform has a mezzanine interface that supports 8 voice modules. The DSP software is validated with one voice module. This parameter is ignored and should set as 0 for IXP435 reference platform and IXDPG425 platform.
  - 0: board1
  - 1: board2
  - 2: board3
  - 3: board4
  - 4: board5
  - 5: board6
  - 6: board7
  - 7: board8
- chip - The following FXS chips are supported by the board.
  - 0-1: FXS1-FXS2 for IXP435 reference platform, 0-3: FXS1-FXS4 for IXDP465 development platform

**Return**
- zero - if the function executes successfully
- non zero - if the function does not execute successfully

### ixFXSHookStatus(int slot, int board, int chip)

**Description**
This function allows clients to read the hook status of the FXS chips used in the board.

**Input**
- slot - The index number of possible slots supported. (Note: Only HSS Slot 0 is validated and supported.)
  - 0: HSS Slot 0
  - 1: HSS Slot 1.
- board - The number of boards supported. The IXDP465 development platform has a mezzanine interface that supports 8 voice modules. The DSP software is validated with one voice module. This parameter is ignored and should set as 0 for IXP435 reference platform and IXDPG425 platform.
  - 0: board1
  - 1: board2
  - 2: board3
  - 3: board4
  - 4: board5
  - 5: board6
  - 6: board7
  - 7: board8
- chip - The following FXS chips are supported by the board.
  - 0-1: FXS1-FXS2 for IXP435 reference platform, 0-3: FXS1-FXS4 for IXDP465 development platform

**Return**
- zero - if the function executes successfully
- non zero - if the function does not execute successfully
### ixFXSGetPCMConfig(int slot, int board, int chip)

**Description**
This function allows clients to read PCM configuration of the FXS chip.

**Input**
- **slot** - The index number of possible slots supported. (Note: Only HSS Slot 0 is validated and supported.)
  - 0: HSS Slot 0
  - 1: HSS Slot 1.
- **board** - The number of boards supported. IXDP465 development platform has a mezzanine interface that supports 8 voice modules. The DSP software is validated with one voice module. This parameter is ignored and should set as 0 for IXP435 reference platform and IXDP425 platform.
  - 0: board1
  - 1: board2
  - 2: board3
  - 3: board4
  - 4: board5
  - 5: board6
  - 6: board7
  - 7: board8
- **chip** - The following FXS chips are supported by the board.
  - 0-1: FXS1-FXS2 for IXP435 reference platform, 0-3: FXS1-FXS4 for IXDP465 development platform

**Return**
- zero - if the function executes successfully
- non zero - if the function does not execute successfully

### ixFXSOChipDetails(int slot, int board, int chip)

**Description**
This function allows clients to read the make and manufacture of the FXO/FXS device. The chips can be Si3216, Si3210 or Si3050. This will print the chip details to the output terminal.

**Input**
- **slot** - The index number of possible slots supported. (Note: Only HSS Slot 0 is validated and supported.)
  - 0: HSS Slot 0
  - 1: HSS Slot 1.
- **board** - The number of boards supported. IXDP465 development platform has a mezzanine interface that supports 8 voice modules. The DSP software is validated with one voice module. This parameter is ignored and should set as 0 for IXP435 reference platform and IXDP425 platform.
  - 0: board1
  - 1: board2
  - 2: board3
  - 3: board4
  - 4: board5
  - 5: board6
  - 6: board7
  - 7: board8
- **chip** - The following FXS and FXO chips are supported by the board.
  - 0-1: FXS1-FXS2 for IXP435 reference platform, 0-3: FXS1-FXS4 for IXDP465 development platform
  - 30: FXO1 for IXP435 reference platform, 4: FXO1 for IXDP465 development platform

**Return**
- zero - if the function executes successfully
- non zero - if the function does not execute successfully
### ixFXSODirectRegRead(int slot, int board, int chip, int reg)

**Description**
This function allows clients to read from the direct registers of the FXS/FXO device. Register 1-59 in case of FXO and Register 1-108 for FXS.

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>slot - The index number of possible slots supported. (Note: Only HSS Slot 0 is validated and supported.)</td>
<td></td>
</tr>
<tr>
<td>0: HSS Slot 0</td>
<td></td>
</tr>
<tr>
<td>1: HSS Slot 1.</td>
<td></td>
</tr>
<tr>
<td>board - The number of boards supported. IXDP465 development platform has a mezzanine interface that supports 8 voice modules. The DSP software is validated with one voice module. This parameter is ignored and should set as 0 for IXP435 reference platform and IXDPG425 platform.</td>
<td></td>
</tr>
<tr>
<td>0: board1</td>
<td></td>
</tr>
<tr>
<td>1: board2</td>
<td></td>
</tr>
<tr>
<td>2: board3</td>
<td></td>
</tr>
<tr>
<td>3: board4</td>
<td></td>
</tr>
<tr>
<td>4: board5</td>
<td></td>
</tr>
<tr>
<td>5: board6</td>
<td></td>
</tr>
<tr>
<td>6: board7</td>
<td></td>
</tr>
<tr>
<td>7: board8</td>
<td></td>
</tr>
<tr>
<td>chip - The following FXS and FXO chips are supported by the board.</td>
<td></td>
</tr>
<tr>
<td>0-1: FXS1-FXS2 for IXP435 reference platform, 0-3: FXS1-FXS4 for IXDP465 development platform</td>
<td></td>
</tr>
<tr>
<td>30: FXO1 for IXP435 reference platform, 4: FXO1 for IXDP465 development platform</td>
<td></td>
</tr>
<tr>
<td>reg - Reads from the direct registers of FXS/FXO device. Register 1-59 in case of FXO and Register 1-108 for FXS.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero - if the function executes successfully</td>
<td></td>
</tr>
<tr>
<td>non zero - if the function does not execute successfully</td>
<td></td>
</tr>
</tbody>
</table>
# ixFXSODirectRegWrite(int slot, int board, int chip, int reg, int data)

This function allows clients to write to the direct registers of the FXS/FXO device. Register 1-59 in case of FXO and Register 1-108 for FXS.

## Description

- **slot** - The index number of possible slots supported. (Note: Only HSS Slot 0 is validated and supported.)
  - 0: HSS Slot 0
  - 1: HSS Slot 1.
- **board** - The number of boards supported. IXDP465 development platform has a mezzanine interface that supports 8 voice modules. The DSP software is validated with one voice module. This parameter is ignored and should set as 0 for IXP435 reference platform and IXDP425 platform.
  - 0: board1
  - 1: board2
  - 2: board3
  - 3: board4
  - 4: board5
  - 5: board6
  - 6: board7
  - 7: board8
- **chip** - The following FXS and FXO chips are supported by the board.
  - 0-1: FXS1-FXS2 for IXP435 reference platform, 0-3: FXS1-FXS4 for IXDP465 development platform
  - 30: FXO1 for IXP435 reference platform, 4: FXO1 for IXDP465 development platform
- **reg** - Reads from the direct registers of FXS/FXO device. Register 1-59 in case of FXO and Register 1-108 for FXS.
- **data** - This is the data to be written into a given register.

## Input

- slot - The index number of possible slots supported. (Note: Only HSS Slot 0 is validated and supported.)
  - 0: HSS Slot 0
  - 1: HSS Slot 1.
- board - The number of boards supported. IXDP465 development platform has a mezzanine interface that supports 8 voice modules. The DSP software is validated with one voice module. This parameter is ignored and should set as 0 for IXP435 reference platform and IXDP425 platform.
  - 0: board1
  - 1: board2
  - 2: board3
  - 3: board4
  - 4: board5
  - 5: board6
  - 6: board7
  - 7: board8
- chip - The following FXS and FXO chips are supported by the board.
  - 0-1: FXS1-FXS2 for IXP435 reference platform, 0-3: FXS1-FXS4 for IXDP465 development platform
  - 30: FXO1 for IXP435 reference platform, 4: FXO1 for IXDP465 development platform
- reg - Reads from the direct registers of FXS/FXO device. Register 1-59 in case of FXO and Register 1-108 for FXS.
- data - This is the data to be written into a given register.

## Return

- **zero** - if the function executes successfully
- **non zero** - if the function does not execute successfully
### ixFXSOSetDeviceLoopback(int slot, int board, int chip)

**Description**
This function sets the device FXS/FXO into digital loop back mode.

**Input**
- **slot** - The index number of possible slots supported. (Note: Only HSS Slot 0 is validated and supported.)
  - 0: HSS Slot 0
  - 1: HSS Slot 1.
- **board** - The number of boards supported. IXDP465 development platform has a mezzanine interface that supports 8 voice modules. The DSP software is validated with one voice module. This parameter is ignored and should set as 0 for IXP435 reference platform and IXDPG425 platform.
  - 0: board1
  - 1: board2
  - 2: board3
  - 3: board4
  - 4: board5
  - 5: board6
  - 6: board7
  - 7: board8
- **chip** - The following FXS and FXO chips are supported by the board.
  - 0-1: FXS1-FXS2 for IXP435 reference platform, 0-3: FXS1-FXS4 for IXDP465 development platform
  - 30: FXO1 for IXP435 reference platform, 4: FXO1 for IXDP465 development platform

**Return**
- zero - if the function executes successfully
- non zero - if the function does not execute successfully

### ixFXSORESetDeviceLoopback(int slot, int board, int chip)

**Description**
This function disables the digital loop back mode of the FXS/FXO device.

**Input**
- **slot** - The index number of possible slots supported. (Note: Only HSS Slot 0 is validated and supported.)
  - 0: HSS Slot 0
  - 1: HSS Slot 1.
- **board** - The number of boards supported. IXDP465 development platform has a mezzanine interface that supports 8 voice modules. The DSP software is validated with one voice module. This parameter is ignored and should set as 0 for IXP435 reference platform and IXDPG425 platform.
  - 0: board1
  - 1: board2
  - 2: board3
  - 3: board4
  - 4: board5
  - 5: board6
  - 6: board7
  - 7: board8
- **chip** - The following FXS and FXO chips are supported by the board.
  - 0-1: FXS1-FXS2 for IXP435 reference platform, 0-3: FXS1-FXS4 for IXDP465 development platform
  - 30: FXO1 for IXP435 reference platform, 4: FXO1 for IXDP465 development platform

**Return**
- zero - if the function executes successfully
- non zero - if the function does not execute successfully
### ixFXSOGetDeviceLoopback(int slot, int board, int chip)

**Description**  
This function gets the loop back mode status of the FXS/FXO device.

**Input**
- **slot** - The index number of possible slots supported. (Note: Only HSS Slot 0 is validated and supported.)
  - 0: HSS Slot 0
  - 1: HSS Slot 1.
- **board** - The number of boards supported. IXDP465 development platform has a mezzanine interface that supports 8 voice modules. The DSP software is validated with one voice module. This parameter is ignored and should set as 0 for IXP435 reference platform and IXDPG425 platform.
  - 0: board1
  - 1: board2
  - 2: board3
  - 3: board4
  - 4: board5
  - 5: board6
  - 6: board7
  - 7: board8
- **chip** - The following FXS and FXO chips are supported by the board.
  - 0-1: FXS1-FXS2 for IXP435 reference platform, 0-3: FXS1-FXS4 for IXDP465 development platform
  - 30: FXO1 for IXP435 reference platform, 4: FXO1 for IXDP465 development platform

**Return**
- zero - if the function executes successfully
- non zero - if the function does not execute successfully

### ixFXSOVerifyComm(int slot, int board, int chip)

**Description**  
This function verifies the communication with the chip (FXS/FXO).

**Input**
- **slot** - The index number of possible slots supported. (Note: Only HSS Slot 0 is validated and supported.)
  - 0: HSS Slot 0
  - 1: HSS Slot 1.
- **board** - The number of boards supported. IXDP465 development platform has a mezzanine interface that supports 8 voice modules. The DSP software is validated with one voice module. This parameter is ignored and should set as 0 for IXP435 reference platform and IXDPG425 platform.
  - 0: board1
  - 1: board2
  - 2: board3
  - 3: board4
  - 4: board5
  - 5: board6
  - 6: board7
  - 7: board8
- **chip** - The following FXS and FXO chips are supported by the board.
  - 0-1: FXS1-FXS2 for IXP435 reference platform, 0-3: FXS1-FXS4 for IXDP465 development platform
  - 30: FXO1 for IXP435 reference platform, 4: FXO1 for IXDP465 development platform

**Return**
- zero - if the function executes successfully
- non zero - if the function does not execute successfully
### `ixFXSSetPCMConfig(int slot, int board, int chip, int pcmed, int mode, int pcmtfr)`

**Description**

This function sets the PCM configuration of the FXS device. The configuration can be PCM enabled/disabled, A law/µ law and Narrow band/Wide band. This is an optional feature. If SLIC is already initialized, then this is not required.

**Input**

- **slot** - The index number of possible slots supported. (Note: Only HSS Slot 0 is validated and supported.)
  - 0: HSS Slot 0
  - 1: HSS Slot 1.
- **board** - The number of boards supported. IXDP465 development platform has a mezzanine interface that supports 8 voice modules. The DSP software is validated with one voice module. This parameter is ignored and should set as 0 for IXP435 reference platform and IXDPG425 platform.
  - 0: board1
  - 1: board2
  - 2: board3
  - 3: board4
  - 4: board5
  - 5: board6
  - 6: board7
  - 7: board8
- **chip** - The following FXS chips are supported by the board.
  - 0-1: FXS1-FXS2 for IXP435 reference platform, 0-3: FXS1-FXS4 for IXDP465 development platform
- **pcmed** - PCM enabled/disabled.
  - PCM enabled
  - PCM disabled
- **mode**
  - 0: A law
  - 1: µ law
  - 2: Reserved
  - 3: Linear
- **pcmtfr**
  - 0: 8 bits
  - 1: 16 bits transfer

**Return**

- zero - if the function executes successfully
- non zero - if the function does not execute successfully

### `ixFXSOSlicInit(int mode, int band)`

**Description**

This function initializes both FXS and FXO.

**Input**

- **mode** - This is the companding mode.
  - 0: A law
  - 1: µ law
  - 2: Linear
- **band**
  - 0: Narrow band
  - 1: Wide band.

**Return**

- IX_ANALOG_SUCCESS - if the function executes successfully
- IX_ANALOG_FAIL - if the function does not execute successfully
### ixFSXRegCallback(int chip)

**Description**
This function registers the callback functions for the FXS chips.

**Input**

- **chip** - The following FXS chips are supported by the board.
  - 0-1: FXS1-FXS2 for IXP435 reference platform, 0-3: FXS1-FXS4 for IXDP465 development platform
  - 30: FXO1 for IXP435 reference platform, 4: FXO1 for IXDP465 development platform

**Return**

- zero - if the function executes successfully
- non zero - if the function does not execute successfully

### ixFXOOffhook(int slot, int board, int chip)

**Description**
This function allows clients to apply an Off-Hook signal on a particular SI3050 chip in the IXDP465 development platform and IXP435 reference platform.

**Input**

- **slot** - The index number of possible slots supported. (Note: Only HSS Slot 0 is validated and supported.)
  - 0: HSS Slot 0
  - 1: HSS Slot 1.
- **board** - The number of boards supported. IXDP465 development platform has a mezzanine interface that supports 8 voice modules. The DSP software is validated with one voice module. This parameter is ignored and should set as 0 for IXP435 reference platform and IXDP425 platform.
  - 0: board1
  - 1: board2
  - 2: board3
  - 3: board4
  - 4: board5
  - 5: board6
  - 6: board7
  - 7: board8
- **chip** - The following FXO chip is supported by the board.
  - 30: FXO1 for IXP435 reference platform, 4: FXO1 for IXDP465 development platform

**Return**

- zero - if the function executes successfully
- non zero - if the function does not execute successfully
### ixFXOOnhook(int slot, int board, int chip)

**Description**
This function allows clients to perform On-Hook on a particular SI3050 chip in IXDP465 development platform and IXP435 reference platform.

**Input**
- `slot` - The index number of possible slots supported. (Note: Only HSS Slot 0 is validated and supported.)
  - 0: HSS Slot 0
  - 1: HSS Slot 1.
- `board` - The number of boards supported. IXDP465 development platform has a mezzanine interface that supports 8 voice modules. The DSP software is validated with one voice module. This parameter is ignored and should set as 0 for IXP435 reference platform and IXDPG425 platform.
  - 0: board1
  - 1: board2
  - 2: board3
  - 3: board4
  - 4: board5
  - 5: board6
  - 6: board7
  - 7: board8
- `chip` - The following FXO chip is supported by the board.
  - 30: FXO1 for IXP435 reference platform, 4: FXO1 for IXDP465 development platform

**Return**
- `0` - if the function executes successfully
- `non 0` - if the function does not execute successfully

### ixFXORingDetectOn(void)

**Description**
This function allows clients to enable Ring Detection on the SI3050 chip in IXP435 reference platform.

**Input**
- No input required

**Return**
- `0` - if the function executes successfully
- `non 0` - if the function does not execute successfully

### ixFXORingDetectOff(void)

**Description**
This function allows clients to disable Ring Detection on the SI3050 chip in IXP435 reference platform.

**Input**
- No input required

**Return**
- `0` - if the function executes successfully
- `non 0` - if the function does not execute successfully
The following are the list of SLIC callback events available:

1. Hook Event from FXS (both OnHook and OffHook)
2. Ring Detection Event from FXO port (if Ring Detection for FXO is enabled)

You can configure FXS or FXO port to work with other components to perform different operations, such as Caller-ID generation and detection. The following is an example to demonstrate the procedure to test FSK data generation and FSK data detection through FXS and FXO port of IXP435 reference platform using codelets application. FSK generation and detection is commonly used for Caller-ID application.

2. Set up FXS-FXO local board loopback with RJ11 cable connecting FXS port 0 (channel 1) and FXO port (channel 3).
3. Start the codelets application with US country code, u-law companding mode, and narrow band.
4. For socket configuration, set up the default configuration.
5. Channel setup for all channels with G.711 u-law.
6. Using SLIC APIs menu, enable FXS Onhook Transmission for FXS chip number 0 (FXS port 0). This will enable the transmission of FSK data during On-hook condition.
7. Using Diagnostic Menu, enable Receive FSK for FXO port (channel 3). You need to specify sufficient time (example: 180 seconds) to set up the next step.
8. Using Diagnostic Menu, Play FSK data to instance 1 (FXS port 0). Specify a set of ASCII string as test data to be send from FXS port 0 to FXO port. FXO port will receive the sent data in Hex.