Intel® IXP400 Digital Signal Processing (DSP) Software Version 2.6.2

API Reference Manual

February 2005

Document Number: 273811, Revision: 008
February 2005

Contents

Figures

Revision History

Date	Revision	Description
February 2005	008	Updates for the release of Intel® IXP400 DSP Software v2.6.2.
September 2004	007	Further updates for the release of Intel® IXP400 DSP Software v2.5. Change bars indicate areas of change.
June 2004	006	Updates for the release of Intel® IXP400 DSP Software v2.5.
January 2004	005	Updates for the release of Intel® IXP400 DSP Software Version 2.4.
September 2003	004	Clarified input for `XStatus_t xMsgReceive` message function.
September 2003	003	Updates for the release of Intel® IXP400 DSP Software Version 2.3
March 2003	002	Added minor updates to represent features of Intel® IXP400 DSP Software Version 1.1.
January 2003	001	First release of this document.

This page is intentionally left blank.

1.0 Introduction

The Intel® IXP400 DSP Software v2.6.2 Release is a software module that provides the basic voice processing functionalities for VoIP residential gateway applications. It can be viewed as a completed media processing layer with control and data interfaces as its API.

This document defines the API specifications.

1.1 General

The Intel® IXP400 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, etc., as required in any IP media gateway or real-time media streaming functionalities.

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

It additionally describes the data interface and format as well as message and data delivery mechanisms.

1.2 Scope

The interface of Intel® IXP400 DSP Software is a set of functions, macros, and 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

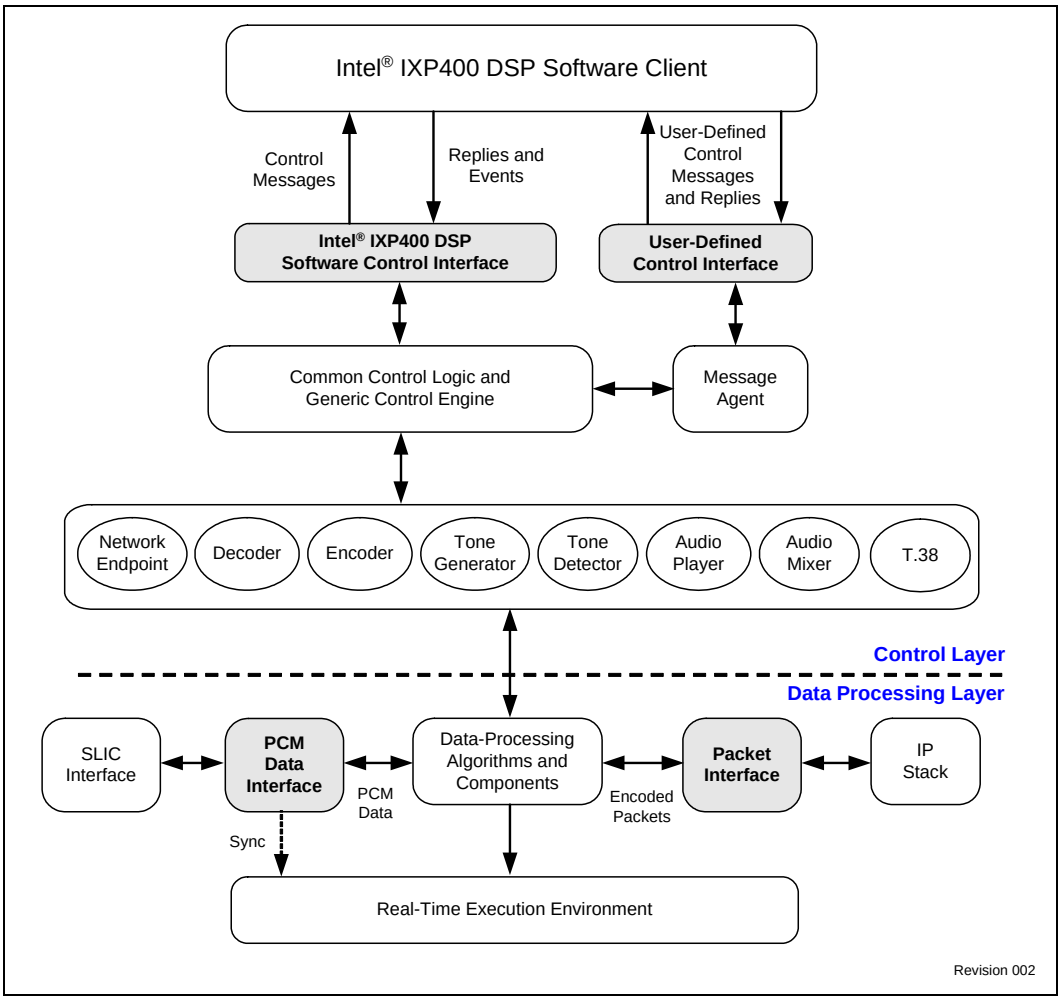
AGC	Automatic Gain Control for voice data towards IP network
ALC	Automatic Level Control
CPE	Consumer Premise Equipment
EC	Echo Cancellation
FEC	Forward Error Correction
FSK	Frequency Shift Keying
IP	Internet Protocol
ISR	Interrupt Service Routine
NLP	Non-linear Processing (for EC)
SP	Signal Processing
VAD	Voice Activity Detection

2.0 Architectural Overview

Intel® IXP400 DSP Software is implemented as an independent module having its own tasks and runtime environment. The software architecture is of a two-layer hierarchy – a control layer that provides the control interface and control logic, and 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 forms a channel for full duplex media processing. They are the addressable entities that can be controlled individually by the applications.

Figure 1. Architecture of Intel® IXP400 DSP Software



3.0 Media Processing Resource Components

As shown in Figure 1, the addressable control entities of DSP software are Media Processing Resource (MPR) components. There are nine resource components, working together to provide all the media processing needed by a gateway or server channel. Each resource component has a unique identifier as shown below. In the following, we will refer to each of these nine media processing entities as either a resource or a resource component.

Figure 2. Resource Component Identifiers

```
typedef enum{
    XMPR_ANY=0,         /* any resource, not currently supported */
    XMPR_NET,           /* Network Endpoint resource */
    XMPR_DEC,           /* Decoder resource */
    XMPR_ENC,           /* Encoder resource */
    XMPR_TNGEN,         /* Tone generator resource */
    XMPR_TNDET,         /* Tone detector resource */
    XMPR_PLY,           /* Audio player resource */
    XMPR_MIX,           /* Audio mixer resource */
   XMPR_T38,            /* T38 IP fax resource */
    XMPR_MA             /* Message Agent resource */
} XMPResource_t;
```

Each resource contains a particular set of algorithms to perform a specific set of media-processing functions. For example, the Network Endpoint resource consists of echo cancellation, high pass filter and PCM A-law or µ-law conversion algorithms to perform TDM front-end processing. Each resource, therefore, has a unique set of parameters associated with the particular set of algorithms it contains.

Communications of control information to these resource components are through 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 though the messages because they can only be updated by the internal control task.

3.1 Network Endpoint Resource Component

Resource Type: XMPR_NET

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

Parameters

Identifier	Description, Values	Attr.	Direct Write
XPARMID_RES_STATE	Current state (0: idle, 1: active)	R	N
XPARMID_NET_LP_STREAM	The L-Port stream ID. Default: the stream assigned to the IP termination's T-Port of the same channel if exist, otherwise –1.	R/W	N
XPARMID_NET_LAW	PCM data format on HSS TDM bus. XPARM_NET_ALAW or XPARM_NET_MULAW. Default: XPARM_NET_MULAW	R/W	N
XPARMID_NET_ECENABLE	EC enabling flag, XPARM_ON or XPARM_OFF. Default: XPARM_ON	R/W	Y
XPARMID_NET_ECTAIL	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.	R/W	N
XPARMID_NET_ECNLP	EC NLP and suppress flag, XPARM_OFF, XPARM_EC_NLP_ON or XPARM_EC_NLP_SUP_ON. Default: XPARM_OFF	R/W	N
XPARMID_NET_ECFREEZE	EC freezing flag, XPARM_ON (freeze) or XPARM_OFF (adaptive). Typically, freeze is used only in debug situations. Default: XPARM_OFF	R/W	N
XPARMID_NET_DELAYCOMP	EC delay compensation (0 ~ 240 in **0.125-ms** units). Default: 20 (or 2.5 ms delay compensation)	R/W	Y
XPARMID_NET_FLASH_HK	The window of flash hook detection (in 10-ms units) Default: 100	R/W	Y
XPARMID_NET_TIMER	Timer counter (in 10 ms unit). This timer can be used for timing that is synchronized to the TDM clock. Default: 0	R/W	Y
XPARMID_NET_GAIN_RX	Input gain of HSS interface (+15 ~ –40 in 1-dB units) Default: 0	R/W	N
XPARMID_NET_GAIN_TX	Output gain of HSS interface (+15 ~ –40 in 1-dB units) Default: 0	R/W	N
XPARMID_NET_HSS_BYPASS	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	R/W	N

Events

- XEVT_NET_HOOK_STATE — Hook state change detected.
- XEVT_NET_TIMER — Timer expired.

3.2 Decoder Resource Component

Resource Type: XMPR_DEC

Media Processing Functions

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

Resource-Specific Control Messages

- `XMSG_CODER_START` (inbound)
- `XMSG_CODER_STOP_ACK` (outbound)

Parameters

Identifier	Description and Values	Attr.	Direct Write
XPARMID_RES_STATE	Current state (0: idle, 1: active)	R	N
XPARMID_DEC_VOL	Decoder volume adjustment; +15 ~ -40 in 1-dB units. Default: 0 (Set to -99 to mute)	R/W	N
XPARMID_DEC_ALC	ALC enable flag. `XPARM_ON` or `XPARM_OFF`. Default: `XPARM_ON`	R/W	N
XPARMID_DEC_CNG	CNG enable flag. `XPARM_ON` or `XPARM_OFF`. Default: `XPARM_OFF`	R/W	Y
XPARMID_DEC_CTYPE	Coder type. Currently supported types are `XCODER_TYPE_G711MU_10MS`, `XCODER_TYPE_G711A_10MS`, `XCODER_TYPE_G729A` or `XCODE_TYPE_G723`, `XCODER_TYPE_G722`, `XCODER_TYPE_G726_40`, `XCODER_TYPE_G726_32`, `XCODER_TYPE_G726_24`, `XCODER_TYPE_G726_16`, and `XCODER_TYPE_G729`. Default: `XCODER_TYPE_G711MU_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_ON`.	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 (0 ~ 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

Identifier	Description and Values	Attr.	Direct Write
XPARMID_DEC_G726_40_RTP_PLD	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	R/W	Y
XPARMID_DEC_G726_32_RTP_PLD	RTP payload type for G.726 32-Kkbps coder, The payload type is negotiated and set by the call stack. The range of values is 96 to 127. Default: 97	R/W	Y
XPARMID_DEC_G726_24_RTP_PLD	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	R/W	Y
XPARMID_DEC_G726_16_RTP_PLD	RTP payload type for G.726 16kbps coder, The payload type is negotiated and set by the call stack. The range of values is 96 to 127. Default: 99	R/W	Y
XPARMID_DEC_G726_PACK	G.726 packing format. Set to `XPARM_G726_PACK_LSB` for RFC 3551 format, or `XPARM_G726_PACK_MSB` for I.366.2 Annex E format. Default: `XPARM_G726_PACK_LSB`	R/W	N

Events

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

3.3 Encoder Resource Component

Resource Type: `XMPR_ENC`

Media Processing Functions

- Encoding
- Automatic Gain Control
- Voice Activity Detection

Resource-Specific Control Messages

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

Parameters

Identifier	Description and values	Attr.	Direct Write
XPARMID_RES_STATE	Current state (0: idle, 1: active)	R	N
XPARMID_ENC_LP_STREAM	L-Port stream ID. Default: the stream assigned to the TDM termination's T-Port of the same channel if exist, otherwise –1.	R/W	N
XPARMID_ENC_AGC	AGC enable flag. XPARM_ON or XPARM_OFF. Default: XPARM_OFF	R/W	N
XPARMID_ENC_VAD	VAD enable flag. XPARM_ON or XPARM_OFF. Default: XPARM_OFF	R/W	N
XPARMID_ENC_CTYPE	Coder type. Currently supported types are XCODER_TYPE_G711MU_10MS, XCODER_TYPE_G711A_10MS, XCODER_TYPE_G729A or XCODE_TYPE_G723, XCODER_TYPE_G722,XCODER_TYPE_G726_40, XCODER_TYPE_G726_32, XCODER_TYPE_G726_24, XCODER_TYPE_G726_16, and XCODER_TYPE_G729. Default: XCODER_TYPE_G711MU_10MS	R/W	N
XPARMID_ENC_MFPP	Number of frames per packet. Supported range is 1~6 for G.711 and G.722, 1~8 for G.723, 1~9 for G.726 40 Kbps, 1~12 for G.726 32 Kbps, 1~16 for G.726 24 Kbps, and 1~24 for G.729 and G.726 16 Kbps. Default: 1.	R/W	N
XPARMID_ENC_EVT_PKT	Enable packet lost event. XPARM_ON or XPARM_OFF. Default: XPARM_OFF	R/W	Y
XPARMID_ENC_G726_40_RTP_PLD	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	R/W	Y
XPARMID_ENC_G726_32_RTP_PLD	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	R/W	Y
XPARMID_ENC_G726_24_RTP_PLD	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	R/W	Y
XPARMID_ENC_G726_16_RTP_PLD	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	R/W	Y
XPARMID_ENC_G726_PACK	G.726 packing format. Set to XPARM_G726_PACK_LSB for RFC 3551 format, or XPARM_G726_PACK_MSB for I.366.2 Annex E format. Default: XPARM_G726_PACK_LSB	R/W	N
XPARMID_ENC_VOL	Encoder gain adjustment, +15 ~ – 40 in 1-dB units. Default: 0 (Set to -99 to mute)	R/W	N

Events

- XEVT_LOST_PACKET — Bad packet.
- XEVT_DEC_PACKET_CHNG — Received RTP payload type changed.

3.4 Tone Generation Resource Component

Resource Type: XMPR_TNGEN

Media Processing Functions

- Generating multiple frequency tone signals
- Generating call progress tones

Resource-Specific Control Messages

- XMSG_TG_PLAY *(inbound)*
- XMSG_TG_PLAY_FSK *(inbound)*
- XMSG_TG_PLAY_CMPLT (outbound)

Parameters

Identifier	Description and values	Attr.	Direct Write
XPARMID_RES_STATE	Current state (0: idle, 1: active)	R	N
XPARMID_TNGEN_VOL	Tone Generator's volume adjustment, +15 ~ –20 in 1-dB units. Default: 0	R/W	N
XPARMID_TNGEN_FSK_MOD	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	R/W	Y
XPARMID_TNGEN_FSK_CS	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.	R/W	Y
XPARMID_TNGEN_FSK_MARK	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.	R/W	Y
XPARMID_TNGEN_FSK_RATE	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, i.e., 1200 bps	R/W	N
XPARMID_TNGEN_FSK_POSTMK	Postmark bit length of FSK modulator (in bit unit) Default: 72	R/W	Y
XPARMID_TNGEN_RFC2833	RFC2833 enable flag. XPARM_ON or XPARM_OFF. Default: XPARM_ON	R/W	N

Events

None.

3.5 Tone Detection Resource Component

Resource Type: `XMPR_TNDET`

Media Processing Functions

- Receiving DTMF digits
- Detecting individual tone event

Resource-Specific Control Messages

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

Parameters

Identifier	Description and values	Attr.	Direct Write
XPARMID_RES_STATE	Current state (0: idle, 1: active)	R	N
XPARMID_TD_LP_STREAM	L-Port stream ID. Default: the stream assigned to the DTM termination's T-Port of the same channel if exist, otherwise –1.	R/W	N
XPARMID_TD_TC	Tone Clamping enable flag. `XPARM_ON` or `XPARM_OFF`. Default: `XPARM_OFF`	R/W	Y
XPARMID_TD_TC_FRAMES	Tone Clamping buffer size. 0 ~ 3 in 10 ms unit. Default: 3	R/W	N
XPARMID_TD_RPT_EVENTS	Tone event enable flag. `XPARM_OFF`, `XPARM_TD_RPT_TONE_ON`, `XPARM_TD_RPT_TONE_OFF` or `XPARM_TD_RPT_TONE_ON_OFF`. Default: `XPARM_OFF`	R/W	Y
XPARMID_TD_RFC2833E_ENABLE	RFC2833 event enable flag. `XPARM_ON` or `XPARM_OFF`. Default: `XPARM_OFF`	R/W	Y
XPARMID_TD_RFC2833E_UPDATERATE	RFC 2833 packet rate in 10-ms units, i.e., the period between the packets generated when a tone event is detected. Default: 5	R/W	N
XPARMID_TD_RFC2833E_NUMEOE	Redundancy of end-of-event packet. Range 0-255. Default: 3	R/W	Y
XPARMID_TD_RFC2833E_NUMBOE	Redundancy of begin-of-event packet. Range 0-255. Default: 0	R/W	Y
XPARMID_TD_RFC2833E_AUDIOSUPRESS	Flag of audio encoding suppression when event detected. `XPARM_ON` or `XPARM_OFF`. Default: `XPARM_ON`	R/W	N

Identifier	Description and values	Attr.	Direct Write
XPARMID_TD_RFC2833E_PAYLOADTYPE	RFC 2833 Payload type, Range is in the RTP dynamic payload type range of 96 to 127. Default: 0x65.	R/W	Y
XPARMID_TD_FSK_CS	Minimum CS-bit length required by FSK receiver. Default: 200 if country code set to COUNTRY_CODE_US or COUNTRY_CODE_PRC, otherwise 0.	R/W	Y
XPARMID_TD_FSK_MARK	Minimum mark-bit length required by FSK receiver. Default: 100 if country code set to COUNTRY_CODE_US or COUNTRY_CODE_PRC, otherwise 60.	R/W	Y
XPARMID_TD_FSK_STOP	Extra stop bits allowed between data. Default: 20	R/W	Y
XPARMID_TD_FSK_RATE	Baud rate of FSK receiver. (Reserved for future, currently only support 1,200 bps rate)	R/W	Y

Events

- XEVT_CODE_TD_TONEON – tone on event for an individual tone
- XEVT_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

Media Processing Functions

- Play back recorded audio data.

Resource-Specific Control Messages

- XMSG_PLY_START *(inbound)*
- XMSG_PLY_CMPLT (outbound)

Parameters

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

Events

None.

3.7 Audio Mixer Resource Component

Resource Type: XMPR_MIX

Media Processing Functions

Mixing multiple audio streams for three-way call or small audio conference. The maximum number of parties to the mixer is currently five.

Resource-Specific Control Messages

None.

Parameters

Identifier	Description and values	Attr.	Direct Write
XPARMID_RES_STATE	Current state (0: idle, 1: active)	R	N
XPARMID_MIX_LP_STREAM	The first L-Port stream ID. Default: –1	R/W	N
XPARMID_MIX_LP_STREAM+1	The 2nd L-Port stream ID. Default: –1	R/W	N
XPARMID_MIX_LP_STREAM+n-1	The *n*th L-Port stream ID. Default: –1	R/W	N

Events

None.

3.8 T.38 Fax Resource Component

Resource Type: XMPR_T38

Media Processing Functions

- Real-time fax gateway between TDM interface and IP network

Resource-Specific Control Messages

- XMSG_T38_START (inbound)
- XMSG_T38_CMPLT (outbound

Parameters

Identifier	Description and Values	Attr.	Direct Write
XPARMID_RES_STATE	Current state (0: idle, 1: active)	R	N
XPARMID_T38_ELLIPSIS	Flag of enabling support of ellipsis added to Internet Fax Protocol in T.38 Corrigendum 1 (2001). XPARM_ON or XPARM_OFF. Default: XPARM_OFF	R/W	N
XPARMID_T38_FEC	Flag of enabling FEC. XPARM_ON or XPARM_OFF. Default: XPARM_OFF	R/W	N
XPARMID_T38_REDUNDANCY	Redundancy level, (0 ~ 7) Default: 0	R/W	N
XPARMID_T38_RATE_NEG	Method of modem rate negotiation. XPARM_T38_RATE_NEG_LOCAL or XPARM_T38_RATE_NEG_REMOTE. Default:XPARM_T38_RATE_NEG_REMOTE if packet transferred over UDP, otherwise XPARM_T38_RATE_NEG_LOCAL	R/W	N
XPARMID_T38_TCF_THRSHLD	TCF error threshold (in percentage). Only applies if local modem rate negotiation is selected. Default: 5	R/W	N
XPARMID_T38_TRANSPORT	Protocol used to transfer T.38 packets over IP network. XPARMID_T38_TRANS_UDP or XPARMID_T38_TRANS_TCP (only XPARMID_T38_TRANS_UDP is supported in this release). Default : XPARMID_T38_TRANS_UDP	R/W	N
XPARMID_T38_MODE	T.38 mode, XPARM_T38_MODE_ITU or XPARM_T38_MODE_CHINA. Default: XPARM_T38_MODE_ITU	R/W	N
XPARMID_T38_DISCONNECT	Enable China T.38 disconnect message generation. Applies only if China T.38 mode is selected. XPARM_ON or XPARM_OFF. Default: XPARM_OFF	R/W	N
XPARMID_T38_FEC_NMESSAGES	Number of FEC messages per UDPTL packet when FEC is enabled. (1 ~ 5) Default : 2	R/W	N
XPARMID_T38_FEC_NPACKETS	Number of previous packets per FEC message when FEC is enabled. (2 ~ 3) Default : 2	R/W	N

Events

XEVT_T38_END — End of the T.38 session. Event Data 1 gives the reason of the termination.

3.9 Message Agent Resource Component

Resource Type: XMPR_MA

Media Processing Functions

- No media processing function.

- Converting the user-defined messages and executing the control accordingly.

Resource-Specific Control Messages

None.

Parameters

Identifier	Description and values	Attr.	Direct Write
XPARMID_RES_STATE	Current state (0: idle, 1: active)	R	N
XPARMID_MA_DEBUG	Enable trace during processing user's messages. `XPARM_ON` or `XPARM_OFF` Default: XPARM_OFF	R/W	Y

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.

4.1 Message Functions

Three functions are provided to send and receive messages.

XStatus_t xMsgSend (void *pMsgBuf);	
Description	Sends a control message to the in-bound message queue
Input	`pMsgBuf` – Pointer to the message buffer.
Output	None
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.

XStatus_t xMsgReceive (void *pMsgBuf, UINT16 channel, int timeout);	
Description	Receives acknowledgement or event from the outbound message queue.
Input	• pMsgBuf – Pointer to the message buffer • channel – Channel number. (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.

XStatus_t xMsgWrite (void *pMsgBuf);	
Description	Posts a message (e.g. an user defined external event message) to the out-bound queue so that it can be retrieved by XMsgReceive().
Input	pMsgBuf — Pointer to the message buffer.
Output	None
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

Format	`typedef struct{`
	`UINT32 transactionId; /* used by apps to track the message */`
	`UINT16 instance; /* instance ID (1-0xffff), 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_t;`

Caution	Message content must follow the header in contiguous memory.

Macros	`#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_T38_START,         /* start T.38 resource */
    XMSG_T38_CMPLT,         /* T.38 session complete message */
    XMSG_END                      /* end of list */
} XMsgType_t;
```

5.0 Common Control Message

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

5.1 Reset Message

Type	**XMSG_RESET**
Direction	Inbound
Description	Stops the current action and resets the resource to idle state.

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

Macro	`#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

Type	**XMSG_Start**
Direction	Inbound
Description	Generic start message. Starts the media-processing functions on a resource.

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

Macro	`#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

Type	**XMSG_STOP**
Direction	Inbound
Description	Stops the current action.

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

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

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

5.4 Ping Message

Type	**XMSG_PING**
Direction	Inbound
Description	Verifies if the resource is alive.

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

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

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

5.5 Set Parameter Message

Type	**XMSG_SET_PARM (Sheet 1 of 2)**
Direction	Inbound
Description	Sets a parameter to a resource.

Type	**XMSG_SET_PARM (Sheet 2 of 2)**

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

Type	**XMSG_SET_MPARMS**
Direction	Inbound
Description	Set multiple parameters to a resource

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(XMsgSetmParms_t),\`
	`  XMSG_SET_MPARMS, 0)\`
	`  ((XMsgSetmParms_t *)(pMsg))->numParms = num; \`
	`}`
	
	`#define XMSG_FIELD_SET_MPARMS(pMsg, pIDs, pVals) \`
	`{\`
	`  pIDs =  ((XMsgSetmParms_t *)(pMsg))->parmIDs;\`
	`  pVals =  ((XMsgSetmParms_t *)(pMsg))->values;\`
	`}`

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

5.7 Get Parameter Message

Type	**XMSG_GET_PARM**
Direction	Inbound
Description	Gets a parameter from a resource.

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) \`
	`{\`
	`  XMSG_MAKE_HEAD(pMsg, trans, res, inst, sizeof(XMsgGetParm_t),\`
	`  XMSG_GET_PARM, 0)\`
	`  ((XMsgGetParm_t *)(pMsg))->parmId= id;\`
	`}`

Response	• Specific acknowledgement message (XMSG_GET_PARM_ACK) • Error message (XMSG_ERROR) if error.

5.8 Get Parameter Acknowledge Message

Type	**XMSG_GET_PARM_ACK**
Direction	Outbound
Description	Resource returns the parameter enquired.

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)\`
	`{\`
	`  id  = ((XMsgGetParmAck_t *)(pMsg))->parmId;\`
	`  val  = ((XMsgGetParmAck_t *)(pMsg))->value;\`
	`}`

5.9 Get All Parameters Message

Type	**XMSG_GET_ALLPARMS (Sheet 1 of 2)**
Direction	Inbound
Description	Gets all parameters from a resource.

Type	XMSG_GET_ALLPARMS (Sheet 2 of 2)

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

Type	XMSG_GET_ALLPARMS_ACK
Direction	Outbound
Description	Resource returns the parameter inquired.

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

Type	XMSG_ACK
Direction	Outbound
Description	Resource indicates the control message has been processed successfully.

Format	`typedef struct{`
	`  XMsgHdr_t head;  /* message header */`
	`} XMsgAck_t;`

5.12 Error Message

Type	**XMSG_ERROR**
Direction	Outbound
Description	Resource reports an error condition. (See constant data section for error codes.)

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

Macro	`#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

Type	**XMSG_EVENT**
Direction	Outbound
Description	Resource reports an event condition. (See constant data section for error codes.)

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

Macro	`#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

Type	**XMSG_CODER_START**
Direction	Inbound
Description	Starts a decoder or encoder.

Format	`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

Type	**XMSG_CODER_STOP_ACK**
Direction	Outbound
Description	Decoder or encoder resource acknowledges the `XMSG_STOP` message

Format	`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

Type	**XMSG_TG_PLAY**
Direction	Inbound
Description	Requires Tone Generator to play a tone string. (Tone ID's are listed in the constant data section.)

Format	`typedef struct{`
	`  XMsgHdr_t  head;  /* message header */`
	`  UINT8  numTones;  /* number of tones to play */`
	`  UINT8  toneId[XMAX_TONEBUFSIZE];  /* tone ID string */`
	`} XMsgTGPlay_t;`

Macro	`#define XMSG_MAKE_TG_PLAY(pMsg, trans, inst, num)\`
	`{\`
	`  XMSG_MAKE_HEAD(pMsg, trans, XMPR_TNGEN, inst, sizeof(XMsgTGPlay_t),\`
	`  XMSG_TG_PLAY, 0)\`
	`  ((XMsgTGPlay_t *)(pMsg))->numTones = num;\`
	`}`
	
	`#define XMSG_FIELD_TG_PLAY(pMsg, pToneID) \`
	`{\`
	`  pToneID= ((XMsgTGPlay_t *)(pMsg))->toneId;\`
	`}`

6.4 Tone Generator Play FSK Message

Type	**MSG_TG_PLAY_FSK**
Direction	Inbound
Description	Require Tone Generator to play a FSK modulated data

Format	`typedef struct{`
	`  XMsgHdr_t  head;  /* message header */`
	`  UINT8  numBytes;  /* number of bytes to play */`
	`  INT8  data[XMAX_FSKDATASIZE];  /* data string */`
	`} XMsgTGPlayFSK_t;`

Macro	`#define XMSG_MAKE_TG_PLAY_FSK(pMsg, trans, inst, num)\`
	`{\`
	`  XMSG_MAKE_HEAD(pMsg, trans, XMPR_TNGEN, inst, sizeof(XMsgTGPlayFSK_t),\`
	`  XMSG_TG_PLAY_FSK, 0)\`
	`  ((XMsgTGPlayFSK_t *)(pMsg))->numBytes = num;\`
	`}`
	
	`#define XMSG_FIELD_TG_PLAY_FSK(pMsg, pData) \`
	`{\`
	`  pData= ((XMsgTGPlayFSK_t *)(pMsg))->data;\`
	`}`

Response	• Tone Generator Play-Completed message (`XMSG_TG_PLAY_CMPLT`)

6.5 Tone Generator Play Completed Message

Type	**XMSG_TG_PLAY_CMPLT**
Direction	Outbound
Description	Tone Generator indicates the completion of playing tones.

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

Type	**XMSG_TD_RCV**
Direction	Inbound
Description	Require Tone Detector to receive a tone string.

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`)

6.7 Tone Detector Receive Completed Message

Type	**XMSG_TD_RCV_CMPLT**
Direction	Outbound
Description	Tone detector indicates the completion of receiving DTMF tones.

Format	`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`

Macro	`#define XMSG_FIELD_TD_RCV_CMPLT(pMsg, rsn, num, pBuf)\`
	`{\`
	`  rsn = ((XMsgTDRcvCmplt_t *)(pMsg))->reason;\`
	`  num = ((XMsgTDRcvCmplt_t *)(pMsg))->numDigits;\`
	`  pBuf= ((XMsgTDRcvCmplt_t *)(pMsg))->digits;\`
	`}`

6.8 Tone Detector Receive FSK Message

Type	**MSG_TD_RCV_FSK**
Direction	Inbound
Description	Require Tone Detector to receive FSK data

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

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

Response	Tone Detector FSK receive-completed message (`XMSG_TD_RCV_FSK_CMPLT`)

6.9 Tone Detector FSK Receive Completed Message

Type	**XMSG_TD_RCV_FSK_CMPLT**
Direction	Outbound
Description	Tone Detector indicates the completion of receiving FSK data

Format	`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	`#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

Type	**XMSG_PLY_START (Sheet 1 of 2)**
Direction	Inbound
Description	Start Player to play back pre-recorded audio data

Type	XMSG_PLY_START (Sheet 2 of 2)

Format	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
	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	#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)

6.11 Player Play Completed Message

Type	XMSG_PLY_CMPLT
Direction	Outbound
Description	Player indicates the completion of playing audio data.

Format	typedef struct{
	  XMsgHdr_t head; /* message header */
	  UINT16 reason; /* the reason of completion */
	} XMsgPlyCmplt_t;
	
	where the reason may be:
	#define XMSG_STOP_REASON_USER 1
	#define XMSG_STOP_REASON_EOD 2

Macro	#define XMSG_FIELD_PLY_CMPLT(pMsg, rsn)\ {\ rsn = ((XMsgPlyCmplt_t *)(pMsg))->reason;\ }

6.12 Get Jitter Buffer Statistics Message

Type	**XMSG_GET_JBSTAT**
Direction	Inbound
Description	Get the jitter buffer statistics from a Decoder instance.

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

Macro	`#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`)

6.13 Complete Message of Getting Jitter Buffer Statistics

Type	**XMSG_GET_JBSTAT_CMPLT**
Direction	Outbound
Description	Response to the message of getting the jitter buffer statistics.

Format	`typedef struct{`
	`  XMsgHdr_t  head;  /* message header */`
	`  XJBStatistics_t  stat;  /* jiter buffer statistics */`
	`} XMsgGetJBStatCmplt_t;`
	
	`where the XMsgGetJBStatCmplt_t date structure of jitter buffer statistics is defined as`
	`typedef struct{`
	`  UINT32  rcvdPackets;  /* total packets received */`
	`  UINT32  lostPackets;  /* lost packets */`
	`  UINT32  badFrames;  /* decoder bad frames */`
	`  UINT32  rcvdTonePackets;  /* RFC2833 packets received */`
	`} XJBStatistics_t;`

Macro	`#define XMSG_FIELD_GET_JBSTAT_CMPLT(pMsg, pStat)\`
	`{\`
	`  pStat = &(((XMsgGetJBStatCmplt_t *)(pMsg))->stat);\`
	`}`

6.14 T.38 Session Start Message

Type	**XMSG_T38_START**
Direction	Inbound
Description	Start a T.38 session. The toneID field indicates the tone which was detected that caused this message to be issued. The options are 0 (no tone detected), Fax CED tone (RFC_TID_FAX_CED), Fax CNG tone (RFC_TID_FAX_CNG) or V.21 modem signal (RFC_TID_FAX_V21).
Format	typedef struct{ XMsgHdr_t head; /* message header */ UINT16 toneId; /* fax tone id */ } XMsgT38Start_t;
Macro	#define XMSG_MAKE_T38_START(pMsg, trans, inst, tnid)\ {\ XMSG_MAKE_HEAD(pMsg, trans, XMPR_T38, inst,\ sizeof(XMsgT38Start_t), XMSG_T38_START, 0)\ ((XMsgT38Start_t *)(pMsg))->toneId = tnid;\ }
Response	T38 session completed message (XMSG_T38_CMPLT)

6.15 T.38 Session Complete Message

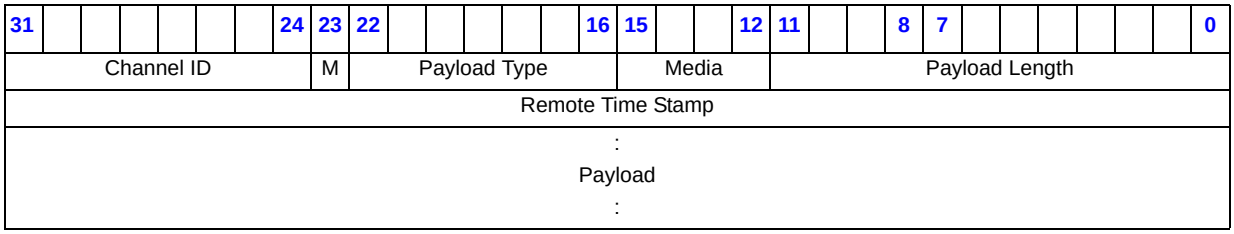
Type	**XMSG_T38_CMPLT**
Direction	Outbound
Description	Indicate the completion of a T.38 session
Format	typedef struct{ XMsgHdr_t head; /* message header */ UINT16 reason; /* the reason of completion */ } XMsgT38Cmplt_t;
Macro	#define XMSG_FIELD_T38_CMPLT(pMsg, rsn)\ {\ rsn = ((XMsgT38Cmplt_t *)(pMsg))->reason;\ }

7.0 Packet Data Interface

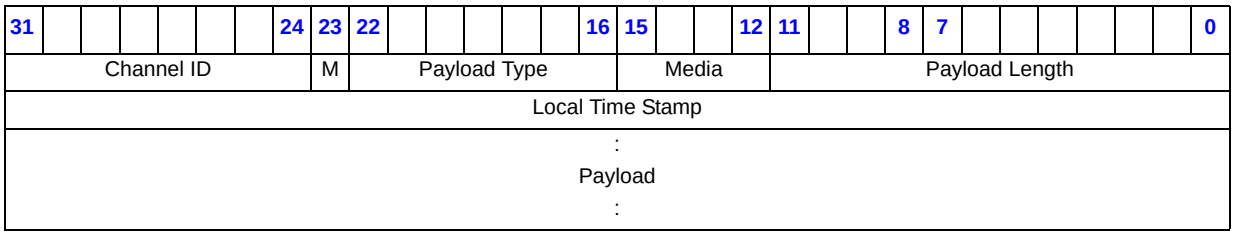
The packet data interface is a protocol for the Intel® IXP400 DSP Software to exchange the encoded data packets with IP stack. This interface is defined as a packet format and two callback functions – one is provided by DSP software release and another is provided by the user (IP stack).

7.1 Packet Formats

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



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



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

Field	Description
Local Time Stamp	Packet arrival time as measured by a local clock.
Remote Time Stamp	Packet data sampling time measured by a remote clock.
Payload Length	Payload length in bytes.
Media	4-bit media type field is defined as: • 0x01 – Audio • 0x02 – Tone (RFC 2833 event type) • 0x04 – Tone (RFC 2833 tone type) • 0x08 – T.38 UDP • 0x09 – T.38 TCP
M	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.
Payload type	RTP payload type as defined in RFC 1990.
Payload	Encoded audio data or RFC-2838, tone-event information.

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;
```

In ingress, the header information of Remote Time Stamp, Payload Type and Marker bit are directly copied from a RTP packet. In egress, the header information is filled by 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 DSP software and IP stack via the callback functions. The packet delivery module calls the function and passes the packet each time when 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 (i.e., 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 (i.e., the processing must be guaranteed to complete within a certain short period of time).

The function that the DSP software receives the packets from the IP stack is provided as follows:

XStatus_t xPacketReceive (UNIT16 channel, XPacket_t *buffer);	
Description	Call-back function to receive packets.
Input	`Buffer` – memory address of the packet `Channel` – Channel numbers
Output	None
Return	`XSUCC` – If successful `XERROR` – If the packet receptor is unable to process the packet.

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. DSP software will call the function each time when a packet generated. That function must be registered during initialization as described in next section.

8.0 Configuration and Initialization

The Intel® IXP400 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

Prototype	`void xDspSysInit(XDSPSysConfig_t *pSysConfig);`
Input	`pSysConfig` – System configuration information
Output	None
Return	None

Description

This function performs the following procedures:

- Initialize and start HSS port
- Create TDM termination channels (i.e., Network Endpoint resource instance) and link them to the HSS time slots sequentially. Error will occur if not enough time slots are enabled for all the TDM channels
- Create the IP terminations (i.e., Decoder, Encoder, Tone Generator and Tone Detector resources)
- Create media service resources (i.e., 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:

```
typedef struct{
    int       numChTDM;             /* number of channels of TDM termination(1~4) */
    int       numChIP;              /* number of channels of IP termination (1~4) */
    int       numPlayers;           /* number of Player instances (1~4) */
    int       numMixers;            /* number of Audio Mixers (must be 1) */
    int       numPortsPerMixer;     /* number of ports per mixer (3~5) */
    int       countryCode;          /* country code */
    int       taskPriBase;          /* the base priority of DSP module */
    int       taskPriOrder;         /* the priority ordering of the OS */
    IxHssAccHssPort       port;          /* HSS port (must be Port 0) */
    IxHssAccConfigParams  *pHssCfgParms; /* HSS configuration parameters */
    IxHssAccTdmSlotUsage  *pHssTDMSlots; /* HSS TDM time slot mapping */

   XDSPChanTdmSlots_t *pChanTsMap; /* channel vs. time slot mapping */
    XPktRcvFxn_t          pktRcvFxn;     /* packet receiver function in egress */

    XMsgAgentDec_t         msgDecoder;    /* optional message decoder function of MA */

    XMsgAgentEnc_t         msgEncoder;    /* optional message encoder function of MA */
} XDSPSysConfig_t;

where:
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);
```

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:

```
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. The time slot locations for the channels in a 32-slot frame are shown as:

<table>
<tr><td>0</td><td>1</td><td>2</td><td>...</td><td>16</td><td>17</td><td>...</td><td>31</td></tr>
<tr><td>MSA</td><td>LSB</td><td>µ-law</td><td>...</td><td>MSB</td><td>LSB</td><td>...</td><td></td></tr>
<tr><td colspan="2">1st WB sample</td><td>NB Sample</td><td></td><td colspan="2">2nd WB sample</td><td></td><td></td></tr>
</table>

Then the array that describes such configuration is given as:

```
XDSPChanTdmSlots_t chanTsMapping[2] =

{

    {0, 16},                          /* channel 1 - WB, time slot 0 and 16 */

    {2, XCHAN_TDM_SLOT_NULL}          /* channel 2 - NB, time slot 2 */

};
```

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 (e.g., 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 System Configuration with External PCM Interface

Prototype	`void xDspSysInit2(XDSPSysConfig2_t *pSysConfig);`
Input	`pSysConfig` – System configuration information
Output	None
Return	None

Description

This function performs the similar system initialization to xDspSysInit(), except it does not initialize the HSS device, opening an external PCM data interface and allowing users to obtain the PCM data in alternative way. The users specify configuration information as defined in the data structure XDSPSysConfig2_t.

```
typedef struct{
    int                  numChTDM;         /* number of PCM channels */
    int                  numChIP;          /* number of channels of IP termination */
    int                  numPlayers;       /* number of player instances */
    int                  numMixers;        /* number of Audio Mixers */
    int                  numPortsPerMixer; /* number of ports per mixer */
    int                  countryCode;      /* country code */
    int                  taskPriBase;      /* the base priority of DSP module */
    int                  taskPriOrder;     /* the priority ordering of the OS */
int                   framesPerBuf;     /* PCM buffer size in terms of frame size */
int                   transferType;     /* data transfered via DMA or CPU */
    XDSPExtChan_t        *pExtChannel;     /* array of external PCM channel */
    XPktRcvFxn_t         pktRcvFxn;        /* packet receiver function */
    XMsgAgentDec_t       msgDecoder;       /* message decoder function of MA */
    XMsgAgentEnc_t       msgEncoder;       /* message encoder function of MA */
} XDSPSysConfig2_t;
```

The user provides the information of the data format and transfer buffers through an array of XDSPExtChan_t structure, which is defined as

```
typedef struct{
    void    *pRxBuffer;     /* address of Rx circular buffer */
    void    *pTxBuffer;     /* address of Tx circular buffer */
    int     format;         /* data format, 8-bit, 16-bit or 16-bit wideband */
} XDSPExtChan_t;
```

The restriction of the external PCM interface includes

- xDspSysInit2()and xDspSysInit()are mutual exclusive. The users can choose either of them but not both.
- The user application is responsible to allocate two data transfer buffers (Rx and Tx buffers) for each channel if using external PCM interface. (Here Rx refers the direction going to DSP module and Tx for the opposite).
- The data formats can be 8-bit compressed (A-law or -law), 16-bit linear or 16-bit wideband-linear (16KHz sampling rate), specified by the format in XDSPExtChan_t as XPCM_FORMAT_8BIT, XPCM_FORMAT_16BIT and XPCM_FORMAT_16BIT_WB respectively. For 16-bit linear format, it must be left-adjust signed fraction or Q.15 format.

- The length of the data transfer buffers is specified in 0.125ms unit. All the channels must have the same length regarding this time unit.
- The buffer length must be the multiple of the frame size defined by XPCM_FRAME_SIZE (80 in 0.125ms unit) and must be at least two times of this frame size. The buffer length is specified by framesPerBuf in XDSPSysConfig2_t in term of frames per buffer.
- The external device must transfer the data in the synchronous manner, i.e., the device maintain the common access index for all the channels.
- The device must call the function xDspPcmSync(rxOffset, txOffset) every frame period (10ms) and pass its current access index - rxOffset and txOffset. The index is given in 0.125ms unit and must be always at the frame boundary. (e.g, 0, 80,160 if the buffer length is 240)
- Cache flush/invalidation will be performed is the transferType field in XDSPSysConfig2_t is set to XPCM_XFER_TYPE_DMA.

8.3 Adding Tones to Tone Generator

Prototype	`XStatus_t xBuildToneTG(UINT16 toneId, UINT16 numSegs, XTGToneSeg_t *pToneSegs, UINT32 *pErrCode);`
Input	• `toneTd` — 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`

Description

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

```
typedef struct {
   UINT16  repCount;    /* repetition number 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. */
   UINT32  durationOff;/* silence duration in 1-ms unit. */
   INT16   freqA;       /* 1st frequency if single or dual frequency wave,
                           or 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 segement.
                          e.g., 1 means to go 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;
```

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

8.4 Change the DTMF Tone Parameters

Prototype	`Status_t xSetTGParmDTMF(int toneOn, int toneOff, int ampdBm);`

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`

Output	
Return	`XSUCC if successful` `otherwise XERROR`

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.

8.5 Adding Tones to Tone Detector

Prototype	`Status_t xBuildToneTD(UINT8 toneId, XTDToneInfo_t *pToneInfo, UINT32 *pErrCode);`
Input	• `toneId` – Tone ID, must be in the range of 16 ~ 255 • `pToneInfo` — Tone detection criterion information
Output	`pErrCode` – Error code if errors
Return	• `XSUCC` if successful • Otherwise `XERROR`

Description

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

```
/* 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()`.

8.6 Amplitude Check in Tone Detection

Prototype	`XStatus_t xSetAmplitudeRangeTD(int category, int ampMinF0, int ampMaxF0, int ampMinF1, int ampMaxF1)`

Input	`Category - Tone category to specify DTMF tones or fax 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`
	`ampMinF1 - Maximum amplitude of the high frequency, +3 ~ -45 in dBm`

Output	`None`
Return	`XSUCC if successful, otherwise XERROR`

Description

The Tone Detector is able to detect the pre-defined DTMF tones and fax tones in the full amplitude level range of +3 ~ -43 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()`.

8.7 Getting DSP Resource Configuration and Routing Information

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

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:

```
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 TMD 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 stream ID information is used for the application to connect the T-Ports and L-Ports of the resources.

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:

Prototype	`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.

Prototype	`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 Flash Hook Detection

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 followed by an off-hook one within the time specified by the XPARMID_NET_FLASH_HK parameter, a flash hook event is reported.

The hook states are defined as:

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

9.3 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.

```
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;
```

9.4 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. Depending on each release, the build number may indicate the release types like normal release, service package (SP), early access release (EAR), etc. For example, the Intel® IXP400 DSP Software v2.6.2 EAR gives the string 02060201.

9.5 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 up 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:

```
#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      /* real 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 XERR_RESOURCE_BUSY              0x407  /* resource is busy */
#define XERR_RESOURCE_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 */
```

10.3 Tone IDs

The DTMF tone IDs used by the Tone Generator and Detector are defined as:

```
#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
```

Fax-tone IDs reported by the Tone Detector for fax bypass applications. Not supported by the Tone Generator.

```
#define RFC_TID_FAX_CED                 32
#define RFC_TID_FAX_CNG                 36
#define RFC_TID_FAX_V21                 40
```

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

```
#define RFC_TID_DIAL                66
#define RFC_TID_PBX_DIAL            67
#define RFC_TID_SP_DIAL             68
#define RFC_TID_2ND_DIAL            69
#define RFC_TID_RING                70
#define RFC_TID_SP_RING             71
#define RFC_TID_BUSY                72
#define RFC_TID_CONGESTION          73
#define RFC_TID_SP_INFO             74
#define RFC_TID_COMFORT             75
#define RFC_TID_HOLD                76
#define RFC_TID_REC                 77
#define RFC_TID_CALLER_WT           78
#define RFC_TID_CALL_WT             79
#define RFC_TID_PAY                 80
#define RFC_TID_POS_IND             81
#define RFC_TID_NEG_IND             82
#define RFC_TID_WARNING             83
#define RFC_TID_INSTRUSION          84
#define RFC_TID_CAL_CARD            85
#define RFC_TID_PAYPHONE            86
```

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:

```
#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:

```
#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 */
```

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

```
#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:

```
typedef enum{
    XCODER_TYPE_PASSTHRU = 0,
    XCODER_TYPE_G711MU_10MS,
    XCODER_TYPE_G711A_10MS,
    XCODER_TYPE_G729A,
    XCODER_TYPE_G723,
    XCODER_TYPE_G722,
    XCODER_TYPE_G726_40,
    XCODER_TYPE_G726_32,
    XCODER_TYPE_G726_24,
    XCODER_TYPE_G726_16,
    XCODER_TYPE_G729 = 17,
    XCODER_TYPE_UNDEF = -1
} XCoderType_t;
```

Mask bits used to specify the coder type subset in Decoder auto-switch parameter are defined as:

```
#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
```

Mask bits used to specify the termination digits in the XMSG_TD_RCV message are defined as:

```
#define XTD_TERM_DIGIT_NONE              0x0000
#define XTD_TERM_DIGIT_0                 0x0001
#define XTD_TERM_DIGIT_1                 0x0002
#define XTD_TERM_DIGIT_2                 0x0004
#define XTD_TERM_DIGIT_3                 0x0008
#define XTD_TERM_DIGIT_4                 0x0010
#define XTD_TERM_DIGIT_5                 0x0020
#define XTD_TERM_DIGIT_6                 0x0040
#define XTD_TERM_DIGIT_7                 0x0080
#define XTD_TERM_DIGIT_8                 0x0100
#define XTD_TERM_DIGIT_9                 0x0200
#define XTD_TERM_DIGIT_STAR              0x0400
#define XTD_TERM_DIGIT_POUND             0x0800
#define XTD_TERM_DIGIT_A                 0x1000
#define XTD_TERM_DIGIT_B                 0x2000
#define XTD_TERM_DIGIT_C                 0x4000
#define XTD_TERM_DIGIT_D                 0x8000
```

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:

```
#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 */
```

This page is intentionally left blank.