Intel® IXP400 Digital Signal Processing (DSP) Software Library Release 1.1

API Reference Manual

March 2005

Document Number: 273811-002a

Contents

Figures

Tables

None.

Revision History

Date	Revision	Description
March 2005	002a	Branding and document-title updates.
March 2003	002	Added minor updates to represent features of DSP 1.1.
January 2003	001	First release of this document.

1.0 Introduction

The Intel® IXP400 DSP Software Library Release 1.1 (DSP 1.1) is a software module that provides basic voice-processing functionalities for Voice-over-IP (VoIP), residential-gateway applications for the Intel® IXP425 Network Processor. DSP 1.1 can be viewed as a complete media-processing layer with control and data interfaces at its API.

This document defines the DSP 1.1 specifications.

1.1 General

DSP 1.1 is a firmware module for media processing, targeted for the next generation of integrated access devices (IADs) such as consumer-premise equipment (CPE). Specifically, the module provides the functionality — such as media compression, echo cancellation, tone processing, and jitter control — that is required in any IP media gateway, for real-time, media-streaming functionalities.

All media processing in DSP 1.1 occurs within a particular media channel created for communications between either two users or a user and machine. Therefore, the configurations and controls of DSP 1.1 are mostly real-time by nature and are carried out in close associations to call signaling and control states and operations.

This document describes the control and data interfaces of DSP 1.1 so that third-party developers can incorporate the module into a media-gateway or server system and integrate it with a client software. The document provides sufficient details of these interfaces so that a DSP 1.1 software client can fully configure and control DSP 1.1's processing operations and services.

Additionally, the document describes the data interface, format, and message- and data-delivery mechanisms in DSP 1.1.

1.2 Scope

The DSP 1.1 API is a set of functions, macros, and message and packet formats that determines how the applications access the media-processing resource components in DSP 1.1.

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 Related Documentation

Title	Document Number
Intel® IXP400 Digital Signal Processing (DSP) Software Library Release 1.1 Programmer's Guide	252725
Intel® IXP400 Digital Signal Processing (DSP) Software Library Release 1.1 Software Release Notes	273810
Intel® IXP400 Product Line Programmer's Guide (Version 1.1)	252539

2.0 Architecture Overview

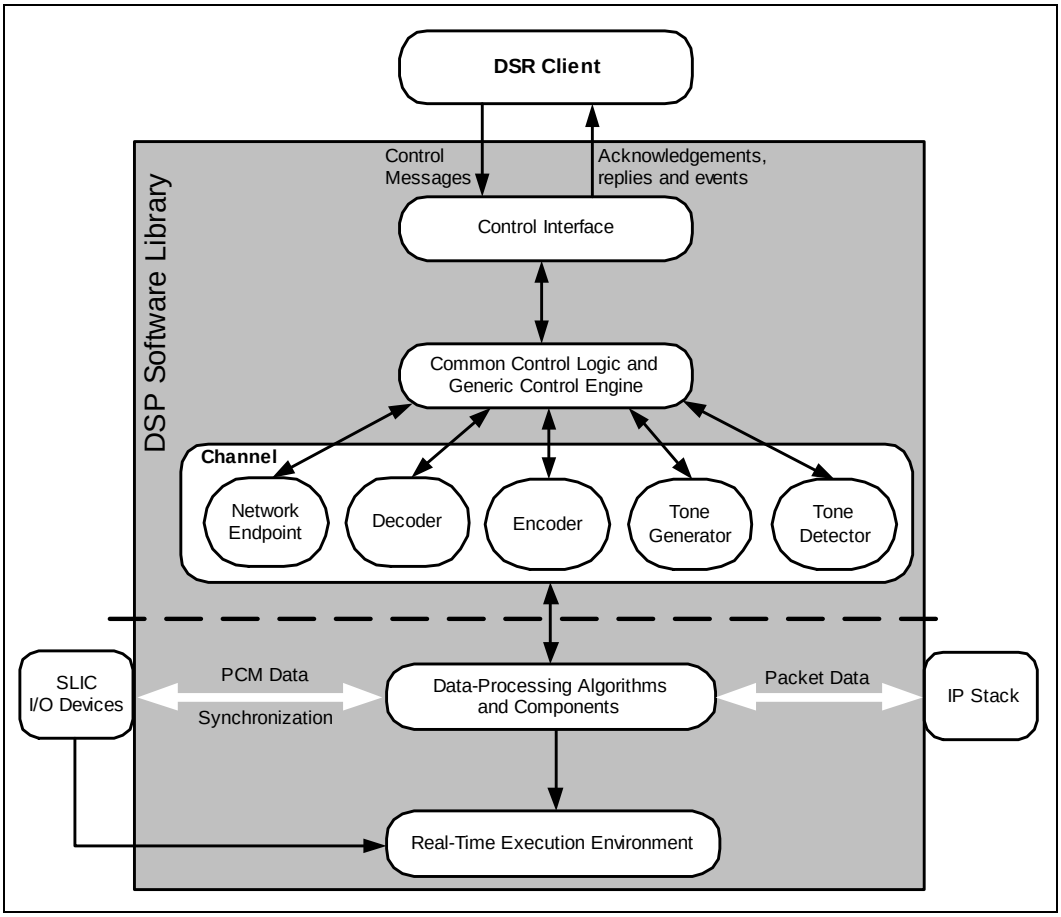
The DSP 1.1 is implemented as an independent module having its own tasks and run-time environment. The software architecture has a two-layer hierarchy:

- A control layer that provides the control interface and control logic
- A data-processing layer where the media data streams are processed by appropriate algorithms

Figure 1 shows the decomposition of the DSP 1.1 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. Intel® IXP400 DSP Software Library Release 1.1 Software Architecture



3.0 Media-Processing Resource Components

The addressable control entities of the DSP 1.1 are media-processing resource (MPR) components. DSP 1.1 has five 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 in Figure 2. In the remainder of this document, each of these five media processing entities are referred to as a resource or a resource component.

Figure 2. Resource-Component Identifiers

```
typedef enum{
    XMPR_ANY=0,   /* Any resource, not supported in 1.1 */
    XMPR_NET,     /* Network Endpoint resource */
    XMPR_DEC,     /* Decoder resource */
    XMPR_ENC,     /* Encoder resource */
    XMPR_TNGEN,   /* Tone generator resource */
    XMPR_TNDET,   /* Tone detector 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-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 resources while others are unique only to a particular resource.

The following sections describe the each resource in terms of their identifiers, media processing functions, parameters, and control messages.

3.1 Network Endpoint Resource Component

Resource Type	XMPR_NET
Media Processing Functions	**Resource-Specific Control Messages**
• A-law or µ-law compression and decompression • High-Pass Filter • Echo Cancellation (EC)	None

Parameters

Identifier	Description, Values	Attr.
XPARMID_NET_TX	Tx time slot ID on HSS TDM bus. Default: Sequentially assigned	R
XPARMID_NET_RX	Rx time slot ID on HSS TDM bus. Default: Sequentially assigned	R
XPARMID_NET_LAW	PCM data format on HSS TDM bus. XPARM_NET_ALAW or XPARM_NET_MULAW. Default: XPARM_NET_MULAW	R/W

Resource Type	XMPR_NET	
XPARMID_NET_ECENABLE	EC enabling flag, XPARM_ON or XPARM_OFF. Default: XPARM_ON	R/W
XPARMID_NET_ECTAIL	EC tail length (2, 4, 6, 8,…up to 16 in 1 ms unit). The resource must be reset after setting the parameter. Default: 6	R
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
XPARMID_NET_ECFREEZE	EC freezing flag, XPARM_ON (freeze) or XPARM_OFF (adaptive). Default: XPARM_OFF	R/W
XPARMID_NET_DELAYCOMP	EC delay compensation (0 ~ 240 in 0.125 ms unit). Default: 32 (32 generates 32 x 0.125 ms = 4.0 ms tail length)	
Events		
None		

3.2 Decoder Resource Component

Resource Type	XMPR_DEC
Media Processing Functions	**Resource-Specific Control Messages**
• Decoding • Automatic level control and/or volume control • Comfort noise generation • Jitter compensation	XMSG_CODER_START

Parameters

Identifier	Description, Values	Attr.
XPARMID_DEC_VOL	Decoder volume adjustment; +6 ~ -20 in 1-dB units. Default: 0 (Not implemented in DSP 1.1.)	R/W
XPARMID_DEC_ALC	ALC enable flag. XPARM_ON or XPARM_OFF. Default: XPARM_ON	R/W
XPARMID_DEC_CNG	CNG enable flag. XPARM_ON or XPARM_OFF. Default: XPARM_OFF	R/W
XPARMID_DEC_CTYPE	Coder type. Default: N/A	R
XPARMID_DEC_EVT_PKT	Enable packet lost event. XPARM_ON or XPARM_OFF. Default: XPARM_OFF	R/W
XPARMID_DEC_EVT_CTM	Enable codec-type-mismatch event. XPARM_ON or XPARM_OFF. Default: XPARM_OFF (Not implemented in DSP 1.1.)	R/W

Resource Type	XMPR_DEC	
XPARMID_DEC_JB_MAXDLY	Jitter buffer maximum delay in ms. Default: 200	R/W
XPARMID_DEC_JB_PLR	Jitter buffer packet loss rate in 0.1% unit. Default: 1	R/W
Events		
XEVT_LOST_PACKET – lOst packet.		
XEVT_DEC_CT_MISMATCH – Received a packet whose code type mismatches decoder's code type.		

3.3 Encoder Resource Component

Resource Type	XMPR_ENC	
Media Processing Functions	**Resource-Specific Control Messages**	
• Encoding • Automatic Gain Control • Voice-Activity Detection	XMSG_CODER_START	
Parameters		
Identifier	**Description, Values**	**Attr.**
XPARMID_ENC_AGC	AGC enable flag. XPARM_ON or XPARM_OFF. Default: XPARM_OFF	R/W
XPARMID_ENC_VAD	VAD enable flag. XPARM_ON or XPARM_OFF. Default: XPARM_OFF	R/W
XPARMID_ENC_CTYPE	Coder type. Default: N/A	R
XPARMID_ENC_EVT_PKT	Enable packet lost event. XPARM_ON or XPARM_OFF. Default: XPARM_OFF	R/W
Events		
XEVT_LOST_PACKET – Lost packet		

3.4 Tone Generation Resource Component

Resource Type	XMPR_TNGEN
Media Processing Functions	**Resource-Specific Control Messages**
• Generating multiple-frequency tone signals • Generating call-progress tones	XMSG_TG_PLAY XMSG_TG_PLAY_FSK

Parameters

Identifier	Description, Values	Attr.
XPARMID_TNGEN_VOL	Decoder volume adjustment; +6 ~ -20 in 1-dB units. Default: 0	R/W
XPARMID_TNGEN_FSK_MOD	FSK modulator mode. XPARM_TNGEN_FSK_V23 or XPARM_TNGEN_FSK_B202. Default: XPARM_TNGEN_FSK_V23	R/W
XPARMID_TNGEN_FSK_CS	CS bit length of FSK modulator (in bit unit). Default: 0	R/W
XPARMID_TNGEN_FSK_MARK	Mark bit length of FSK modulator (in bit unit). Default: 100	R/W
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

Events

None

3.5 Tone Detection Resource Component

Resource Type	XMPR_TNDET
Media Processing Functions	**Resource-Specific Control Messages**
• Receiving continuous tone sequence • Detecting individual tone event	XMSG_TD_RCV (Not supported in DSP 1.1.)

Parameters

Identifier	Description, Values	Attr.
XPARMID_TD_TC	Tone Clamping enable flag. XPARM_ON or XPARM_OFF. Default: XPARM_OFF	R/W
XPARMID_TD_TC_FRAMES	Tone Clamping buffer size. 0 ~ 3 in 10-ms units. Default: 3	R/W

Resource Type	XMPR_TNDET	
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
Events		
XEVT_CODE_TD_TONEON – Tone-on event		
XEVT_CODE_TD_TONEOFF – Tone-off event		
NOTE: Event *data1* gives the tone ID.		

4.0 Message Format and Delivery Mechanism

There are two message queues — one for *sending* messages to DSP 1.1 and the other for *receiving* messages from DSP 1.1. The message queues are created from pre-allocated memory buffers in consideration of the 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 DSP and other applications.

4.1 Message Functions

Three functions are provided to send and receive messages.

XStatus_t xMsgSend (void *pMsgBuf);	
Description	Sends a message to DSP-inbound 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 DSP-outbound message queue.
Input	• pMsgBuf – Pointer to the message buffer • channel – Channel number. (Reserved for future extension) • timeout – Time to wait. — `XWAIT_NONE` — If return immediately — `XWAIT_FOREVER` — If never time out
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); **(Sheet 1 of 2)**	
Description	Posts a message to DSP-outbound, user-defined message queue, so that it can be retrieved by `XMsgReceive()`.
Input	pMsgBuf — Pointer to the message buffer.
Output	None

XStatus_t xMsgWrite (void *pMsgBuf); (Sheet 2 of 2)	
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 channel; /* channel ID (1-0xffff), 0:all channels */ 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, ch, sz, typ, attr) \ ((XMsgRef_t)(pMsg))->transactionId = trans;\ ((XMsgRef_t)(pMsg))->channel = ch;\ ((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 SP 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 set 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 channel */
    XMSG_TG_PLAY_FSK,       /* play FSK modulated data */
    XMSG_TG_PLAY_CMPLT,     /* play-completed message from a channel */
    XMSG_TD_RCV,            /* receive a digit string on a channel */
    XMSG_TD_RCV_CMPLT,      /* receive-completed message from a channel */
    XMSG_END                /* end of list */
} XMsgType_t;
```

5.0 Common Control Message

This section defines the control messages that can be applied to all 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_thead; /* message header */`
	`} XMsgReset_t;`

Macro	`#define XMSG_MAKE_RESET(pMsg, trans, res, ch) \`
	`  {\`
	`  XMSG_MAKE_HEAD(pMsg, trans, res, ch, sizeof(XMsgReset_t),\`
	`  XMSG_RESET, 0)\`
	`  }`

Response	General acknowledgement message (XMSG_ACK)
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_thead; /* message header */`
	`} XMsgStart_t;`

Macro	`#define XMSG_MAKE_START(pMsg, trans, res, ch) \`
	`  {\`
	`  XMSG_MAKE_HEAD(pMsg, trans, res, ch, sizeof(XMsgStart_t),\`
	`  XMSG_START, 0)\`
	`  }`

Response	General acknowledgement message (XMSG_ACK)
Caution	Some resources only support resource-specific start messages.

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_START(pMsg, trans, res, ch)\`
	`{\`
	`XMSG_MAKE_HEAD(pMsg, trans, res, ch, 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_thead; /* message header */`
	`} XMsgPing_t;`

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

Response	General acknowledgement message (XMSG_ACK)

5.5 Set Parameter Message

Type	**XMSG_SET_PARM**
Direction	Inbound
Description	Sets a parameter to a resource.

Format	`typedef struct{`
	`  XMsgHdr_t head; /* message header */`
	`  UINT16parmId;/* parameter id */`
	`  UINT16value;/* parameter value */`
	`  } XMsgSetParm_t;`

Macro	`#define XMSG_MAKE_SET_PARM(pMsg, trans, res, ch, id, val) \`
	`{\`
	`  XMSG_MAKE_HEAD(pMsg, trans, res, ch, sizeof(XMsgSetParm_t),\`
	`  XMSG_SET_PARM, 0)\`
	`  ((XMsgSetParm_t *)(pMsg))->parmId= id;\`
	`  ((XMsgSetParm_t *)(pMsg))->value= val;\`
	`}`

Response	General acknowledgement message (XMSG_ACK)

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, ch, num) \`
	`{\`
	`  XMSG_MAKE_HEAD(pMsg, trans, res, ch, 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)

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 */`
	`  UINT16parmId;/* parameter id */`
	`} XMsgGetParm_t;`

Macro	`#define XMSG_MAKE_GET_PARM(pMsg, trans, res, ch, id) \`
	`{\`
	`  XMSG_MAKE_HEAD(pMsg, trans, res, ch, sizeof(XMsgGetParm_t),\`
	`  XMSG_GET_PARM, 0)\`
	`  ((XMsgGetParm_t *)(pMsg))->parmId= id;\`
	`}`

Response	Specific acknowledgement message (XMSG_GET_PARM_ACK)

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;\`
	`}`

Response	

5.9 Get All Parameters Message

Type	**XMSG_GET_ALLPARMS**
Direction	Inbound
Description	Gets all parameters from a resource.

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

Macro	`#define XMSG_MAKE_GET_ALLPARMS(pMsg, trans, res, ch) \`
	`{\`
	`  XMSG_MAKE_HEAD(pMsg, trans, res, ch, 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 enquired.

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;\`
	`}`

Response	

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;`

Macro	
Response	

5.12 Error Message

Type	**XMSG_ERROR**
Direction	Outbound
Description	Resource reports an error condition.

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;\`
	`  }`

Response	

5.13 Event Message

Type	**XMSG_EVENT**
Direction	Outbound
Description	Resource reports an event condition.

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;\`
	`  }`

Response	

6.0 Resource Specific Control Message

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  frmsPerPkt;  /* number of frames per packet */`
	`  UINT16  codecType;  /* codec type */`
	`} XMsgCoderStart_t;`

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

Response	General acknowledgement message (XMSG_ACK)

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;\`
	`  }`

Response	

6.3 Tone Generator Play Message

Type	**XMSG_TG_PLAY**
Direction	Inbound
Description	Requires Tone Generator to play a tone string.

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

Response	

6.4 Tone Generator Play FSK Message

Type	**XMSG_TG_PLAY_FSK**
Direction	Inbound
Description	Requires Tone Generator to play 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, ch, num)\`
	`  {\`
	`  XMSG_MAKE_HEAD(pMsg, trans, XMPR_TNGEN, ch, 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;\`
	`  }`

Response	

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  timeout;  /* time out (in 10 ms unit) */`
	`  UINT16  firstToneTimeout /* first tone time out (10 ms uint)*/`
	`  UINT16  interToneTimeout,/* inter tone time out (10 ms unit)*/`
	`  UINT8  termTone;  /* terminate tone */`
	`  UINT8  numTones;  /* number of tones to receive */`
	`} XMsgTDRcv_t;`

Macro	`#define XMSG_MAKE_TD_RCV(pMsg, trans, ch, num, term, tm, fstTm, intTm)\`
	`  {\`
	`  XMSG_MAKE_HEAD(pMsg, trans, XMPR_TNDET, ch, sizeof(XMsgTDRcv_t),\`
	`  XMSG_TD_RCV, 0)\`
	`  ((XMsgTDRcv_t *)(pMsg))->numTones = num;\`
	`  ((XMsgTDRcv_t *)(pMsg))->termTone = term;\`
	`  ((XMsgTDRcv_t *)(pMsg))->timeout = tm;\`
	`  ((XMsgTDRcv_t *)(pMsg))->firstToneTimeout = fstTm;\`
	`  ((XMsgTDRcv_t *)(pMsg))->interToneTimeout = intTm;\`
	`  }`

Response	Tone detector receives completed message (XMSG_TD_RCV_CMPLT)
Note	Not supported in Intel® IXP400 DSP Software Library Release 1.1.

6.7 Tone Detector Receive-Completed Message

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

Format	`typedef struct{`
	`  XMsgHdr_t  head;  /* message header */`
	`  UINT16  reason;  /* the reason of completion */`
	`  UINT8  numTones;  /* number of tones received */`
	`  UINT8  toneId[XMAX_TONEBUFSIZE];/* received tone IDs */`
	`} XMsgTDRcvCmplt_t;`

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

Response	
Note	Not supported in Intel® IXP00 Product Line Software Release 1.1.

7.0 Packet Data Interface

The packet data interface is a protocol for DSP 1.1 to exchange the encoded data packets with the IP stack. This interface is defined as a packet format and two call-back functions — one is provided by DSP and another by the user (IP stack).

7.1 Packet Formats

The inbound packet from the IP stack to DSP 1.1 has an 8-byte header as shown in the following table. Remote Time Stamp, Payload Type, and Marker (M) bits are directly copied from an RTP packet.

31							24	23	22						16	15			12	11			8	7							0
Payload Length								M*	Payload Type							(Reserved)				Media				Channel ID							
Remote Time Stamp																															
: Payload :																															
NOTE: M is for Marker bit.																															

The outbound packet (from DSP 1.1 to the IP stack) has an 8-byte header that is shown in the following table. Local Time Stamp, Payload Type, and Marker (M) bits are directly copied from an RTP packet.

31							24	23	22						16	15			12	11			8	7							0
Payload Length								M*	Payload Type							(Reserved)				Media				Channel ID							
Local Time Stamp																															
: Payload :																															
NOTE: M is for Marker bit.																															

The fields of the packet header are described in the following table:

Field	Description
Local Time Stamp	Packet arrival time as measured by a local clock in samples.
Remote Time Stamp	Packet data sampling time measured by a remote clock in samples.
Payload Length	Payload length in bytes.
Media	4-bit media type field is defined as: 0x0001 – Audio 0x0100 – Tone (RFC2833 event type) 0x0101 – Tone (RFC2833 tone type) Others – Reserved
M	Marker bit indicating the first speech frame after a silence period.
Payload type	RTP payload type as defined in RFC 1990.

The corresponding data structure is defined as:

```
typedef struct{
    UINT8    payloadLen;      /* payload length */
    UINT8    payloadType;     /* bit[0-6] payload type, bit[7] SID mark bit */
    UINT8    mediaType;       /* bit[0-3] media type, bit[4-7] reserved *
    UINT8    channelID;       /* channel ID */
    UINT32   timeStamp;       /* local or remote time stamp */
} __attribute__ ((packed)) XPacketHeader_t;
```

7.2 Packet-Delivery Mechanism

The packets are transferred between DSP 1.1 and IP stack via the call-back functions. The packet delivery module calls the function and passes the packet each time a packet is produced.

The call-back function's rules for delivering the packets include:

- The packet receiver registers a call-back function with the packet deliverer.
- The packet deliverer is responsible for preparing the memory for the packet.
- The packet receiver has to immediately copy the data to its internal buffer, in the call-back function, because the deliverer may reuse the same memory for the next packet. That could result in the packet data not being valid, after the call-back function returns.
- The packet receiver may perform some data processing in the call-back function, provided the execution of such processing is predictable. The processing must be guaranteed to complete within a certain, short period of time.

To receive the packet from the IP stack, DSP provides the function implemented as:

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

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

In egress direction, the IP stack must provide such function to receive from the DSP. DSP will call the function each time, when a packet is generated. That function must be registered with DSP during initialization, by using:

XStatus_t xRegistPktRcvFxn(XPktRcvFxn_t ipReceiveCB);	
Description	Register callback function with DSP.
Input	`ipReceiveCB` – The callback function for IP stack to receive data packets from DSP.
Output	None
Return	`XSUCC` – If successful `XERROR` – If failure.

8.0 Configuration

The DSP 1.1 module must be initialized and configured before any of its functions can be used. There are three functions for this purpose.

8.1 DSP Initialization

Prototype	`Xstatus_t xDspInit(void);`
Input	None
Output	None
Return	XSUCC — If successful XERROR — Otherwise
Description	This function allocates the memory and creates tasks for DSP 1.1 internal software components. The client application must call this function only once after the system boots up and before other DSP 1.1 functions are called.

8.2 HSS Port Initialization

Prototype	`XStatus_t xDspbHssInit(IxHssAccHssPort port,`
	`IxHssAccConfigParams *pConfigParams,`
	`IxHssAccTdmSlotUsage *pTDMSlotMap);`

Input	port – HSS port ID pConfigParms — Pointer to HSS configuration-parameter structure pTDMSlotMap — Pointer to HSS time-slot-map information
Output	None
Return	XSUCC — If successful XERROR — Otherwise
Description	This function initializes, starts, and connects the channelized HSS port to DSP 1.1. It must be called after downloading HSS NPE. See the *Intel® IXP425 Network Processor Based on Intel® XScale™ Microarchitecture Programmer's Guide* (11726, v 1.0) for more details.

8.3 Setting Country Code

Prototype	`XStatus_t xSetCountryCode(int countryCode);`
Input	countryCode – country code
Output	None
Return	XSUCC — If successful XERROR — Otherwise
Description	This function sets the country for the country-specific features such as call-process tone generation. The default country code is set for Japan. An invalid country code may cause the country-specific features to be disabled.

9.0 Error Code and Constant Data

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

9.1 Error Code

Errors are reported via an XMSG_ERROR message with an error code and two pieces of 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 instance id */
#define XERR_INVALID_PARM_ID     0x0013   /* invalid parameter id */
#define XERR_PARM_READONLY       0x0014   /* real only parameter */
#define XERR_PARM_SET_FAIL       0x0015   /* cannot set the parameter */
#define XERR_PARM_GET_FAIL       0x0016   /* cannot get parameter */
#define XERR_UNEXPECTED_MSG      0x0017   /* unexpected message */
#define XERR_UNSUPPORTED_MSG     0x0018   /* 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 */
```

9.2 Constant Data

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_C          13
#define RFC_TID_DTMF_B          14
#define RFC_TID_DTMF_D          15
```

The 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
```