Intel® IXP400 Digital Signal Processing (DSP) Software Version 2.5

Programmer's Guide

December 2004

Document Number: 252725-004a

Contents

Figures

Tables

(No numbered tables.)

Revision History

Date	Revision	Description
December 2004	005	Updated product branding.
June 2004	004	Document updated for release of Intel® IXP400 DSP Software v.2.5.
January 2004	003	Document updated for release of Intel® IXP400 DSP Software v.2.4.
September 2003	002	Document updated for release of Intel® IXP400 DSP Software v.2.3.
March 2003	001	Initial release of this document.

Introduction 1

Intel® IXP400 DSP Software is a software module that provides the basic voice and signal processing functionalities for voice-over-Internet-protocol (VoIP) on Intel® IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor.

This document explains how to use the *Intel® IXP400 Digital Signal Processing (DSP) Software Version 2.5 API Reference Manual* and provides guidelines and examples to the application developers.

1.1 General

The Intel® IXP400 DSP Software is a software module for media processing, targeted for next-generation Integrated Access Devices (IADs) such as Consumer Premise Equipment (CPE), specifically, to perform media compression, echo cancellation, tone processing, and jitter control, 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 DSP software into a media gateway or server system and integrate it with other client software. Together with the *Intel® IXP400 Digital Signal Processing (DSP) Software Version 2.5 Programmer's Guide*, this document provides sufficient details of the interfaces and message and data-delivery mechanisms that the user applications can fully configure and control the processing operations and services.

This release of DSP Software supports the following features:

- G.729ab — G.729a with VAD and CNG support
- G.711 µ-law and A-law CODEC with 10-ms frame size
- G.711 Annex 2. Support for VAD and CNG
- G.723.1 with 5.3 and 6.3 Kbps rates
- G.722
- G.726 with the rates of 16, 24, 32 and 40 Kbps
- Packet loss concealment (PLC) for G.711 G.726, and G.722
- Configurable PCM interface in the wideband or narrowband mode
- Dynamically switch coder types on the fly
- Automatically switch decoder types according to the received RTP packets
- Support multiple frames per packet. The maximum numbers of frames per packet are:
  - 6 for G.711 and G.722
  - 8 for G.723
  - 9 for G.726 40 Kbps
  - 12 for G.726 32 Kbps

  - 16 for G.726 24 Kbps
  - 24 for G.729 and G.726 16 Kbps
- Dynamically changing the frames per packet on the fly
- Dynamically routing the audio streams between any resource components
- Support Automatic Gain Control (AGC) for encoder with provision for manual setting with mute
- Support Automatic Level Control (ALC) for decoder with provision for manual setting with mute
- Echo cancellation
- DTMF generation and detection
- Receiving DTMF digit input
- Fax-tone detection
- Modulated-tone generation capability
- Detection and generation of user-specified tones
- FSK modem signal generating and receiving for caller ID
- United States, China[1], and Japan call-progress tone generation
- Dynamic DTMF tone clamping
- RFC 2833 tone event support for DTMF with variable frame rate
- Dynamic/Adaptive Jitter Buffer algorithm
- Audio mixer for three-way call and small conference (up to five parties)
- Audio player for voice prompts, on-hold music, etc. (playing back G.711 or G.729 encoded data).
- Low-latency TDM switching
- Digital gain control at the front end
- User-defined control interface

1.2 Scope

The *Intel® IXP400 Digital Signal Processing (DSP) Software Version 2.5 API Reference Manual* specifies how user can interface to the DSP software. This document provides more application information on how the interface can be effectively used. Some examples are given for illustration purposes. Details on pre-defined user messages, which are not part of the core DSP software but are provided to help ease integration, are also given here.

1. In this document, all references to China refer to the People's Republic of China.

1.3 Audience

This document is intended for third-party software developers who are using the Intel® IXP400 DSP Software to build a gateway or server application. It is assumed that the reader has general knowledge of VoIP applications and products.

1.4 Related Documents

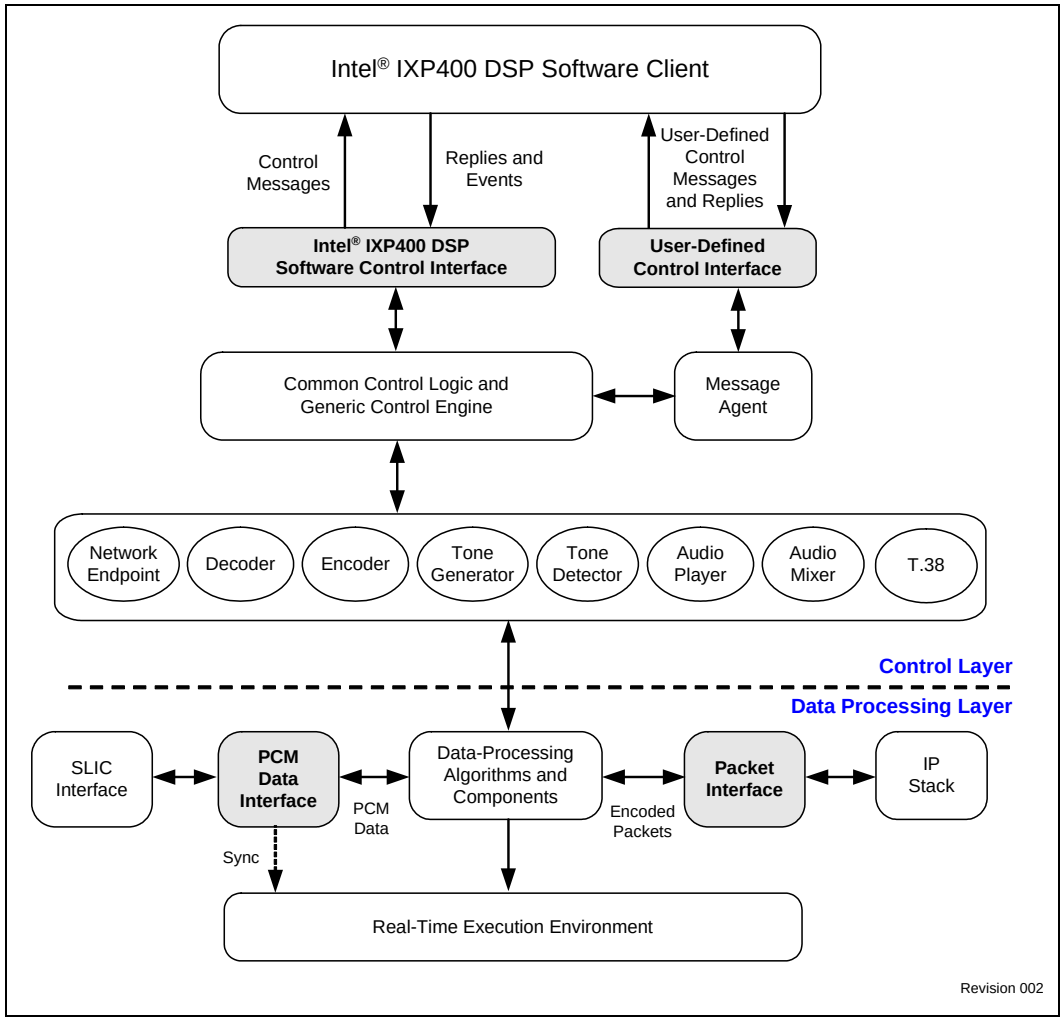
Document	Document Number
Intel® IXP400 Digital Signal Processing (DSP) Software Version 2.5 API Reference Manual	273811
Intel® IXP400 Digital Signal Processing (DSP) Software Version 2.5 Release Notes	N/A
Intel® IXP400 Digital Signal Processing (DSP) Software Specification Update	273810
Intel® IXP400 Software Programmer's Guide	252539

Architecture Overview 2

The 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 handles 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 logic decomposition of the DSP software modules, where the shaded blocks represent the control and data interfaces between the DSP software and other software modules.

Figure 1. Architecture of Intel® IXP400 DSP Software



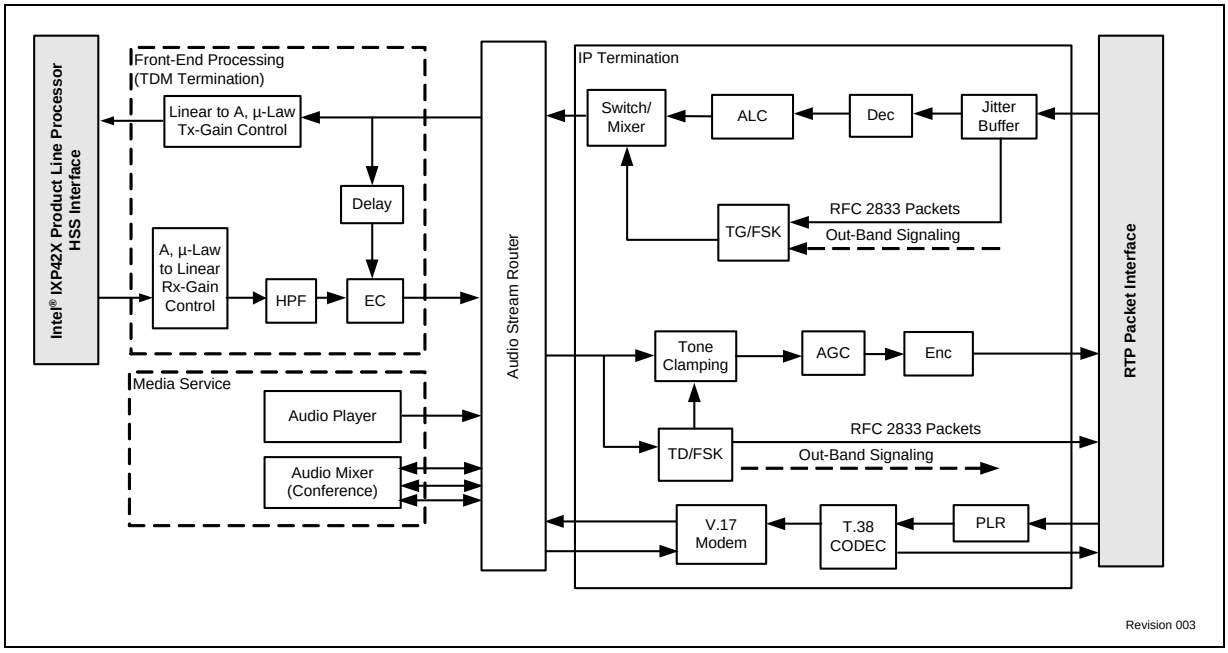
From the control point of view, a DSP software channel consists of a set of Media Processing Resource (MPR) components. Each MPR is an addressable entity and can be controlled independently. That gives the maximum flexibility of setting up a channel with various resource configurations, e.g., half-duplex call or asymmetry Rx and Tx CODEC types, if necessary.

From the perspective of data flows, the data processing functions are depicted in Figure 2. All the functions are executed by real-time tasks (or threads) created during initialization. There is one task for each unique coder frame rate. Currently there is a 10-ms task for G.711 and G.729 coders and a 30-ms task for the G.723 coder, fax modem, and T.38 engine.

The 10-ms task also handles all other non-coder voice processing, such as echo cancellation and tone detection. The real-time tasks are of higher priority than the control task and are synchronized (triggered) by the Network Processing Engine (NPE) of the High Speed Serial (HSS) port in the IXP42X product line processors.

Some of the necessary input and output functions are also performed in the context of the real-time tasks. This includes buffering of data to and from the HSS interface, and the external function registered toDSP software to encode the DSP software's packets into RTP format for forwarding to the IP stack.

Figure 2. Data-Flow and Data-Processing Functions



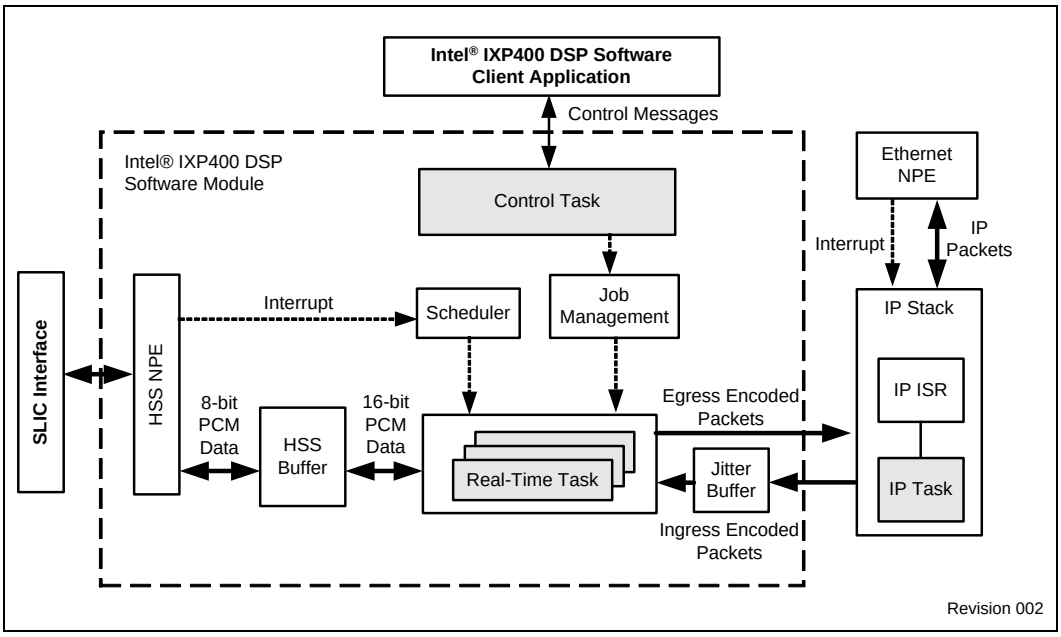
The relation among the messages, data and tasks inside and outside the DSP software, is illustrated by Figure 3 on page 11 and can be summarized as:

- The control task is driven by the in-bound messages from the user application.
- The real-time tasks are synchronized with the data from HSS interface. HSS NPE signals the scheduler via an interrupt service routine (ISR) every 10 ms. The scheduler triggers the real-time tasks according to the algorithms executed by the tasks.
- Real-time tasks generate and consume the encoded audio packets at the fixed rates essentially synchronized with PCM data.
- The encoded audio packets arrive at variable rate asynchronously with the real time tasks.

Note: It is important to understand that the internal, real-time tasks are characterized by their hard task deadlines. That means if a real task cannot finish its processing before the next task period, data

will be lost and consequently voice quality is degraded seriously. That may happen if the real-time task is preempted by ISR or other tasks for a long time or simply the processor is overloaded.

Figure 3. Intel® IXP400 DSP Software Message, Data, and Tasks



Run-Time Interfaces 3

The Intel® IXP400 DSP Software is implemented as an independent module executed by its own tasks. User applications do not directly access the internal functions or data.

The DSP software provides three interfaces for the applications to communicate control information, PCM data, and encapsulated voice packets, respectively, in run-time as shown in Figure 1 and Figure 2 on page 10.

3.1 Control Interface

The applications primarily communicate with DSP software through the control interface defined as a set of functions, messages and macros.

There are two message queues in the control interface for the in-bound messages from applications to the DSP software and the out-bound messages in the other direction. Refer to Figure 4 on page 14. Two interface functions, `xMsgSend()` and `xMsgReceive()`, can be used for the application to send and receive messages to/from the queues, respectively.

The DSP software spawns a dedicated control task pending on the in-bound message queue to handle the control messages. The reason of isolating the DSP software software from user applications by message queues is to avoid the internal control functions being accessed by multiple tasks of the user application, since making the control functions multi-task-safe creates extra complexity and subsequent performance penalties.

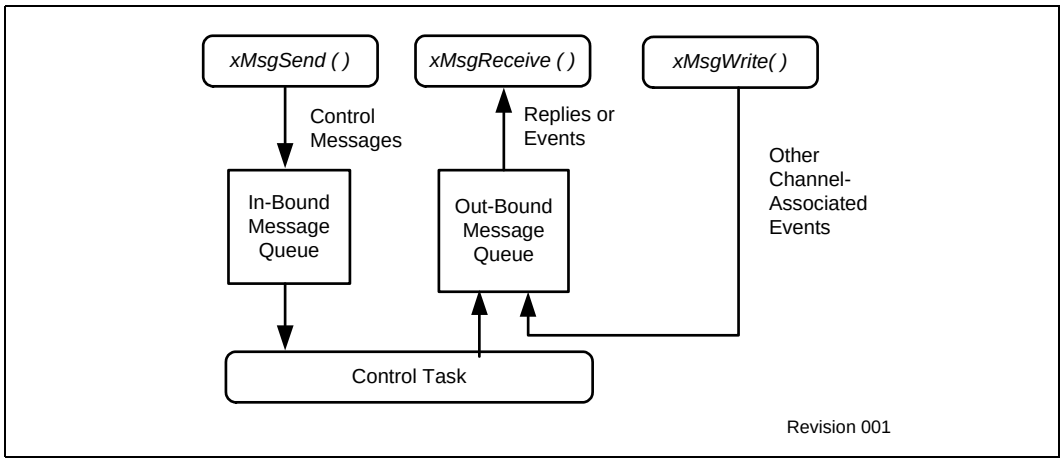
TheDSP software sends replies or events to the application through the out-bound message queue. The application can retrieve the messages using `xMsgReceive()`. The caller's task of `xMsgReceive()` will be blocked forever or until timeout if the out-bound queue is empty.

A third function for the control interface, `xMsgWrite()`, allows the application to directly post external messages to the out-bound message queue back to the user application if necessary. This enables the user application to receive all channel-associated events from one place, even though some of these events are external to the DSP software. For instance, the application may hook a callback function to the ISR that reports the SLIC interface on/off hook events. In the callback function, an external event message as defined by the user is sent to the out-bound message queue to signal the event to the user application.

Because of the limitation of the queue lengths, the queues may overflow and the messages may be lost if the application keeps sending messages without waiting for the replies. In this case, the in-bound queue may overflow if the user application is of higher priority than the DSP software control task, or the out-bound queue may overflow if the user application has lower priority.

Copy-based message delivery is used. That is, the entire message context is actually copied from the deliverer to the receptor rather than passing a pointer around. This avoids dynamically allocating memory for the messages. Since no memory is shared between the DSP software and the application, the application can reuse the memory of a message for any other purpose immediately after the message is sent. On the other hand, to receive a message the application is responsible for preparing the memory that must be able to accommodate the maximum message size with the alignment at 4-byte boundary.

Figure 4. Control Interface and Message Queues



The message format consists of an 8-byte message header plus an optional message payload. The message header contains the common information like channel ID, MPR ID, type, size, etc. A 4-byte transaction ID is provided to allow the user application to keep track of the replies or events. When the DSP software sends a reply or event message to the user application, it copies the transaction ID from the associated message originated from the user application. Refer to the *Intel® IXP400 Digital Signal Processing (DSP) Software Version 2.5 API Reference Manual* for details of the control message format.

3.2 PCM Data Interface

PCM data represents the audio data stream between theDSP software and the telephone interface via the TDM data bus. The PCM data interface relies on the HSS hardware integrated in the IXP42X product line processors.

In contrast to the data network interface, such as the Ethernet interface, the HSS interface is integrated as part of the DSP software. This allows the most efficient transfer of real-time PCM data input since no other application is expected to need this data directly. The user application, however, controls how the HSS is being configured, by parameters being passed to the DSP software during initialization.

From the user application's perspective, the HSS can be viewed as a piece of hardware to be properly configured, to interoperate with the external, customer-specific interface connected to it. Once it is configured and started, there is no further user application involvement.

The user application configures the HSS by specifying the signal format to be presented on TDM bus of the HSS device, including the clock rate, time slots, frame sync, endian, etc. Such information is organized in two data structures:

- IxHssAccConfigParams
- IxHssAccTdmSlotUsage

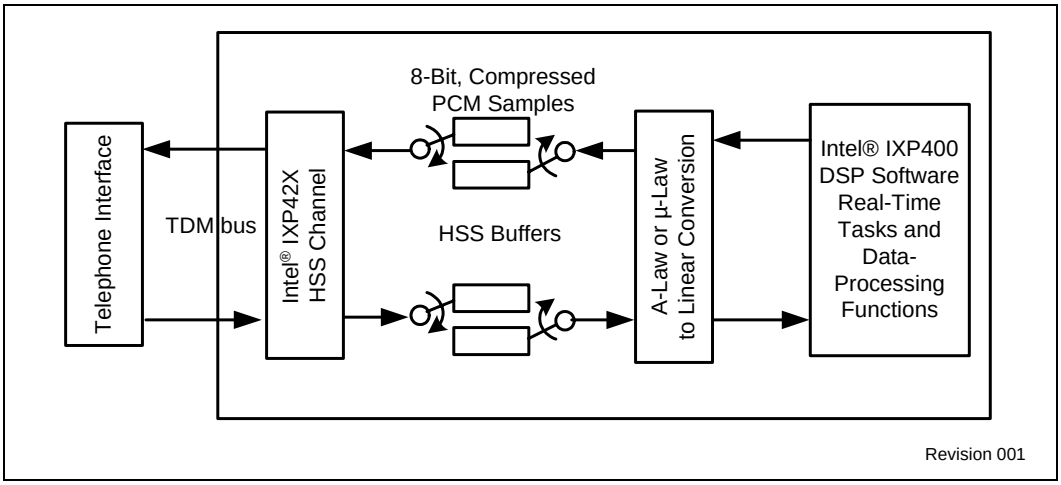
Using this set of information, theDSP software initializes the HSS interface and starts data transfers. Refer to the *Intel® IXP400 Software Programmer's Guide* for details.

The DSP software supports dual-band PCM interface over the HSS. In the narrowband mode, the PCM data format is 8-bit A-law or µ-law compressed data at an 8-KHz sampling rate. In the wideband mode, it is 16-bit linear data at 16-KHz sampling rate. In order to share the TDM bus of the HSS, a wideband audio channel takes four time slots at the 8-KHz frame rate. The user applications need to specify how those time slots are located if a channel is configured to wideband mode during the system initialization. Sampling rate conversion (SRC) is applied automatically if a wideband channel is connected to a narrowband media processing resource or vice versa. The superior voice quality can be expected only when both the interface and the resources operate in the wideband mode.

The user application may enable more HSS time slots than the number of channels supported by the DSP software. In this case the time slots are connected to the channels from the first one sequentially and the extra time slots are ignored. To use the low latency time slot switch feature, at least eight time slots must be enabled. (See "Audio Stream Router" on page 27.)

Internally, the real-time tasks are synchronized with HSS data transfer — the scheduler being signaled by the HSS driver (in an interrupt context) each time when certain amount (10 ms) of data is transferred. The real-time tasks may not be invoked at all if the HSS interface is not configured and started properly.

Figure 5. PCM Data Interface



3.3 Packet Interface

Compared to PCM Data Interface, the Packet Interface is a pure software protocol that defines how the encoded audio data packets are exchanged between the DSP software and the IP interface.

There are two functions and a packet format involved in the Audio Packet Interface as shown in Figure 6 on page 16. The DSP software defines the packet format and provides the packet receive function. The user application is responsible for providing the transmit function.

In ingress (packets coming from the IP interface), the IP interface converts each incoming VoIP packet it receives to anDSP software data packet and then calls `xPacketReceive()` to deliver it to the DSP software. The user application needs to decode the incoming IP packets to forward the RTP packet payloads with the proper DSP software header format, with the extracted RTP timestamp, to the proper DSP software channel.

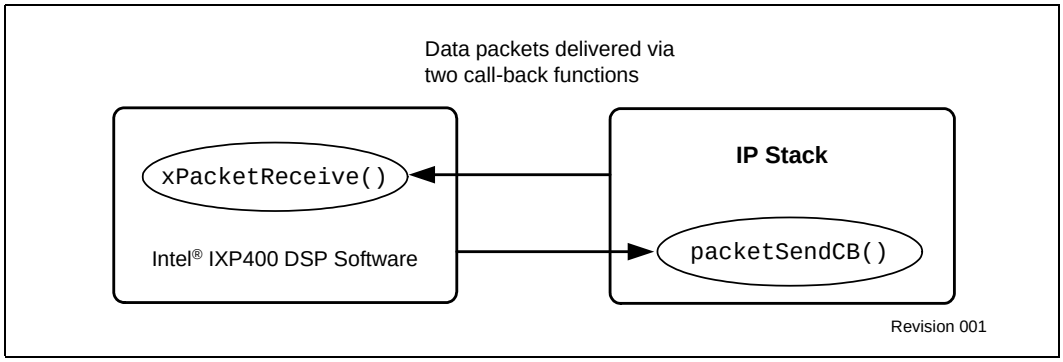
The function `xPacketReceive()` copies the packet to the jitter buffer without further processing. Therefore `xPacketReceive()` can be called from an Interrupt Service Routine context but re-entry is not allowed. Since the packets are copied by the DSP software, the caller of the `xPacketReceive()` can free or reuse any memory it may have allocated to buffer the incoming RTP packets upon return from the function.

In egress (packets going to the IP interface), through `xDspSysInit()`, the application registers a callback function with the DSP software. This callback function is supposed to deliver the data packet to the IP interface and sends it out. The DSP software always prepares the memory for the packet and fills the packet header information (including local time stamp) and packet payload before it calls the function. This user-provided function should create and encode the RTP header with the time stamp in the data packet supplied by the DSP software. After returning from the function, the DSP software will immediately reuse the memory for other purpose. Therefore, it may be necessary for the callback function to make a copy of the packet.

Since the function is called from the internal real-time tasks at regular basis each time when a packet is generated, there are two additional requirements for the callback function:

- It must finish as soon as possible without any blocks inside (to allow real-time data to be acquired and processed without data loss)
- It must be multi-task-safe (it must allow re-entry)

Figure 6. Packet Interface



Components, Features, and Parameters 4

An Intel® IXP400 DSP Software channel consists of several media processing resource (MPR) components, which can be addressed independently by the application. Each component has its particular processing functions and features that are controlled by the messages and parameters. In this section, we will discuss the MPR components and their features and parameters.

4.1 Network Endpoint

Network Endpoint component is a front-end data processing unit connecting the HSS interface to the rest of MPR components. In additional to receive and transmit the data, it also applies the gain, A-law or µ-law conversion (in the narrowband mode) in both directions and high- pass filter (HPF) and echo cancellation in Rx direction (from the HSS to the DSP software).

The channels of Network Endpoint can be configured to narrowband or wideband in the initialization time.

In the narrowband mode, the application can specify A-law or µ-law conversion by setting the parameter XPARMID_NET_LAW. If this parameter is set to XPARM_NET_PASSTHRU, all the front-end processing mentioned above will be automatically bypassed. This is only used for debugging purposes and should not be set in normal applications. When XPARM_NET_PASSTHRU is set, the encoder and decoder should also be set to PASSTHRU codec. In this mode, 8-bit to 16-bit data conversion from HSS to linear is also bypassed and MPR components — such as tone detection and tone generation — are no longer meaningful. This parameter only applies to the narrowband mode.

Digital gain control can be applied to the audio signal in front of the Network Endpoint via the XPARMID_NET_GAIN_RX and XPARMID_NET_GAIN_TX parameters. This feature should be used only if the gain control is not available in the SLIC interface, because it takes extra processing time and may also affect the voice quality if they are not set properly. Gain control is bypassed when setting the gain control parameters to zero. A low-latency HSS channel bypass with gain control is available. For more details, see “Audio Stream Router” on page 27.

A high-pass filter is applied to the input audio data from HSS interface in order to remove the unwanted low frequency noise and safeguard the other algorithms from the harmful DC bias. The HPF has the 3-dB cut-off frequency, at 270 Hz in the narrowband mode, or 150 Hz in the wideband mode. The HPF cannot be disabled until the Network Endpoint is stopped.

Echo cancellation is the most significant function in this component. EC cancels the echo generated by the hybrid of local telephone interface and phone set so that the other party connected to the channel will not hear the echo. In other words, the beneficiary of EC is the remote party.

EC performance is mainly affected by two parameters: tail length and delay compensation (that is, XPARMID_NET_ECTAIL and XPARMID_NET_DELAYCOMP). Depending on the hardware circuits and telephone set, the tail length of 4 ~ 8ms is usually good enough if the telephone set is directly connected to the unit.

Since EC is very computation intensive, the longer tail length results in higher CPU occupancy. Changing the parameter of EC tail length requires that the Network Endpoint component be reset (by sending XMSG_RESET message). The CPU occupancy is about doubled if the channel is configured to the wideband mode. The tail length is limited to 32 ms for wideband mode.

EC can be made the most effective if the reference signal is properly aligned with the delayed echo signal. That is the purpose of adjusting the parameter of delay compensation. The value of the parameter should be determined according to the customer's specific hardware platform.

The user can use the XPARMID_NET_ECENABLE message to enable/disable EC. The message XPARMID_NET_ECFREEZE, used to disable adaptation on the EC algorithm, should only be used in debugging.

Network Endpoint resource also provides a complementary function of reporting hook state and detecting flash hook on behalf of the SLIC interface. The SLIC driver often notices the hook state changes through the interrupt. The SLIC's interrupt service routine can call the `xFlashHookDetect()` function which reports the hook state via the `XEVT_NET_HOOK_STATE` event. The event data gives the hook state. If an on-hook followed by an off-hook transition within the time specified by the `XPARMID_NET_FLASH_HK` parameter, a flash-hook event will be reported.

Another complementary function is timer service. The user applications can set the timer counter via the XPARMID_NET_TIMER parameter. This counter is decremented by 1 each 10 ms. A XEVT_NET_TIMER event is generated when the counter is decremented to 0.

The Network Endpoint component is started with the default setup values automatically after initialization. The application can still start or stop it using `XMSG_START` or `XMSG_STOP` message for debug and test purpose. Stopping the component is to stop EC, HPF, and the complementary functions, but the audio data stream still continues and the A-law or µ-law conversion still functions in the narrowband mode. In other words, stopping the Network Endpoint component does not affect data transfer between the HSS and IP interfaces.

4.2 Encoder

The primary function of this component is to encode and packetize the audio data from HSS and then send to the IP interface. The audio CODEC supports G.711, G.726, G.722, and G.729 on 10-ms frame size and G.723.1 on 30-ms frame size. Other features include Automatic Gain Control (AGC), Voice Activity Detector (VAD), and Multiple Frame per Packet (MFPP). In the following paragraphs, the possible affect of these features on voice quality or system performance is briefly discussed.

This component works in the wideband mode when using G.722.

There are two automatic gain control elements: AGC in the egress side and ALC (Automatic Level Control) in the ingress side. Only one of these should be turned on, depending on what gain control functions are implemented in the remote party.

In the completed audio path when two parties are connected, enabling both AGC on one side and ALC on the other side may cause unexpected interaction and degrade voice quality. Typical VoIP equipment employs ALC, thus it is recommended that AGC is turned off and ALC is turned on (this is the default).

The VAD algorithm can distinguish active speech signal from the silence (background noise). During the silence period the encoder only sends much smaller packets containing only the noise parameter at much lower rate. That helps to reduce network traffic.

Enabling VAD slightly impacts the voice quality.

Another effect of VAD is the change of average CPU occupancy. Enabling VAD in G.729 and G.723.1 will significantly reduce the average occupancy because the most complicated processing of G.729 encoder is eliminated during the silence and background-noise period. However, VAD increases the CPU occupancy, when enabled with G.711, because the VAD algorithm is much more complicated than just the G.711 coder.

VAD is not available in G.726 and G.722.

Packing more frames into a packet (i.e., MFPP) is another way to reduce network traffic. The application either specifies the number of frames per packet — in XMSG_CODER_START message when it starts the encoder — or modifies it — by setting the parameter XPARMID_ENC_MFPP at any time. Obviously, having MFPP increases the total latency and voice quality is more affected if the packet is lost. Typically, this trade-off of network traffic versus latency/voice quality is made depending on the target network and user preference.

The maximum numbers of frames that can be packed in one packet are listed below.

Coder Types	Maximum MFPP
G.711, G.722	6
G.723.1	8
G.726 40 Kbps	9
G.726 32 Kbps	12
G.726 24 Kbps	16
G.729, G.726 16 Kbps	24

The user can query or change the coder type via the `XPARMID_ENC_CTYPE` parameter.

Switching the coder type on the fly may cause a few packets discarded. The number of frames per packet may be reduced automatically during switching if it exceeds the maximum allowed by the new coder type. If the encoder is started by `XMSG_START` message without specifying MFPP and the coder type, the current parameter values take effect.

G.726 has four different rates. Each of them is treated as a different coder type. They use dynamic RTP payload types that are negotiated by the call stack during call setup. The application is responsible for informing the DSP software the payload types to be used in the current call by setting the payload type parameters. The payload type of the coder not used in the current call has no effect. The payload type parameters are:

- `XPARMID_ENC_G726_40_RTP_PLD`
- `XPARMID_ENC_G726_32_RTP_PLD`
- `XPARMID_ENC_G726_24_RTP_PLD`
- `XPARMID_ENC_G726_16_RTP_PLD`

Two packing formats are supported for G.726 of all the rates. One is described in RFC 3551 as commonly used for VoIP. Another is defined for ATM AAL in ITU-T I.366.2 Annex E. The XPARMID_ENC_G726_PACK parameter determines which format takes effect. Setting the parameter to XPARM_G726_PACK_LSB will choose RFC 3551 packing format or XPARM_G726_PACK_MSB for I.366.2 Annex E format.

The `XPARMID_ENC_EVT_PKT` message is used to setup the encoder to report bad packets. This is only intended for debugging since packet loss should not be monitored on an event basis.

Typically, the user application starts Encoder by `XMSG_CODER_START` or `XMSG_START` message when a call is setup and stops it when the call is torn down. The Encoder is the component that enables data flow from HSS to the IP side.

A PASSTHRU CODEC type is provided for debugging purposes in the narrowband mode, in conjunction with the pass through mode of the Network Endpoint component. When using PASSTHRU CODEC, no signal processing is done. The data in RTP G.711 packets are directly copied from HSS.

4.3 Decoder

The Decoder receives the encoded audio packets from the IP interface and converts them to the audio stream to the HSS interface. Similar to the encoder, the decoder supports G.711, G.729, G.723.1, G.722, and G.726 coder types and additional features like Comfort Noise Generator (CNG), ALC, Packet Loss Concealment (PLC), and Jitter Buffer.

CNG is the counterpart of VAD in the encoder. For G.729 and G.723 coders, CNG is built into the decoder algorithms and cannot be turned off. For G.711, disabling CNG will result in the pure silence between active speech periods if VAD is enabled in the remote party.

CNG is not available in G.726 and G.722.

The PLC algorithm uses the previous speech signal to repair the lost frames. But it cannot repair any big chunk of consecutive frames lost. Because of the complexity of PLC algorithm, it will increase the processor occupancy during packet loss when using G.711, G.726, and G.722 coders. But since they are relatively low computation coders, the resultant processor occupancy rates are still lower than that of G.729 and G.723.

The PLC algorithm is always enabled.

The Decoder automatically handles MFPP if a received packet contains multiple frames. The application starts Decoder when a call is setup, using `XMSG_CODER_START` message (`frmsPerPkt` field in the message is ignored for the Decoder). Currently, both the Encoder and Decoder support MFPP frame counts that are limited by internal buffer size.

The Jitter Buffer regulates the flow of data from the IP interface to the HSS interface. This is necessary since encoded audio packets from the IP interface are being transmitted on the IP network in real time using RTP protocol. This means packets can be delayed, out-of-order, duplicated, or lost without re-transmission. To perform this function, the Jitter Buffer delays incoming packets to allow delayed and out-of-order packets to arrive and be delivered to the HSS interface correctly. This delay is dynamically adjusted by the Jitter Buffer depending on IP network conditions.

The Jitter Buffer monitors network conditions by checking the timestamps in the incoming DSP software packets against the local clock. (The correct sequencing of audio packets is also done with the help of the timestamp.) The Jitter Buffer implements a proprietary delay profiling algorithm that provides better tracking and improves voice quality, compared with the algorithm specified by RFC 1889.

There is typically a trade-off of delay versus being able to recover more delayed packets in real data networks. The Jitter Buffer allows the user application to balance this by two parameters:

- `XPARMID_DEC_JB_MAXDLY` — Specifies the maximum desired jitter delay in ms (current range is 0 to 500 ms)
- `XPARMID_DEC_JB_PLR` — Specifies the allowable packet loss rate in 0.1% units

The jitter buffer automatically determines the jitter delay based on the network delay profile it keeps from the desired packet loss rate, subject to the limit of the maximum allowed jitter delay parameter. By setting the allowable packet loss rate judiciously, a balance between voice quality and latency can be achieved in real network conditions.

If a packet has not arrived after the allowable jitter delay, the packet is declared lost and the Decoder is instructed to perform packet loss concealment. The Jitter Buffer also handles VAD packets and MFPP packets appropriately.

The Jitter Buffer handles RFC 2833 tone packets independently, since they can be at a different rate than the CODEC frame rate and the timestamp are event based instead of frame based.

The Jitter Buffer is at the front-end of the ingress side. The user application uses the `xPacketReceive()` function to copy the encoded audio packets from the IP interface directly into the jitter buffer memory.

The user can query the coder type via the `XPARMID_DEC_CTYPE` parameter. During decoding processing, the coder type may be switched automatically according to the received RTP payload type or changed by the user's application.

To allow automatic coder switch, the user need to set the `XPARMID_DEC_AUTOSW` parameter in which each bit represents a coder type. For instance setting the parameter to (`XPARM_DEC_AUTOSW_G711MU` | `XPARM_DEC_AUTOSW_G711A`) means to allow the decoder to automatically switch between G.711 A-Law and µ-Law coder types. The received packets will be discarded if they do not match either of the two coder types.

Setting the parameter to `XPARM_DEC_AUTOSW_OFF` disables the auto-switch feature.

Setting the parameter to `XPARM_DEC_AUTOSW_ALL` enables Decoder to switch to all supported coder types.

The user can also change the coder via the `XPARMID_DEC_CTYPE` parameter at any time. But keep in mind that the coder type may switch anyway if auto-switch is enabled. When the decoder is started by `XMSG_START` message without specifying the coder type, the current parameter takes effect. Changing the coder type on the fly may cause a few packets lost.

The DSP software reports the changes of received RTP payload type through the event message (`XMSG_EVENT`). The event code is `XEVT_DEC_PACKET_CHNG`. The event data 1 gives the coder type associated with the changed payload type and the event data 2 is the received RTP payload type. From the event and the setting of `XPARMID_DEC_AUTOSW` parameter, the user application can determine if the coder type is switched automatically or not.

For example if the coder type reported by event matches any of the ones set in the XPARMID_DEC_AUTOSW parameter, the event also indicates the decoder has switched its coder type accordingly. The event report can be enabled or disabled by the XPARMID_DEC_EVT_PKTCHNG parameter.

G.726 has four different rates. Each of them is treated as a different coder type. They use dynamic RTP payload types that are negotiated by the call stack during call setup. The application is responsible for informing the DSP software the payload type to be used in the current call by setting the payload type parameters. The parameters are:

- XPARMID_DEC_G726_40_RTP_PLD
- XPARMID_DEC_G726_32_RTP_PLD
- XPARMID_DEC_G726_24_RTP_PLD
- XPARMID_DEC_G726_16_RTP_PLD

Two packing formats are supported for G.726 of all the rates. One is described in RFC 3551 as commonly used for VoIP. Another is defined for ATM AAL in ITU-T I.366.2 Annex E. The XPARMID_DEC_G726_PACK parameter determines which format takes effect. Setting the parameter to XPARM_G726_PACK_LSB will choose RFC 3551 packing format or XPARM_G726_PACK_MSB for I.366.2 Annex E format.

The XPARMID_DEC_EVT_PKT parameter is used to set up the decoder to report packet loss. This is only intended for debugging since packet loss should not be monitored on an event basis.

Typically, the user application starts the Decoder together with the Encoder when a call is setup and stops it when the call is torn down. The decoder is the component that enables data flow from the IP side to the HSS. A PASSTHRU CODEC type is provided for debugging purposes, in conjunction with the pass through mode of the Network Endpoint component in the narrowband mode. When using PASSTHRU CODEC, no signal processing is done. The data in RTP G711 packets are directly copied to HSS.

4.4 Tone Generator

The Tone Generator is capable to generate single- or dual-frequency tone and amplitude-modulated tone. It has a set of pre-defined tones. And user-defined tones can be added. Several tone segments can be combined as a single tone signal. This is very useful to generate some special call progress tones.

Internally, a tone is represented by a template that contains information like tone ID, frequencies, amplitude, and cadence. Current supported tones can have one or two frequencies (DTMF), each with its amplitude information. (Modulated tones are supported by specifying the carrier frequency/amplitude and modulating frequency/amplitude.) Tones, (especially call progress tones), can have a cadence, that is, an “on” duration, following by an “off” duration, and a repeat pattern.

The Tone Generator is a narrowband resource and cannot produce the frequency higher than 4,000 Hz.

All the tone templates, including DTMF and call progress tones, are pre-defined. Since call progress tones are country-specific, the application has to set the country code during initialization, so that Tone Generator can select the correct template table accordingly.

Overall tone volume can be changed by the XPARMIDTNGEN_VOL parameter.

The application can play tones by sending XMSG_TG_PLAY message with a list of tone IDs to be played sequentially. The definition of tone ID is compliant to RFC 2833 standard.

If tones are played while decoder has been started, the tone signal will overwrite or mix with the speech signal from the decoder according to the mode specified in the tone template. Most tones are of the overwrite-mode so that the speech is muted during the whole tone period. However, some tones have the cadence of a tone-on duration followed by a silent duration. For example, a call-progress tone, such as the call waiting notification tone, may require a short tone, followed by a long pause, and then the repetition of the tone-on/tone-off sequence. For these tones, the mix-mode is more appropriate, which allows the tone signal to be added to the speech so that the speech is not suppressed during the silence duration, or non-activated part of the tone.

If a continuous tone (e.g., call-progress tone) is played, the user application can stop it by playing another tone or stop it explicitly using XMSG_STOP message.

The Tone Generator can also generate FSK modem signals compliant to ITU-V.23 or Bellcore* 202 specifications, depending on user mode selection via the XPARMID_TNGEN_FSK_MOD parameter. This is implemented for caller ID generation. To implement caller ID functionality, a user application has to directly control the SLIC telephone interface and implement the caller ID transmit sequence, which are beyond the scope of the current DSP software.

FSK parameters such as baud rate, channel seizure bits (CS) length, mark bits length, and postmark bits length can be modified by the XPARMID_TNGEN_FSK_RATE, XPARMID_TNGEN_FSK_CS, XPARMID_TNGEN_FSK_MARK, and XPARMID_TNGEN_FSK_POSTMK parameters, respectively.

The Tone Generator also generates the corresponding tones when RFC 2833 packets are received, if RFC 2833 tone generation is enabled by the XPARMID_TNGEN_RFC2833 parameter. The RTP user application needs to classify the RFC-2833 packets based on the negotiated dynamic payload type, and encode the media field in the headers to indicate to the DSP software that these are RFC-2833 packets. RFC-2833 tones will override audio frames if both are present.

Although the Tone Generator has a set of pre-defined tones including the DTMF tones and the call progress tones of the United States, Japan, and China, the user applications can add more tone definitions through the xBuildToneTG() function in which a new tone is defined by a list of tone segments and associated tone ID.

Each segment is specified by a set of parameters including the signal types (single or dual frequency or amplitude-modulated tone), amplitudes or modulation rate, on/off durations and numbers of repetitions. A total of 64 tone segments can be added. Since a tone can contain multiple segments, the number of tones that can be added can be less than 64. The multiple segment tones are typically necessary in the country-specific call progress tone definitions.

Users can replace the pre-defined call progress tones with the newly added tones by specifying the same tone IDs.

The user-defined tones must be added during initialization time following xDspSysInit().

4.5 Tone Detector

The Tone Detector is also a narrowband resource and is able to detect single- or dual-frequency tones with the frequency range from 300 ~ 3,500 Hz, using an FFT analyzer. Besides the pre-defined tones, users can add new criterion tables during initialization to detect user-specified tone signals.

To reliably detect a dual tone, it is required that the frequencies of the dual-tone signal are separated by at least 200 Hz.

Internally all the tones to be detected (that is, DTMF tones and fax tones) are described by a list of templates that contain the criteria of frequencies, energy, SNR, durations, and so on.

To use any features provided by the Tone Detector, the user application needs to first start Tone Detector by sending `XMSG_START` message. The basic function of Tone Detector is to report tone events that are enabled by setting the parameter XPARMID_TD_RPT_EVENTS. Tone-on and/or tone-off events are reported according to the parameter. Tone events are reported via the `XMSG_EVENT` message in which the event data 1 field indicates tone ID and event data 2 field is the time stamp in 10-ms units.

Instead of being notified by tone events, the user application may want to receive a DTMF digit string, for example, a telephone number entered from the telephone set. For this purpose, the user application can use the `XMSG_TD_RCV` message and specify number of digits it expects and the termination conditions. Tone Detector will return the result via `XMSG_TD_RCV_CMPLT` message once the digits are collected or the termination conditions are met.

One scenario of using this feature is call setup. For example when the application detects the off-hook state of the telephone, it plays the dial tone and then starts to collect 10 digits of calling number entered by the telephone. It waits for 20 seconds for the first entering. After that, it stops collecting the entering of the digits if getting all the 10 digits as expected, or no entering in 5 seconds after any digits, or any special digits (star or pound) entered, or the total time of 25 seconds passed before getting 10 digits.

In this case, the application use the `XMSG_TD_RCV` message, specifying all the termination conditions mentioned above in the message correspondingly. The `XMSG_TD_RCV_CMPLT` message returns the collected digits and tells the reasons why collecting the digits is stopped. If the tone event report is enabled during receiving digits, the first digit entering is also reported as an event. The application can use that event to stop the dial tone in the above example. Then the tone event report is temporarily disabled for the rest of digits automatically.

The Tone Detector can also receive and decode FSK signals used in Caller ID specifications. Currently it works for Bellcore 202 or ITU-V.23 at a fixed, 1,200-bps baud rate. To start receiving FSK data, the application sends the `XMSG_TD_RCV_FSK` message and receives the `XMSG_TD_RCV_FSK_CMPLT` message with the decoded data once completed, or when the specified timeout has expired.

During receiving FSK, all other tone detection features are temporally suspended.

Another feature of the Tone Detector is tone clamping. The Tone Detector mutes the input audio stream from HSS during the period when a tone signal is detected. For VoIP applications, this feature is primarily used to implement out-band DTMF, because the tone signal is often distorted by speech coder like G.729. Since it takes about 30 ms to detect a tone, up to 30-ms tone signal may already leak out before it is clamped. To prevent tone leakage, the user application can enable the look-ahead buffer by setting the buffer size parameter `XPARMID_TD_TC_FRAMES` to 1, 2 or 3 (in 10-ms units). Remember that enabling the look-ahead buffer increases the latency accordingly.

If RFC 2833 is enabled (XPARMID_TD_RFC2833E_ENABLE), the Tone Detector will generate RFC-2833 payloads for transmission from the user RTP application, via the registered RTP transmit function (using xDspSysInit). The RTP payload type for the RFC-2833 packets is specified by the user via the XPARMID_TD_RFC2833E_PAYLOADTYPE message. The marker bit in the packet header is also set by the DSP software.

The rate for RFC-2833 packet generation can be set by the user application (XPARMID_TD_RFC2833E_UPDATERATE, typical rate is either 50 ms or coder frame rate). The number of beginning-of-tone (XPARMID_TD_RFC2833E_NUMBOE) and end-of-tone (XPARMID_TD_RFC2833E_NUMEOE) redundant packet transmission can also be set by the user application.

Normally, audio RTP packets are not transmitted during tones, but they can be enabled by turning off audio suppression (XPARMID_TD_RFC2833E_AUDIOSUPRESS).

Besides a set of built-in criteria to detect the DTMF and fax tones, users can add new criterion tables, using xBuildToneTD(), to detect user-specified tone signals. Currently users can add the new tone detection ability for single or dual frequency tones but not amplitude modulated (AM) tones. The user-specified tone will be reported via the XMSG_EVENT message along with the tone ID and time stamp.

The user cannot replace the pre-defined tone detection criteria. New tones are always added in addition.

4.6 Audio Player

The Audio Player component resource plays back the pre-recorded audio data to TDM and /or IP terminations. The Audio Player is currently designed to play cached voice prompts, that is, the audio data must be all pre-loaded into memory. The user application registers the audio data with the DSP software via xDspRegCachePrompt() and obtains the prompt handles. Each handle represents a piece of audio data stored in contiguous memory.

Currently up to 32 handles can be registered permanently. The audio data must be recorded in G.711 A-law/µ-law or G.729 format without VAD and loaded into the memory as raw data format without any extra embedded information such as header and time stamps, etc..

The demo source code included in this release gives the examples of using hard coded audio data and loading the audio data from wave format files.

During playback, the application can play any selected data segments by specifying the handle, offset and length. This segment information must be supplied with the XMSG_PLY_START message. Each message can carry up to 14 segments which can be played back in any given order once or repeatedly.

The number of player instances in the DSP software is configurable at initialization time. Each player instance has a dedicated location of the output audio stream where the encoded audio data is converted. To play back to an HSS or IP channel, the Network Endpoint resource or the Decoder resource has to listen to a player instance by connecting its input to the player. For details of audio stream routing, see “Audio Stream Router” on page 27.

If an application uses the player resource only for playing on-hold music, one player instance is enough for the purpose since all the channels can listen to the same player. Otherwise each channel may need a dedicated player instance.

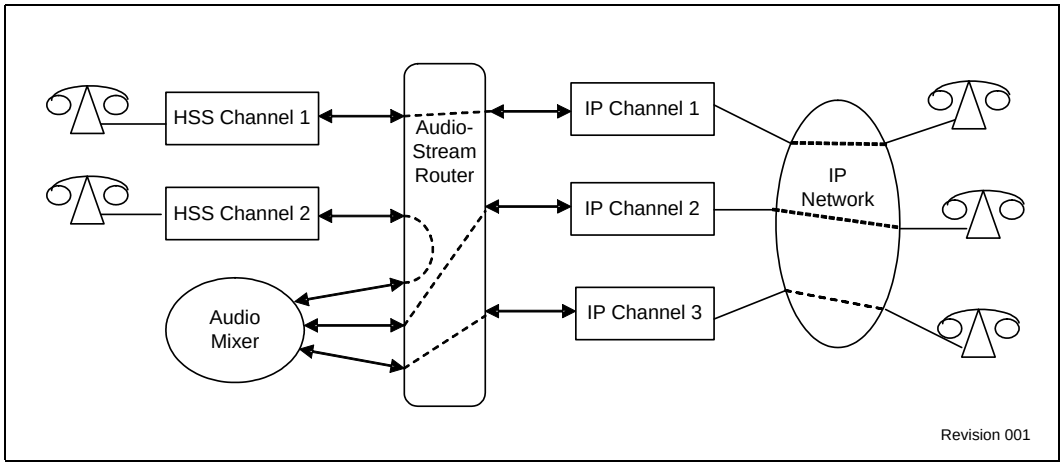
4.7 Audio Mixer

Audio Mixer mixes a number of audio streams to form an audio conference. The Mixer resource in theDSP software is primarily used for three-way call applications. It does not have the pre-processing functions that are found in the audio conference resources such as active talker selection, and volume balance. Therefore, mixing too many parties may results in voice quality problems like background noise built up, unbalanced volumes on different parties when the network condition is not good.

This release of the DSP software provides one Mixer instance. A Mixer can be configured to have 3 ~ 5 ports (or parties) during the system configuration time.

Figure 7 on page 26 shows how the audio streams are connected when a normal two-way and a three-way call are set up simultaneously. We can see during the three-way call there is no longer 1:1 association between HSS channels and IP channels and a mechanism of dynamically routing the audio streams is required. This will be discussed in the next section. Also we may need more IP channels than HSS channels if two parties of the three-way call come from IP side.

Figure 7. Audio Stream Connections in a Three-Way Call



The Mixer has multiple ports (pairs of input and output audio streams). Each port is to be connected to the resource (or party) that joins the call. The output of a port is the summation of all the inputs except for itself.

For example, consider three-party mixing:

- First party with input port L1 and output port T1.
  The output of first party on port T1 is sum of data of input ports L2 and L3.
- Second party with input port L2 and output port T2.
  The output of second party on port T2 is sum of data of input ports L1 and L3.
- The third party with input port L3 and output port T3.
  The output of third party on port T3 is sum of data of input ports L1 and L2.

The Mixer resource is started and stopped by the `XMSG_START` and `XMSG_STOP` messages. It has the parameters that are used to link its audio input and output to other resources.

Currently, the Mixer operates only in the narrowband mode. The wideband audio data is converted to narrowband if a wide channel is connected to the Mixer.

4.8 Audio Stream Router

The three-way call is an example that requires the audio streams be routed among the resources. Other examples are call transfer and IP tone detection.

To route the audio streams, we first break the DSP resources along the data path into a TDM termination and an IP termination which are connected by the router in between as shown in Figure 8 on page 28.

The TDM termination contains the Network Endpoint resource

The IP termination contains a set of resources (Decoder, Encoder, Tone Detector, and Tone Generator).

The TDM termination has a talk-port (T-Port) that supplies data to the router and a listen-port (L-Port) that receives the data from the router.

The IP termination has one T-Port shared by Decoder and Tone Generator and two L-Ports for Encoder and Tone Detector separately.

In general, a resource that generates PCM audio data has a T-Port as its output and a resource that receives the audio has an L-Port as its input. For example, an Audio Player instance has only one T-Port and a Mixer has multiple pairs of T-Ports and L-Ports.

The Router applies sampling rate conversion (SRC) automatically if the resources being connected are in different modes (wideband or narrowband).

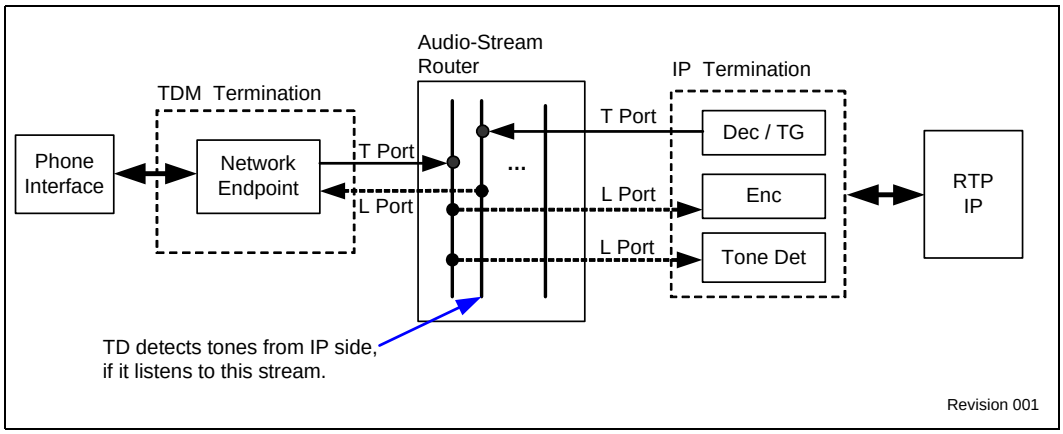
The DSP software implements a distributed switch method to route the audio streams. The Audio Stream Router is not a control entity but a set of streams that can link the T-Ports and L-Ports.

All the T-Ports of the resources are assigned the dedicated streams permanently.

Routing is done by enabling an L-Port of a resource to listen to any streams by setting a parameter to the resource. In this way any T-Ports can be linked to any L-Ports.

Figure 8 shows a full-duplex connection between a TDM termination and an IP termination. In this figure, if the L-Port of the Tone Detector listens to the stream of the T-Port of the same IP termination instead of the one of TDM termination, then it will detect tones coming from the remote IP side.

Figure 8. Terminations and Router



Each stream is specified by a unique ID number from 0. A null stream is given the ID as (-1). Any L-Ports listen to the null stream receive silence.

To make a connection between two resources, the user has to know what stream IDs are assigned to the T-Ports of the resources. Such information is available by calling `xDspGetResConfig()`. The function returns the base stream IDs for the T-Port for each type of terminations and resources (the TDM and IP terminations, Player and Mixer).

For example, the base stream ID of the TDM termination means the stream ID assigned to the T-Port of the first TDM termination channel. The T-Port stream of *n*th channel (n=1,2,..) is calculated as (base stream + n - 1). The base stream of the Mixer means the output stream of the Mixer's first port. The Mixer has 3~5 L-Ports that it mixes and it has the same numbers of T-Ports where the outputs of the mixes are transmitted.

Having the stream ID information for the T-Ports, the user can have a resource listen to a particular T-Port by setting the L-Port stream parameter of the resource. For example, to detect the tone from IP side in the channel 2 of the IP termination, the user first obtains the base stream ID of the IP termination (suppose it is 4), Then the T-Port stream ID of IP termination channel 2 is 5 (4 + 2 - 1). The user needs to set the XPARMID_TD_LP_STREAM parameter of the Tone Detector to 5.

Network Endpoint and Encoder have their L-Port stream parameters too.

The `XPARMID_MIX_LP_STREAM` is such parameter ID of the first port of the mixer. For the rest of the ports, parameter IDs increases by 1 sequentially.

Examples of high-level message interfaces that link the terminations and the Mixer are also provided using the Message Agent approach.

In some applications, the user may want to link two TDM terminations without IP involved (also called TDM switch or TDM bypass).

There are two modes for such connection. In the normal mode — when the `XPARMID_NET_HSS_BYPASS` parameter in Network Endpoint resource is set to `XPARM_OFF(0)` — the echo cancellation and front end gain control are applied to the audio path. This achieves a latency of approximately 25 ms. This is a bypass at the Intel XScale® Core level.

In the short bypass mode when the parameter is set to `XPARM_ON(1)`, the connection is made within the NPE between the corresponding time slots, therefore the latency is reduced significantly to approximately less than 2 ms. In this mode, only the gain control remains in effect.

The short bypass can only be enabled if both TDM terminations to be linked are in narrowband mode or the audio data will be corrupted.

4.9 T.38 Fax

The T.38 Component serves as the real-time fax gateway between G3E fax machines and IP network. Unlike the fax bypass mode in which the modulated fax data are directly packed in G.711 format and transmitted over RTP packets, the T.38 component transfers the demodulated T.30 commands and fax image data over UDP or TCP packets.

As depicted in Figure 2 on page 10, the T.38 component contains three modules:

- A fax modem that establishes the T.30 session between the fax gateway and the local fax machine
- T.38 CODEC that encapsulates the demodulated T.30 commands and HDLC data together with redundancy or forward error correction, into fax data packets suitable for transmission over UDP or TCP protocols
- Packet Loss Recovery (PLR) that recovers lost packets from the redundancy or forward error correction on the receive side

The T.38 component is implemented as a separated entity from the voice resources (the Encoder, Decoder, Tone Detector, and Generator). It accepts the common control messages such as `XMSG_START`, `XMSG_STOP`, and `XMSG_SETPARM`.

The T.38 component is mutually exclusive with voice resource components within the same channel during the run-time. It is the user applications' responsibility to stop the voice resources and start the T.38 component when switching over from voice mode to T.38 fax mode.

The included DSP codelet source code provides examples of how this can be accomplished in the VoIP gateway demonstration.

The DSP software uses the same packet format to exchange voice and T.38 packets with the user applications. The media type field in the packet header indicates the packet types. In the TDM side, the fax modem uses the same PCM stream IDs assigned to the Encoder and Decoder with the same instance number to receive or generate the modulated fax data.

There are different modes that T.38 can operate in: in UDP or TCP mode, specified by the parameter `XPARMID_T38_TRANSPORT`, with packet redundancy or FEC (Forward Error Correction), specified by the parameter `XPARMID_T38_FEC`.

For UDP mode, the T.38 packets are transmitted to the IP in UDP packets. Packet loss in the network is recovered by either FEC or packet redundancy.

For TCP mode, the fax payload is transmitted via TCP/IP protocol. Packet loss in the network is recovered by retransmission via the TCP/IP protocol.

Encapsulation of the UDP or TCP packets is the responsibility of the user application. In UDP mode, the DSP software emits the formatted UDPTL packet; in TCP mode, it emits the raw fax payload. The media type field in the DSP packet header identifies the type of packet being transmitted or received.

The `XPARMID_T38_RATE_NEG` parameter determines whether the rate negotiation is performed locally or remotely. Rate negotiation is typically done remotely for UDP mode, since the network conditions affect rate selection. Rate negotiation is typically done locally for TCP mode. In this case, `XPARAID_T38_TCF_THRSHLD` determines the error level threshold used to locally determine the rate.

In UDP mode, T.38 specifies either packet redundancy or FEC for error recovery. For packet redundancy, the `XPARMID_T38_REDUNDANCY` parameter specifies the level of redundancy. This is only an indication of the overall level of redundancy. The actual redundancy in the payload is also determined by the type of fax payload (for example, signaling or image data).

The DSP software also optionally supports some variations on the T.38 protocol for Ellipsis* and China Telecom* versions.

4.10 Message Agent

The DSP software exposes the individual media-processing resources and provides a basic set of message interface to user applications. This allows the maximal flexibility, but may not be convenient to the application development.

For example, the user application may have a state machine driven by the asynchronous events from the call stack and user inputs of the telephone set. For each event, the application has to send several control messages to the resource components and handle the replies. The large number of messages and their replies make the state machine more complicated. Ideally, the user may want to have just one comprehensive message for each event which can accomplish all the control over all the resource components involved and to receive only one reply message for the results.

The Message Agent can be viewed as a macro or scripting facility that allows multiple basic messages to be executed by one user message command. By eliminating multiple messages being passed between the DSP software and user application, the associated context swaps are removed and operating efficiency gained. By providing a base of helpful pre-defined user messages, which can be modified and expanded, the integration between user application and the DSP software can be expedited.

If users are going to replace their existing DSP solution with the DSP software, they may have to modify their applications significantly because of the differences in the interfaces, or they may implement a translation layer to convert the interface. To build such a layer on top of the DSP software may introduce extra overhead and inefficiency. With the Message Agent, the user can embed such translation layer inside the DSP software much more easily and efficiently because the message traffic is greatly reduced.

The Message Agent is a special resource component which does not have any media processing functions. To support the user-defined, high-level messages, the user needs to supply a message decoder function registered with the Message Agent. The function decomposes the user message into a series of original control messages. The Message Agent will directly execute the control to resources based on the decoded message sequence. During the procedure, the responses from the resources are redirected to another user-supplied message encoder function which composes the responses into one user-defined reply message sent back to the user application by the Message

Agent. The only responses which are directly the results of the control messages such as `XMSG_ACK` and `XMSG_ERROR` are redirected. The messages that are the results of media data processing like `XMSG_EVENT` and `XMSG_TG_PLAY_CMPLT` are still sent to the applications as usual.

The Message Agent is enabled if a message decoder function is registered during the initialization via the `xDspSysInit()` function. The message encoder function is optional. If not registered, the replies from the resources are always sent to the application as usual.

As examples, this release includes a set of high-level messages and the source code of message decoder and encoder functions. User can further extend and modify that message interface.

Programming Guide 5

This section discusses the rules and guidelines that should be followed when building user applications on top of the Intel® IXP400 DSP Software.

5.1 Initialization

As the DSP software is a standalone module or a layer of media processing, it must be configured and initialized properly before the application can interact with it. To configure the software, the user must provide configuration information as defined in the `XDSPSysConfig_t` data structure. That includes:

- The Signal formats and time slot assignment on the HSS's TDM bus as defined by the data structures `IxHssAccConfigParams` and `IxHssAccTdmSlotUsage`.

  Each instance of the Network Endpoint resource must be linked to one or four time slots depending on the mode. If more time slots are activated, the data from the extra time slots are ignored and the data to those time slots are undetermined.

  The link between the effective time slots and the instances of the Network Endpoint is specified by the `XDSPChanTdmSlots_t` data structure. If not given, all the instance of Network Endpoint will be configured to the narrowband mode and the first N time slots will be linked to the total N instance of Network Endpoint component sequentially. (For more details, see Section 8.1 in API Reference Manual.)

  In current release, the number of active time slots must be at least eight if the low-latency TDM switching feature is required. (The latency of HSS NPE will be minimized if eight or more time slots are enabled).

- The number of instances of media processing resource components. The maximum number of instances is four (except for Mixer which has only one instance). The default number of four will be used if an invalid number is given.
- Country code which determines the call-progress tone definitions and some the default FSK parameters.
- The base priority for the internal real-time data-processing and control tasks.
- The call back functions.

With user-supplied configuration information, the initialization follows these steps:

1. Download HSS NPE code and initialize HSS dependents.

   (For more information, see HSS Interface document and demo source code.)
2. Add user-specified tone detection criterion tables to Tone Detector using `xBuildToneTD()`.
3. Call `xDspSysInit()` with the configuration information as described above.
4. An assertion occurs if fatal errors happen (for example, if the memory is exhausted).
5. Add user-defined tone definitions to Tone Generator using `xBuildToneTG()`.
6. Use `xDspGetResConfig()` to retrieve the base stream information assigned to the different resource components.

Such information is required when routing the audio streams between the resources. Also the function returns the actual resource configuration that can be different from what a user may have incorrectly specified.

5.2 Programming Model

A VoIP gateway application may contain several modules such as user interface, IP call stacks, and the DSP software. The key functionality of the gateway application is to handle the call progress procedure: establishing calls and connecting the audio data path between two remote and local parties, then dropping the calls and disconnecting the data path accordingly. From control point of view, this procedure can be characterized as the interactions among the DSP software, IP call stack, and SLIC driver through asynchronous messages and commands. Such control logic is best implemented by a message-driven state machine model. The DSP software's control interface is suitably designed to support this programming model.

To use the state machine approach, it is recommended that the user application spawn a dedicated task to handle the call-progress procedures. The DSP software allows the user to use the DSP software release output message queue via the `xMsgWrite()` function.

The user can use this function to send external messages (such as SLIC driver events or IP call stack messages) back to the user application to allow all message inputs to be consolidated. (In Linux*, this can be done in the client driver module). Then the user control task is pending on the message queue, using `xMsgReceive()`, to handle all the call progress-related messages from all these modules.

Figure 9 shows a general approach of such state machine model. In this programming model, a call progress scenario is represented by a sequence of states. Each state is characterized by:

The actions it takes

The messages it expects

The next state it goes to

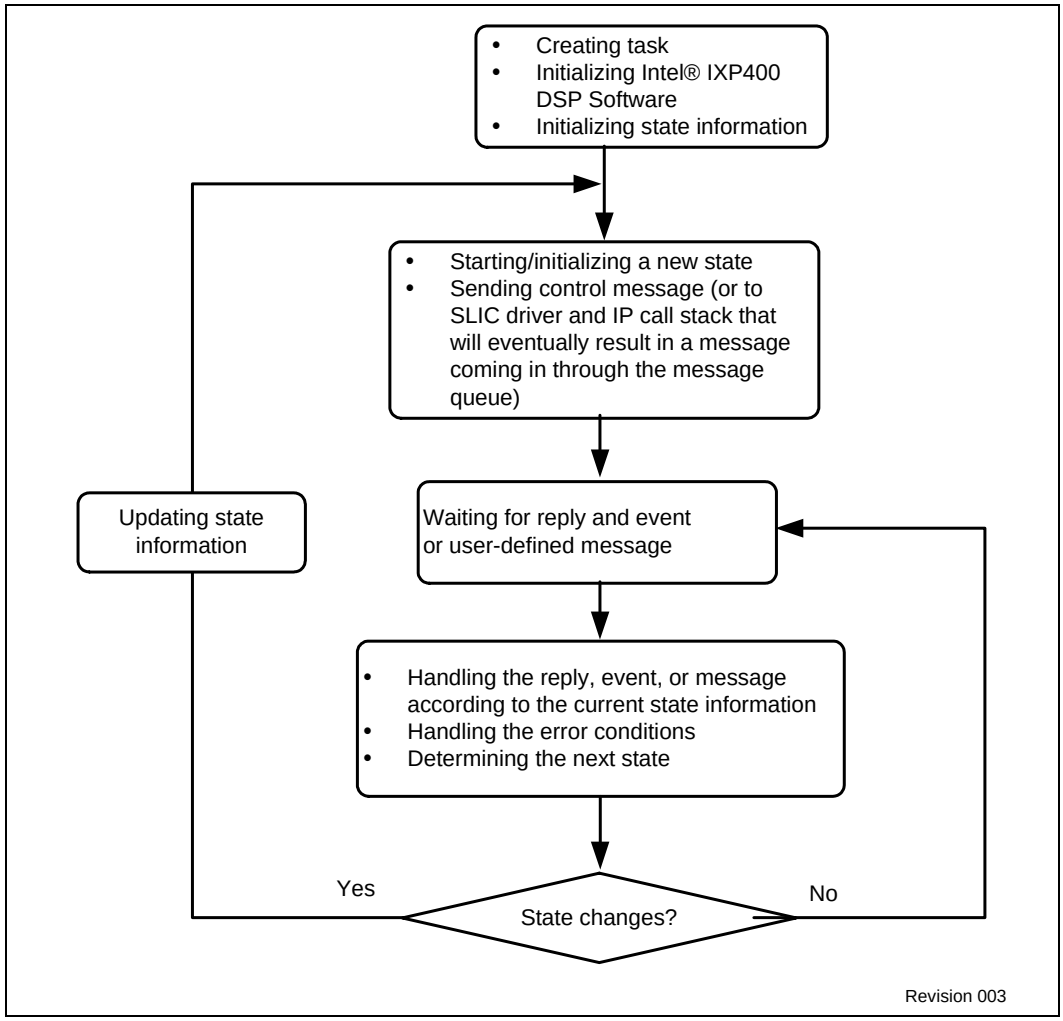
For example, the scenario of accepting a remote call can be represented by the following states:

- Idle State — Waiting for call-setup message from IP call stack.
- Ring State — Ringing the local telephone set and waiting for an off-hook event.
- Channel Setup State — Sending control messages to the DSP software to start encoder, decoder, and tone-detector resources and waiting for the acknowledges.
- Connected State — Acknowledging IP call stack that a local channel has been set up. Waiting for disconnect message from the call stack or on-hook event from the local telephone set.
- Teardown State — Sending control messages to the DSP software to stop the resources and waiting for acknowledgements. Acknowledging IP call stack that the channel has been teardown. Going back to Idle State.

The actual state machine will be more complicated when taking all the possible error conditions into account. For instance, timeout message must be handled in Ring State if the call is not answered.

The major advantage of such programming model is the high efficiency and good performance. In Linux, it also helps the DSP software to maintain its real-time behavior. The Gateway Demo included in this release is a good example of using this programming model.

Figure 9. General State-Machine Approach for Client Applications



OS-Specific Issues 6

Because of the substantial differences between VxWorks* and Linux, the DSP software's internal real-time environment and interface are implemented differently. The exposed APIs look, however, are identical.

Users need to understand some OS-specific issues in order to design the overall software appropriately.

6.1 VxWorks*

The application development in VxWorks is quite straightforward because of the excellent real-time properties and development tools provided by the OS. There are two aspects that make the implementation in VxWorks simpler.

- It provides the preemptive multitasking environment with enormous supporting functionalities
- All the software modules reside under the same memory address space

In the current release for VxWorks, two internal tasks are spawned and two sets of task properties are reserved for future use, as listed below. (The higher the number, the lower the priority is.) The priorities of the real-time tasks are assigned according to rate-monotonic-scheduling (RMS) — that is, the higher-frequency periodic task gets higher priority.

Task Name	Priority	Description
DspCtrlTsk	40	Control task. Pending on in-bound message queue. Triggered by incoming control messages.
DspRtTsk30	39	Real-time task. Wake up in every 30 ms synchronously with PCM data. Executes G.723, fax modem and T.38 CODEC algorithms.
DspRtTsk10	38	Real-time task. Wakes up in every 10 ms synchronously with PCM data. Execute all DSP algorithms supported in the current release.

The user applications can change the default priority settings during the initialization by specifying the base priority level — the priority of the control task. The real-time tasks have higher priorities than the base priority, increasing by one level sequentially. The users have to assign the priorities to their application tasks properly in order to coexist with the DSP software. In summary, the rules for the user applications are:

- User control tasks that are not involved in data- and packet-processing should not have a higher priority than the internal control task.
- User time-critical tasks may have a higher priority only if their execution is predictable and does not significantly affects the internal real time task. (For example, the task does not preempt the real-time tasks long enough to prevent the real-time task from meeting its deadline, every 10 ms.)
- The applications must not send a burst of control messages without waiting for the replies, or the message queues may overflow.

6.2 Linux*

Linux will be the choice for the DSP software if the cost of the target products is the major consideration. This OS will require some extra development effort and caution because:

- Linux is not a real-time OS and does not support priority-based, preemptive multitasking.
- User-mode applications cannot directly access the interfaces which reside in kernel-mode.

  A shim layer of driver software must be developed to allow the user application to communicate with the DSP software.

The DSP software in Linux is fully in kernel mode. The software creates the following kernel-mode threads:

Thread Name	Priority	Description
DspCtrlTsk	kernel	Control thread. Pending on in-bound message queue. Triggered by incoming control messages.
DspRtTsk30	kernel	Real- time task, Wakes up every 30 ms synchronously with PCM data. Executes G.723, fax modem and T.38 CODEC algorithms.
DspRtTsk10	kernel	Real time task. Wakes up every 10 ms synchronously with PCM data. Execute all the DSP algorithms supported in the current release.

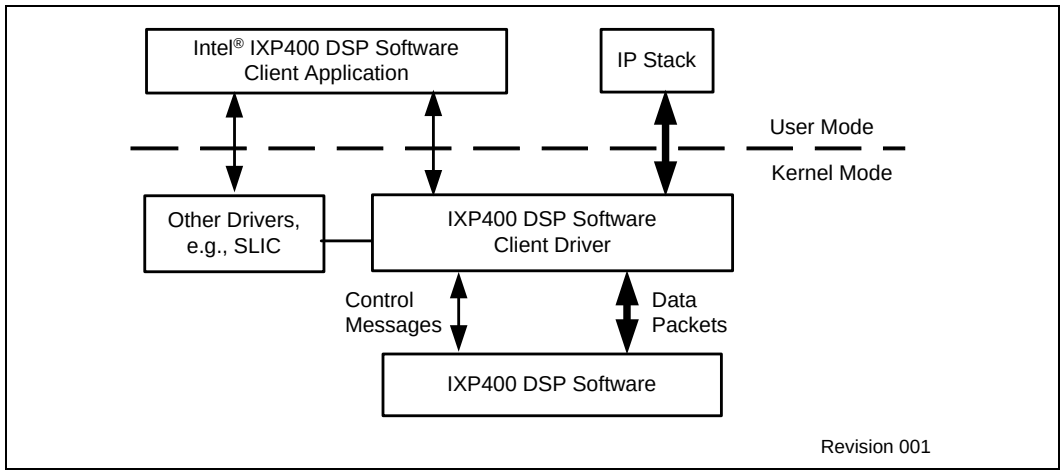
The two threads have the same priorities and do not preempt each other. To enforce real-time behavior, it is important that `DspCtrlTsk` never takes too much time in any 10-ms period. Although the DSP software is designed to avoid the burst execution in the control task, it can still be affected by the user applications.

For the performance and reliability reasons, it is suggested the user applications that are non-time-critical such as call-control and call-progress modules be implemented in Linux user-mode. It is the user's responsibility to develop the client driver module as shown in Figure 10.

The demo code in the DSP software release provides an example of the client driver module.

As the middleware, the primary responsibility of the driver module is to act as a transport layer between the DSP software's control interface and the user application and between the packet interface and the IP stack. The secondary responsibility is to perform the module initialization, which can be done as part of driver module initialization function. Additionally, the driver may also consolidate the messages and events from SLIC and other related modules into the same format and through a single queue to the user applications.

Figure 10. Intel® IXP400 DSP Software Client Driver in Linux*



The driver can be implemented as an active or passive transport layer. In active mode, the driver spawns a dedicated kernel thread pending on the out-bound queue and automatically pumps the messages to applications once there is a message in the queue. In passive mode, it retrieves the message from the queue once the user application requests it.

As discussed in "Programming Guide" on page 33, the programming model for user application is still recommended. The applications should not send a burst of control messages without waiting for the replies, or the real-time behavior of the DSP software may be affected.

If the user application has to create kernel threads for time-critical data processing, the execution of the threads must be predictable and not impact the internal real time thread. As a guideline, the total execution time of these other threads should not exceed 1 ms in any 10-ms period.

User-Defined Messages 7

The DSP software provides a facility for users to define custom messages, based on a combination of basic messages, to enable a simpler and more efficient interface.

7.1 Overview

To enable the user control message facility using the Message Agent, the user application needs to register a decoder function and an encoder function with the DSP software, via the function `xDspSysInit()`. The decoder function is called by the DSP software to decode all user control messages. The encoder function is called to handle all the replies to the decoder function, eventually encoding a reply message to the user message.

User control messages have the same format as the basic control messages, which contain a message header defined as:

```
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;
```

The resource field in the header should specify `XMPR_MA`, which directs it to the Message Agent resource.

The instance field must be always 1. Since instance field is always 1 for the messages sent to and received from the Message Agent, the user has to use the `transactionId` field to track the messages associated with the channels.

The type field in the header specifies type of message. User control messages should start with the value `XMSG_USRMSG_TYPE_BEGIN` - values less than this constant represents the basic control messages.

User-defined messages are delivered in the same way as theDSP software control messages — using the same message queues for input and output, respectively.

The Message Agent calls the registered decoder function when the type is beyond its internal range. The user-supplied decoder function is of the format:

```
typedef int (*XMsgAgentDec_t)(XMsgRef_t pUsrMsg,
                 XMsgRef_t pNativeMsg,  int sequenceNo);
```

The first parameter of the decoder function is the input message pointer, referencing the control message to be decoded.

The second parameter is the output message pointer, referencing the output of the decoder function.

The sequenceNo field starts at 1 for the first decoder function call, and is incremented each time the decoder function is called.

The return value of the decoder function indicates whether the decoder function is complete with its message sequencing (returns 0); or whether the decoder function should continue to be called (returns non-zero). The return value can simply be the number of messages left to sequence. If there is an error in the decoder process, the return value is set to negative. The return value — together with the sequence number — are used to drive the decoder function until the entire message sequence required is complete.

One useful feature of the Message Agent is the ability to recursively call other user control messages (maximum level of recursion is 4). This allows more complex functions to be built compactly.

The user encoder function is of the format:

```
typedef void (*XMsgAgentEnc_t)(XMsgRef_t pUsrReply, XMsgRef_t pNativeReply,
                    int sequenceNo, UINT8 usrMsgType);
```

The first parameter of the encoder function is the output message pointer, referencing the output of the encoder function.

The second parameter is the input message pointer, referencing the reply message from the resource component involved.

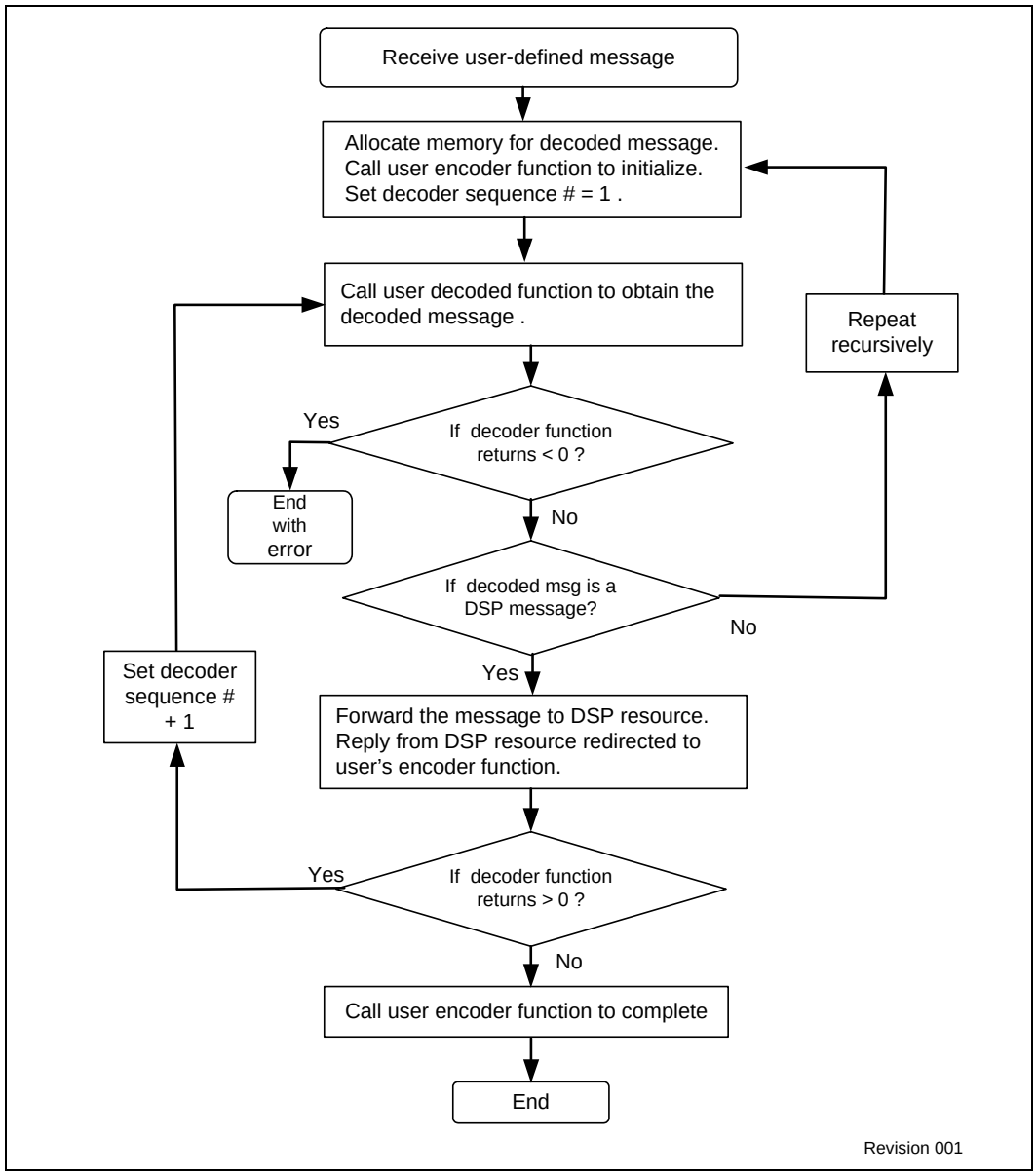
The Message Agent first calls the encoder function once with sequenceNo set to `XMSG_MA_ENCODING_INIT (0)` before receive the replies. Then the sequenceNo field is incremented each time a reply is received. The Message Agent in the DSP softwaresets this field to `XMSG_MA_ENCODING_CMPLT (-1)` when the decoding process is complete.

The `usrMsgType` field informs the encoder of the user message ID, such that specific encoder functions may be called accordingly.

The replies to the decoded messages are re-directed to the encoder function, which can record the number of replies and any errors that may occur. A final reply message will be encoded and sent back to the user application when the message decoding process is complete.

Figure 11 depicts how the Message Agent processes user-defined control messages.

Figure 11. Decoding User-Defined Messages in the Message Agent



7.2 Pre-Defined User Messages

This section describes the user messages that have already been implemented as examples. They can be further extended or modified by the users. These messages form a higher-level control interface for the application scenarios like call setup, call transfer and three-way call. The control entities of this interface are the terminations which can be a TDM or IP terminations or a port of the mixer. The termination is specified by its type and channel defined as:

```
typedef struct{
    UINT8 type;
    UINT8 channel;
} __attribute__ ((packed)) IxDspCodeletTerm;
```

where channel is the channel number, and type can be

```
typedef enum{
    IX_DSP_CODELET_TERM_NULL = 0,    /* null termination, to link to null
                                        means to disconnect the L-Port */
    IX_DSP_CODELET_TERM_TDM,         /* TDM termination contains one DSP
                                        dspResource - Network Endpoint which
                                        has a T-Port and a L-Port */
    IX_DSP_CODELET_TERM_IP,          /* IP termination contains DEC,ENC,
                                        TG and TD resources. It has one
                                        T-Port shared by DEC and TG and
                                        2 L-Ports for ENC and TD. But in
                                        This API, the 2 L-Ports always
                                        listen to the same talker */
    IX_DSP_CODELET_TERM_MIXER_PORT, /* Mixer termination has multiple
                                        T-Ports and L-Ports */
    IX_DSP_CODELET_TERM_EOL          /* End of List */
} IxDspCodeletTermType;
```

The following message types are defined and the corresponding message decoder and encoder functions are implemented:

```
typedef enum{
    /*------ messages send to Message Agent -------*/
    IX_DSP_CODELET_MSG_LINK = IX_DSP_CODELET_MSG_TYPE_BEGIN,
    IX_DSP_CODELET_MSG_LINK_BREAK,
    IX_DSP_CODELET_MSG_LINK_SWITCH,
    IX_DSP_CODELET_MSG_START_IP,
    IX_DSP_CODELET_MSG_STOP_IP,
    IX_DSP_CODELET_MSG_SETUP_CALL,
    IX_DSP_CODELET_MSG_SET_CALL_PARMS,
    IX_DSP_CODELET_MSG_SETUP_CALLWPARMS,
    IX_DSP_CODELET_MSG_SWITCH_CALL,
    IX_DSP_CODELET_MSG_CREATE_3WCALL,
    IX_DSP_CODELET_MSG_EXIT_3WCALL,
    IX_DSP_CODELET_MSG_TEARDOWN_3WCALL,
    IX_DSP_CODELET_MSG_BACKTO_2WCALL,
    IX_DSP_CODELET_MSG_SET_CLEAR_CHAN,
    IX_DSP_CODELET_MSG_T38_SWITCH,
    IX_DSP_CODELET_MSG_SET_PARMS,
    IX_DSP_CODELET_MSG_END_OF_OUTMSG,
    /*------messages received from Message Agent------*/
    IX_DSP_CODELET_MSG_ACK,
    IX_DSP_CODELET_MSG_LINK_ACK,
    IX_DSP_CODELET_MSG_SETUP_ACK,
    IX_DSP_CODELET_MSG_3W_ACK,
    IX_DSP_CODELET_MSG_STOP_ACK,
    IX_DSP_CODELET_MSG_T38_ACK,
    IX_DSP_CODELET_MSG_END_OF_LIST
} IxDspCodeletMsgType;
```

7.2.1 Link Message

Type: IX_DSP_CODELET_MSG_LINK

Direction: Inbound

Description: Connects two specified terminations. Since terminations involve multiple resources, this involves multiple basic control messages.

Format:

```
typedef struct{
    XMsgHdr_t           header;
    IxDspCodeletTerm    term1;
    IxDspCodeletTerm    term2;
} IxDspCodeletMsgLink;
```

Macro:

```
#define IX_DSP_CODELET_MAKE_MSGHDR_LINK(pMsg, trans) \
    {\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_LINK, \
            sizeof(IxDspCodeletMsgLink) \
        )\
    }
```

Response:

General Acknowledgement message (IX_DSP_CODELET_MSG_Link_ACK)

7.2.2 Link Break Message

Type: IX_DSP_CODELET_MSG_LINK_BREAK

Direction: Inbound

Description: Disconnect two terminations. This connects each termination to null, using the `IX_DSP_CODELET_MSG_LINK` user message.

Format:

```
typedef struct{
    XMsgHdr_t           header;
    IxDspCodeletTerm    term1;
    IxDspCodeletTerm    term2;
} IxDspCodeletMsgLinkBreak;
```

Macro:

```
#define IX_DSP_CODELET_MAKE_MSGHDR_LINK_BREAK(pMsg, trans) \
    {\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_LINK_BREAK, \
            sizeof(IxDspCodeletMsgLinkBreak) \
        )\
    }
```

Response:

General Acknowledgement message (IX_DSP_CODELET_MSG_LINK_ACK)

7.2.3 Link Switch Message

Type: IX_DSP_CODELET_MSG_LINK_SWITCH

Direction: Inbound

Description: Disconnects the termination from one termination and connects to another. This connects the `term` termination to the `switchTo` termination, and connects the `switchFrom` termination to null. Again, this uses the `IX_DSP_CODELET_MSG_LINK` user message.

Format:

```
typedef struct{
    XMsgHdr_t           header;
    IxDspCodeletTerm    term;
    IxDspCodeletTerm    switchFrom;
    IxDspCodeletTerm    switchTo;
} IxDspCodeletMsgLinkSwitch;
```

Macros:

```
#define IX_DSP_CODELET_MAKE_MSGHDR_LINK_SWITCH(pMsg, trans) \
    {\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_LINK_SWITCH, \
            sizeof(IxDspCodeletMsgLinkSwitch) \
        )\
    }
```

Response:

General Acknowledgement message (IX_DSP_CODELET_MSG_LINK_ACK)

7.2.4 Start IP Message

Type: IX_DSP_CODELET_MSG_START_IP

Direction: Inbound

Description: Starts an IP termination. This involves the basic messages to start the Encoder, Decoder, and Tone Detector, respectively, and to stop the Tone Generator.

Format:

```
typedef struct{
    XMsgHdr_t   header;
    UINT8       channel;
} IxDspCodeletMsgStartIP;
```

Macros:

```
#define IX_DSP_CODELET_MAKE_MSG_START_IP(pMsg, trans, chanIP) \
    {\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_START_IP, \
```

```
            sizeof(IxDspCodeletMsgStartIP) \
        )\
        ((IxDspCodeletMsgStartIP *)(pMsg))->channel = (chanIP);\
    }
```

Response:

General Acknowledgement message (IX_DSP_CODELET_MSG_SETUP_ACK)

7.2.5 Stop IP Message

Type: IX_DSP_CODELET_MSG_STOP_IP

Direction: Inbound

Description: Stops an IP termination. This involves the messages to stop the Encoder, Decoder, Tone Detector, and Tone Generator, respectively.

Format:

```
typedef struct{
    XMsgHdr_t   header;
    UINT8       channel;
} IxDspCodeletMsgStopIP;
```

Macros:

```
#define IX_DSP_CODELET_MAKE_MSG_STOP_IP(pMsg, trans, chanIP) \
    {\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_STOP_IP, \
            sizeof(IxDspCodeletMsgStopIP) \
        )\
        ((IxDspCodeletMsgStopIP *)(pMsg))->channel = (chanIP);\
    }
```

Response:

Stop Acknowledgement message (IX_DSP_CODELET_MSG_STOP_ACK)

7.2.6 Set Up Call Message

Type: IX_DSP_CODELET_MSG_SETUP_CALL

Direction: Inbound

Description: Sets up a call. This uses two user messages, `IX_DSP_CODELET_MSG_LINK` to connect an HSS termination to an IP termination, and `IX_DSP_CODELET_MSG_START_IP` to start the IP termination.

Format:

```
typedef struct{
```

```
    XMsgHdr_t   header;
    UINT8       channelIP;
    UINT8       channelTDM;
} IxDspCodeletMsgSetupCall;
```

Macros:

```
#define IX_DSP_CODELET_MAKE_MSG_SETUP_CALL(pMsg, trans, chanIP,
chanTDM) \
    {\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_SETUP_CALL, \
            sizeof(IxDspCodeletMsgSetupCall) \
        )\
((IxDspCodeletMsgSetupCall *)(pMsg))->channelIP = (chanIP);\
((IxDspCodeletMsgSetupCall *)(pMsg))->channelTDM = (chanTDM);\
    }
```

Response:

General acknowledgement message (`IX_DSP_CODELET_MSG_SETUP_ACK`)

7.2.7 Set Call Parameters Message

Type: IX_DSP_CODELET_MSG_SET_CALL_PARMS

Direction: Inbound

Description: Sets parameters of a call. These parameters are likely affected by the results of negotiation between the call stacks and may change call by call. The message involves four basic messages to set the parameters for the Encoder, Decoder, Tone Detector, and Tone Generator of an IP termination.

Format:

```
typedef struct{
    XMsgHdr_t               header;
    IxDspCodeletCallParms   parms;
    UINT8                   channelIP;
}IxDspCodeletSetCallParms;
```

where `IxDspCodeletCallParms` is defined as:

```
typedef struct{
    UINT16  decAutoSwitch;
    UINT8   decType;
    UINT8   encType;
    UINT8   frmsPerPkt;
    UINT8   vad;
    UINT8   rfc2833;
    UINT8   rfc2833pyldType;
```

```
    UINT8   toneClamp;
} IxDspCodeletCallParms;
```

Macros:

```
#define IX_DSP_CODELET_MAKE_MSGHDR_SET_CALL_PARMS(pMsg, trans) \
    {\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_SET_CALL_PARMS, \
            sizeof(IxDspCodeletSetCallParms) \
        )\
    }
```

Response:

General acknowledgement message (IX_DSP_CODELET_MSG_ACK)

7.2.8 Set Up Call with Parameters Message

Type: IX_DSP_CODELET_MSG_SETUP_CALLWPARMS

Direction: Inbound

Description: Setup a call with parameters. This involves two user messages, `IX_DSP_CODELET_MSG_SET_CALL_PARMS` to setup the call parameters, and `IX_DSP_CODELET_MSG_SETUP_CALL` to setup the call.

Format:

```
typedef struct{
    XMsgHdr_t               header;
    IxDspCodeletCallParms   parms;
    UINT8                   channelIP;
    UINT8                   channelTDM;
} IxDspCodeletMsgSetupCallwParms;
```

where IxDspCodeletCallParms is defined as:

```
typedef struct{
    UINT16  decAutoSwitch;
    UINT8   decType;
    UINT8   encType;
    UINT8   frmsPerPkt;
    UINT8   vad;
    UINT8   rfc2833;
    UINT8   rfc2833pyldType;
    UINT8   toneClamp;
} IxDspCodeletCallParms;
```

Macros:

```
#define IX_DSP_CODELET_MAKE_MSGHDR_SETUP_CALLWPARMS(pMsg, trans) \
```

```
    {\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_SETUP_CALLWPARMS, \
            sizeof(IxDspCodeletMsgSetupCallwParms) \
        )\
    }
```

Response:

General acknowledgement message (IX_DSP_CODELET_MSG_SETUP_ACK)

7.2.9 Switch Call Message

Type: IX_DSP_CODELET_MSG_SWITCH_CALL

Direction: Inbound

Description: Switches a call. This involves two user messages, `IX_DSP_CODELET_MSG_LINK_SWITCH` to switch an HSS termination to another IP termination, and `IX_DSP_CODELET_MSG_SETUP_CALL` to set up the call.

Format:

```
typedef struct{
    XMsgHdr_t   header;
    UINT8       channelTDM;
    UINT8       ipChanOnHold;
    UINT8       ipChanNewCall;
} IxDspCodeletMsgSwitchCall;
```

Macros:

```
#define IX_DSP_CODELET_MAKE_MSG_SWITCH_CALL(pMsg, trans, chTDM,
chHld, chNew) \
    {\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_SWITCH_CALL, \
            sizeof(IxDspCodeletMsgSwitchCall) \
        )\
((IxDspCodeletMsgSwitchCall *)(pMsg))->channelTDM = (chTDM);\
((IxDspCodeletMsgSwitchCall *)(pMsg))->ipChanOnHold = (chHld);\
((IxDspCodeletMsgSwitchCall *)(pMsg))->ipChanNewCall = (chNew);\
    }
```

Response:

General acknowledgement message (`IX_DSP_CODELET_MSG_LINK_ACK`)

7.2.10 Create Three-Way Call Message

Type: IX_DSP_CODELET_MSG_CREATE_3WCALL

Direction: Inbound

Description: Sets up a three-way call. This involves using user message `IX_DSP_CODELET_MSG_LINK` three times to connect each of the three parties in the three-way call to the mixer. Then a basic message is used to start the mixer resource.

Format:

```
typedef struct{
    XMsgHdr_t           header;
    IxDspCodeletTerm    parties[3];
} IxDspCodeletMsgCreate3wCall;
```

Macros:

```
#define IX_DSP_CODELET_MAKE_MSGHDR_CREATE_3WCALL(pMsg, trans) \
    {\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_CREATE_3WCALL, \
            sizeof(IxDspCodeletMsgCreate3wCall) \
        )\
    }
```

Response:

General acknowledgement message (`IX_DSP_CODELET_MSG_3W_ACK`).

7.2.11 Exit Three-Way Call Message

Type: IX_DSP_CODELET_MSG_EXIT_3WCALL

Direction: Inbound

Description: Exits a three-way call. This is the same as in `IX_DSP_CODELET_MSG_CREATE_3WCALL`, except the `IX_DSP_CODELET_MSG_LINK_BREAK` is used instead. Then a basic message is used to stop the mixer resource.

Format:

```
typedef struct{
    XMsgHdr_t           header;
    IxDspCodeletTerm    parties[3];
} IxDspCodeletMsgExit3wCall;
```

Macros:

```
#define IX_DSP_CODELET_MAKE_MSGHDR_EXIT_3WCALL(pMsg, trans) \
    {\
        XMSG_MA_MAKE_HEADER \
```

```
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_EXIT_3WCALL, \
            sizeof(IxDspCodeletMsgExit3wCall) \
        )\
    }
```

Response:

General acknowledgement message (IX_DSP_CODELET_MSG_3W_ACK)

7.2.12 Tear Down Three-Way Call Message

Type: IX_DSP_CODELET_MSG_TEARDOWN_3WCALL

Direction: Inbound

Description: Tear down a three-way call. This involves first using the user message IX_DSP_CODELET_MSG_EXIT_3WCALL to exit the three-way call. Then the user message IX_DSP_CODELET_MSG_STOP_IP is used to stop any IP channels that have been connected.

Format:

```
typedef struct{
    XMsgHdr_t           header;
    IxDspCodeletTerm    parties[3];
} IxDspCodeletMsgTeardown3wCall;
```

Macros:

```
#define IX_DSP_CODELET_MAKE_MSGHDR_TEARDOWN_3WCALL(pMsg, trans) \
    {\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_TEARDOWN_3WCALL, \
            sizeof(IxDspCodeletMsgTeardown3wCall) \
        )\
    }
```

Response:

Stop acknowledgement message (IX_DSP_CODELET_MSG_STOP_ACK)

7.2.13 Back to Two-Way Call Message

Type: IX_DSP_CODELET_MSG_BACKTO_2WCALL

Direction: Inbound

Description: Changes a three-way call to a two-way call. It involves using the user message `IX_DSP_CODELET_MSG_EXIT_3WCALL` to exit the three-way call. Then the user message `IX_DSP_CODELET_MSG_LINK` is used to create the two-way call. Then the user message `IX_DSP_CODELET_MSG_STOP_IP` is used to stop the IP termination if the disconnected party is one.

Format:

```
typedef struct{
    XMsgHdr_t           header;
    IxDspCodeletTerm    party1;
    IxDspCodeletTerm    party2;
    IxDspCodeletTerm    partyToDrop;
} IxDspCodeletMsgBackto2wCall;
```

Macros:

```
#define IX_DSP_CODELET_MAKE_MSGHDR_BACKTO_2WCALL(pMsg, trans) \
    {\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_BACKTO_2WCALL, \
            sizeof(IxDspCodeletMsgBackto2wCall) \
        )\
    }
```

Response:

General acknowledgement message (`IX_DSP_CODELET_MSG_3W_ACK`).

7.2.14 Set Clear Channel Message

Type: IX_DSP_CODELET_MSG_SET_CLEAR_CHAN

Direction: Inbound

Description: Sets a channel to clear channel. This involves five basic messages to set the parameters of the Encoder, Decoder, Tone Generator, Tone Detector, and Network resources, respectively.

Format:

```
typedef struct{
    XMsgHdr_t   header;
    UINT8       channelIP;
    UINT8       channelTDM;
    UINT8       codeType;
} IxDspCodeletMsgSetClearChan;
```

Macros:

```
#define IX_DSP_CODELET_MAKE_MSG_SET_CLEAR_CHAN(pMsg, trans,
chanIP, ChanTDM, code) \
    {\
```

```
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_SET_CLEAR_CHAN, \
            sizeof(IxDspCodeletMsgSetClearChan) \
        )\
((IxDspCodeletMsgSetClearChan *)(pMsg))->channelIP = chanIP; \
((IxDspCodeletMsgSetClearChan *)(pMsg))->channelTDM = ChanTDM; \
((IxDspCodeletMsgSetClearChan *)(pMsg))->codeType = code; \
    }
```

Response:

General acknowledgement message (IX_DSP_CODELET_MSG_ACK)

7.2.15 T.38 Switch-Over Message

Type: IX_DSP_CODELET_MSG_T38_SWITCH

Direction: Inbound

Description: Switches a channel between voice and T.38 fax modes.

Format:

```
typedef struct{
    XMsgHdr_t       header;
    UINT8           channelIP;
    UINT8           channelTDM;
    UINT8           mode; /* mode to switch, fax or voice */
} IxDspCodeletMsgT38Switch;
```

Macros:

```
#define IX_DSP_CODELET_MAKE_MSG_T38_SWITCH(pMsg, trans,
chanIP,ChanTDM,md) \
    {\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_T38_SWITCH, \
            sizeof(IxDspCodeletMsgT38Switch) \
        )\
((IxDspCodeletMsgT38Switch *)(pMsg))->channelIP = chanIP; \
((IxDspCodeletMsgT38Switch *)(pMsg))->channelTDM = ChanTDM; \
((IxDspCodeletMsgT38Switch *)pMsg)->mode = md;\
}
```

Response:

T.38 acknowledgement message (IX_DSP_CODELET_MSG_T38_ACK)

7.2.16 Set Parameters Message

Type: IX_DSP_CODELET_MSG_SET_PARMS

Direction: Inbound

Description: Sets parameters. It sends basic messages to set the parameters from an input list across the different resource components involved.

Format:

```
typedef struct{
    XMsgHdr_t   header;
    UINT16  numParms;
    IxDspCodeletParm parms[IX_DSP_CODELET_MAX_PARMS];
} IxDspCodeletMsgSetParms;
```

where `IxDspCodeletParm` is defined as:

```
typedef struct{
    UINT16  parmID;
    INT16   value;
    UINT8   dspResource;
    UINT8   dspResInstance;
} __attribute__ ((packed)) IxDspCodeletParm;
```

Macros:

```
#define IX_DSP_CODELET_MAKE_MSGHDR_SET_PARMS(pMsg, trans) \
    {\
        XMSG_MA_MAKE_HEADER \
        (   pMsg, \
            trans, \
            IX_DSP_CODELET_MSG_SET_PARMS, \
            sizeof(IxDspCodeletMsgSetParms) \
        )\
    }
```

Response:

General acknowledgement message (`IX_DSP_CODELET_MSG_ACK`).

7.3 Pre-Defined User-Response Messages

7.3.1 Acknowledge Message

There are three Acknowledge messages which are of the same format but corresponding to different control messages.

Type:

`IX_DSP_CODELET_MSG_ACK`

`IX_DSP_CODELET_MSG_LINK_ACK`

IX_DSP_CODELET_MSG_SETUP_ACK
IX_DSP_CODELET_MSG_T38_ACK.

Direction: Outbound

Description: Acknowledge messages to user control messages.

Format:
```
typedef struct{
    XMsgHdr_t           header;
    INT16               numDspReplies;
    INT16               numErrors;
    IxDspCodeletError   error[IX_DSP_CODELET_MAX_ERR_REPLY];
} IxDspCodeletMsgAck,
IxDspCodeletMsgLinkAck,
IxDspCodeletMsgSetupAck,
IxDspCodeletMsgT38Ack;
```

where IxDspCodeletError is defined as:
```
typedef struct{
    UINT32  errData;
    UINT16  errCode;
    UINT8   dspResource;
    UINT8   dspResInstance;
} IxDspCodeletError;
```

7.3.2 Stop Acknowledge Message

Type: IX_DSP_CODELET_MSG_STOP_ACK

Direction: Outbound

Description: Stops acknowledge message to user stop messages.

Format:
```
typedef struct{
    XMsgHdr_t             header;
    INT16                 numDspReplies;
    INT16                 numErrors;
    IxDspCodeletError     error[IX_DSP_CODELET_MAX_ERR_REPLY];
    INT16                 numStopAck;
    IxDspCodeletStopCmplt stopAck[IX_DSP_CODELET_MAX_STOP_CMPLT];
} IxDspCodeletMsgStopAck;
```

where IxDspCodeletError is defined above and IxDspCodeletStopCmplt is defined as:
```
typedef struct{
    UINT32  totalFrames;
    UINT8   dspResource;
} IxDspCodeletStopCmplt;    UINT8   dspResInstance;
```

Application Examples 8

8.1 IP Interface

The Intel® IXP400 DSP Software uses two interface functions to transfer encoded audio packets to and from the IP interface. These audio packets are transferred on the IP network as RTP (Real-time Transport Protocol) packets. RTP packets are UDP (User Datagram Protocol) packets with a 12-byte RTP header at the beginning of the UDP payload. UDP packets are suitable for transmitting real-time media data since they are low on overhead and thus provide speedy delivery, though packet delivery is not guaranteed.

The RTP packet streams need to be extracted from the overall incoming IP traffic in the ingress direction, and merged to outgoing IP traffic in the egress direction. One way to do this is to examine the IP packets in the Ethernet driver. Incoming RTP packets are routed to the DSP software, while other IP packets are sent to the user's IP stack. Another way is to route all IP traffic to the user application from the Ethernet driver. Then the user uses standard interfaces, such as sockets, to route the appropriate traffic to the respective parties.

The advantage of the second approach is that socket functionalities are already provided both by the VxWorks or Linux operating systems. The `EthAcc` interface of the DSP software is integrated with the Ethernet driver and the user application developers do no need to worry about this service. The user application is only required to perform initialization and then exchange control messages and data packets between the DSP software and the application. Because an application can open multiple sockets as needed, the resource in the DSP software can be shared by multiple clients.

A typical VoIP application using sockets will consist of a few tasks that handle either the data packets or the control messages. Figure 12 and Figure 13 on page 61 show two possible implementations in VxWorks and Linux, respectively.

In VxWorks, the application has direct access of the control and data interface. This makes it straight forward to exchange control messages and data packets between the application and the DSP software. As shown in Figure 12, two tasks are spawned in the application. One uses `xMsgReceive()` to wait for the event and response messages and the other waits for packets from the socket. Control messages are sent directly by calling `xMsgSend()`, and Packets received from the socket will be passed by calling `xPacketReceive()`. The DSP software will directly call a call back function, for example `packetSendCB()`, which is registered with `xDspSysInit()` during initialization, to send packets through the socket.

The case for Linux is slightly different because the application usually runs in user mode, while the DSP software runs in kernel mode. To get the event and response message to the application, a task is needed in user mode — using `xMsgReceive()` to wait for messages from the DSP software. Another task running in user mode waits for packets from the socket and then uses driver-write function call to pass the packets to the driver which then call `xPacketReceive()` to pass the packet along to the DSP software. Control messages are also delivered by using driver write function calls and then `xMsgSend()` is called in kernel mode.

The application may choose to send and receive packets through sockets with a single port. A channel ID can be embedded in the package so that the package can be passed to the corresponding the DSP software channel based on the ID number. Alternatively, the application may choose to map the socket port number with the DSP software channel.

Figure 12. Intel® DSP Software Application in VxWorks*

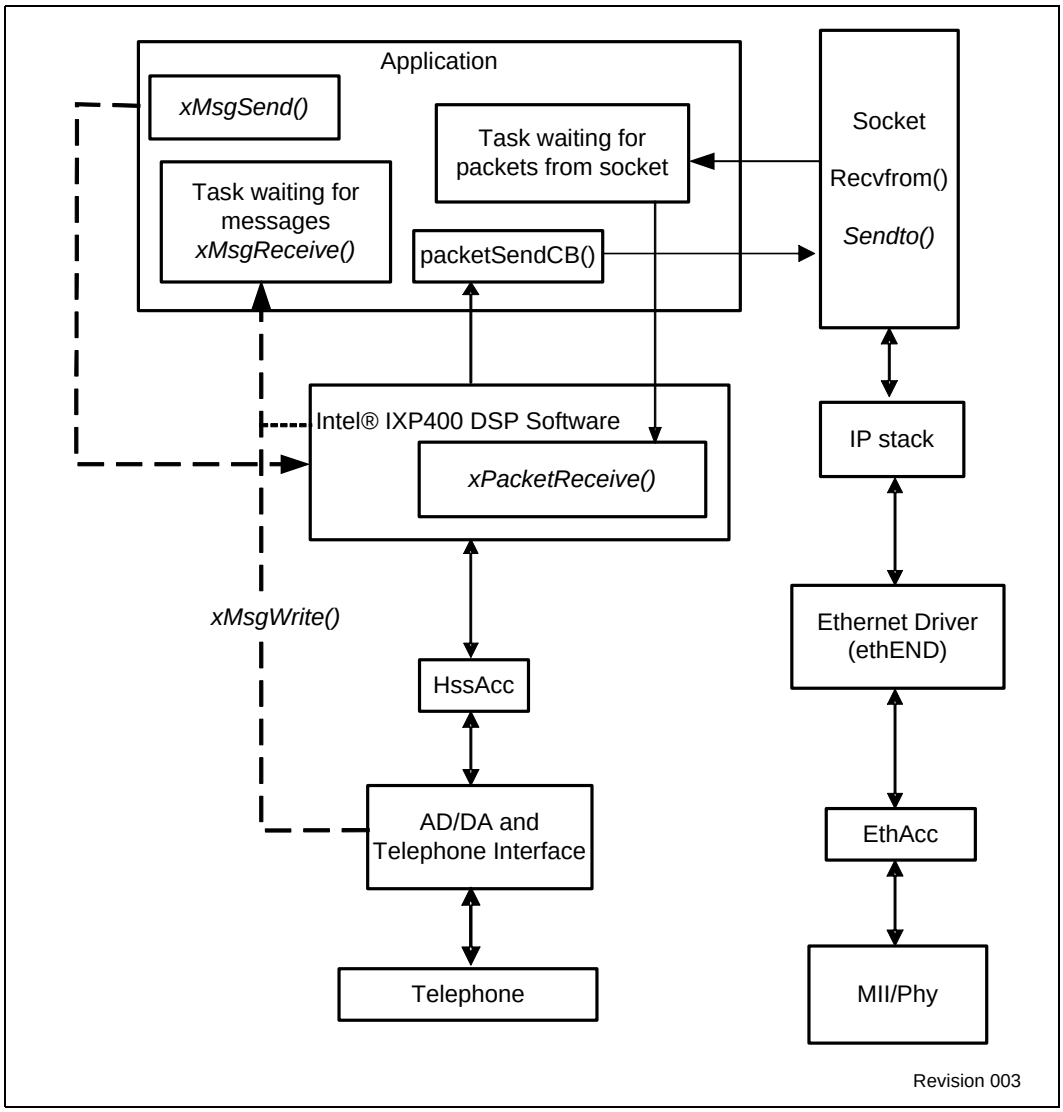
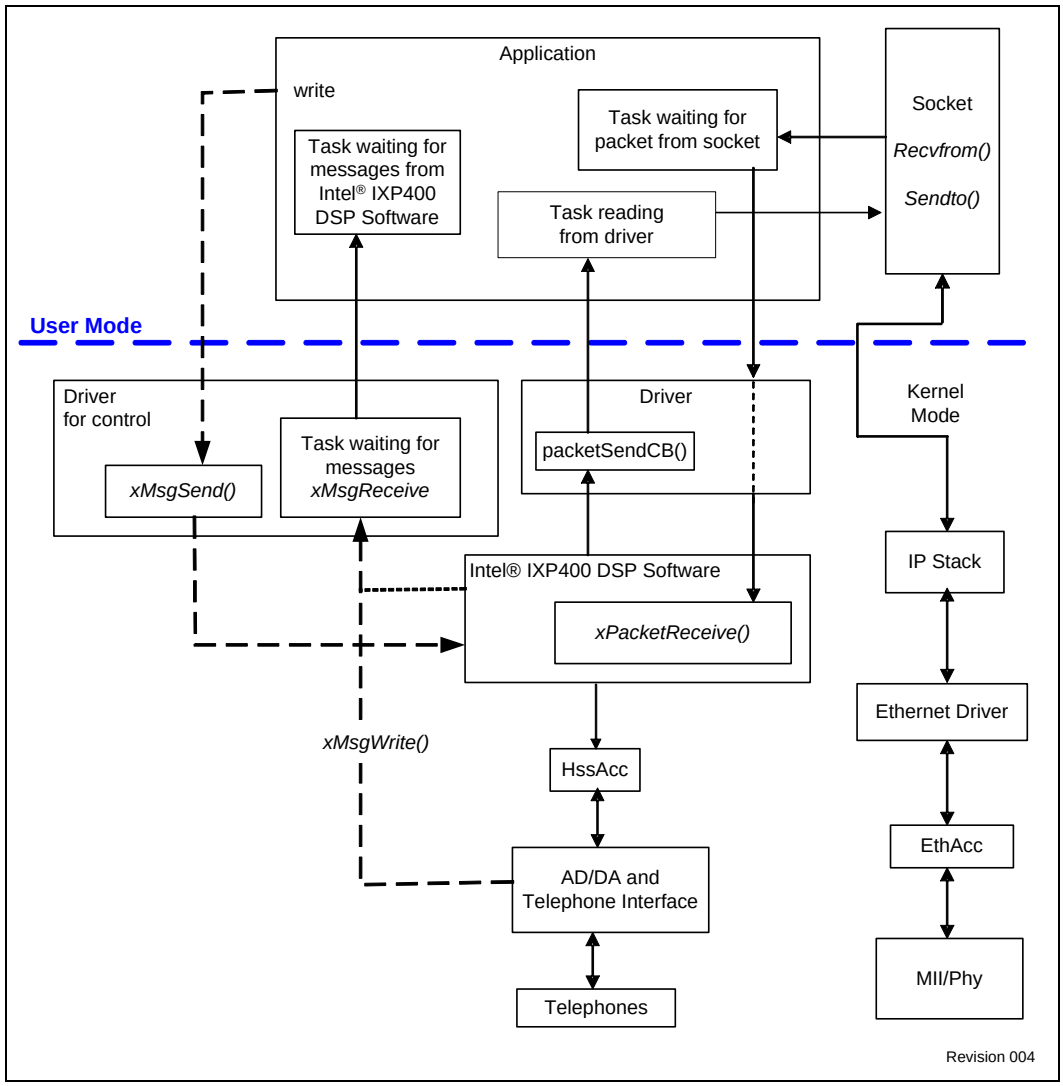


Figure 13. Intel® DSP Application in Linux*



8.2 Caller-ID Generator

The FSK modulator provided in the DSP software is designed primarily to allow user applications to implement the caller-ID generator. It should be noted that caller-ID generation is a function of the user application, since it involves direct interaction with the specific SLIC interface being used.

The caller ID specifications are country-specific and some of them can be found in the documents of Bellcore* 202 for the United States, Technical Specification YDN 069-1977 for China[1], and NTT Technical Reference - Telephone Interfaces, Edition 5 for Japan.

1. In this document, all references to China refer to the People's Republic of China.

In this release, the demo source codes are included to show how to implement U.S., China, and Japan caller-ID generators on the evaluation platform using the FSK feature according to these specifications.

To implement caller-ID generation, the user's applications are responsible to provide the following functions in addition to FSK modulator:

- Generate the complete caller ID data to be transmitted by the FSK modem. The data must be represented in octet (byte) without mark, start, and stop bits. The demo code includes useful utilities that can build the caller ID data format from the information to be displayed and add parity check bits, CRC octets, or check sum if necessary.
- Control the SLIC device to generate the signals such as polarity reverse, short ring (CRA), and normal ring as required by the caller-ID specifications
- Detect the loop connection/disconnection (or off-hook/on-hook status) for Japan caller ID. SLIC driver may report such events through the outbound message queue using the complementary function of hook-event detection.
- Provide timer service using OS services, based on hardware or software resources. The built-in complementary timer service function in the NET component in DSP software can be used for this purpose. The timer events can be reported through the outbound message queue.
- Implement the state machine that follows the signal flow diagram of the caller ID as described in NTT specifications. The data ID data as we discussed above are transmitted using the FSK modem function in the proper state. The demo code gives an example of such state machine.

Most of other country-specific caller-ID generators can be implemented similarly. Some caller-ID specifications, like Japan, require the FSK data to be transmitted in off-hook state, while others transmit the data in on-hook state. The procedure of on-hook transmission is simpler because the interactions between SLIC device and the caller-ID receiver are no longer necessary.