intel®

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

Programmer's Guide

March 2005

Document Number: 252725-001a

Contents

Figures

Tables

Revision History

Date	Revision	Description
March 2005	001a	Branding and document-title updates.
March 2003	001	Initial release of this document.

Introduction 1

Intel® IXP400 Digital Signal Processing (DSP) Software Library Release 1.1 (DSP 1.1) is a software module that provides the basic voice and signal-processing functionalities for voice-over-Internet-protocol (VoIP) applications on the Intel® IXP42X Product Line of Network Processors. This document explains how to use the DSP API and provides the guideline and examples to the application developers.

1.1 General

DSP is a software module for media processing, targeted for next-generation integrated-access devices (IADs) such as consumer-premise equipment (CPE). Specifically, DSP performs media compression, echo cancellation, tone processing, jitter control, and other functions required in any IP media gateway or real-time media streaming functionalities.

This document is intended to describe the control and data interfaces of DSP to enable a third-party developer to incorporate DSP into a media gateway or server system and to integrate with other client software. Together with the *Intel® IXP400 Digital Signal Processing Software Library Release 1.1 API Reference Manual*, this document provides sufficient detail — about the interfaces and message and data delivery mechanisms — that a DSP software client can fully configure and control DSP processing operations and services.

1.2 Scope

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

1.3 Audience

This document is intended for third-party software developers who are using DSP 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 Software Library Release 1.1 API Reference Manual	273811
Intel® IXP400 Digital Signal Processing Software Library Release 1.1 Release Notes	273810
Intel® IXP400 Product Line Programmer's Guide (Version 1.1)	252539

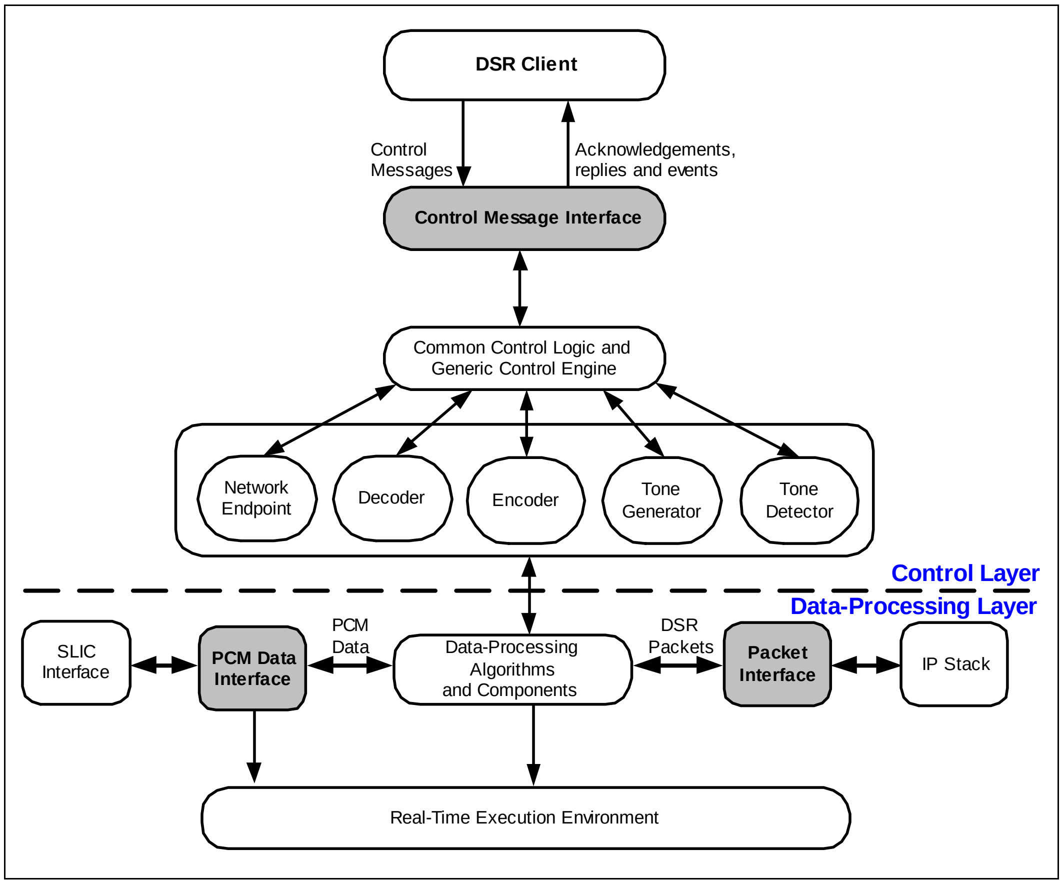
Architecture Overview 2

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

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

Figure 1 shows the logic decomposition of DSP modules. The shaded blocks represent the control and data interfaces between DSP and other software modules external to DSP.

Figure 1. Intel® IXP400 DSP Software Library Release 1.1 Software Architecture



From the control point of view, a DSP channel consists of a set of media-processing resource (MPR) components. Each MPR is an addressable entity and can be controlled independently. This provides the maximum flexibility for setting up a channel with various resource configurations — for example, half-duplex call or asymmetric Rx and Tx CODEC types.

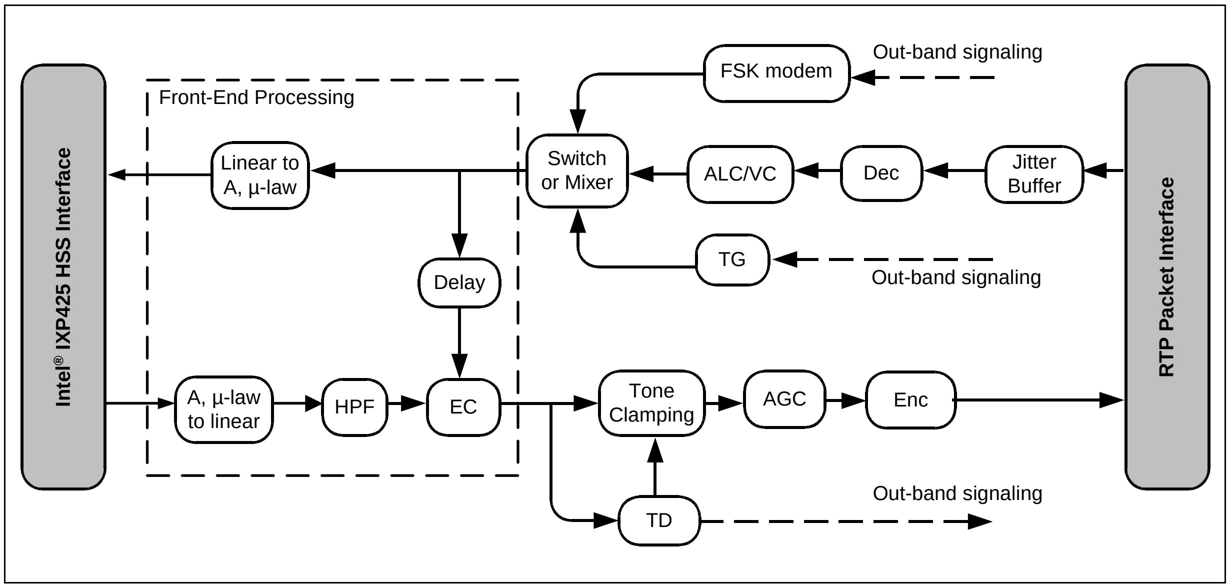
From the perspective of data flows, the data-processing functions are depicted in Figure 2. All the functions are executed by real-time tasks that are set up by DSP, during initialization.

There is one task for each unique coder frame rate. For example, there will be a 10-ms task and a 30-ms task — if both G.729 (10 ms frames) and G.723 (30 ms frames) are supported. The 10-ms task also handles all other, non-coder voice processing, such as echo cancellation and tone detection.

Currently, there is only one 10-ms, real-time task because only 10-ms coders are supported. The real-time tasks are of higher priority than the control task and are synchronized (triggered) by the high-speed serial (HSS) Network Processing Engine (NPE).

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

Figure 2. Data Flow and Data-Processing Functions

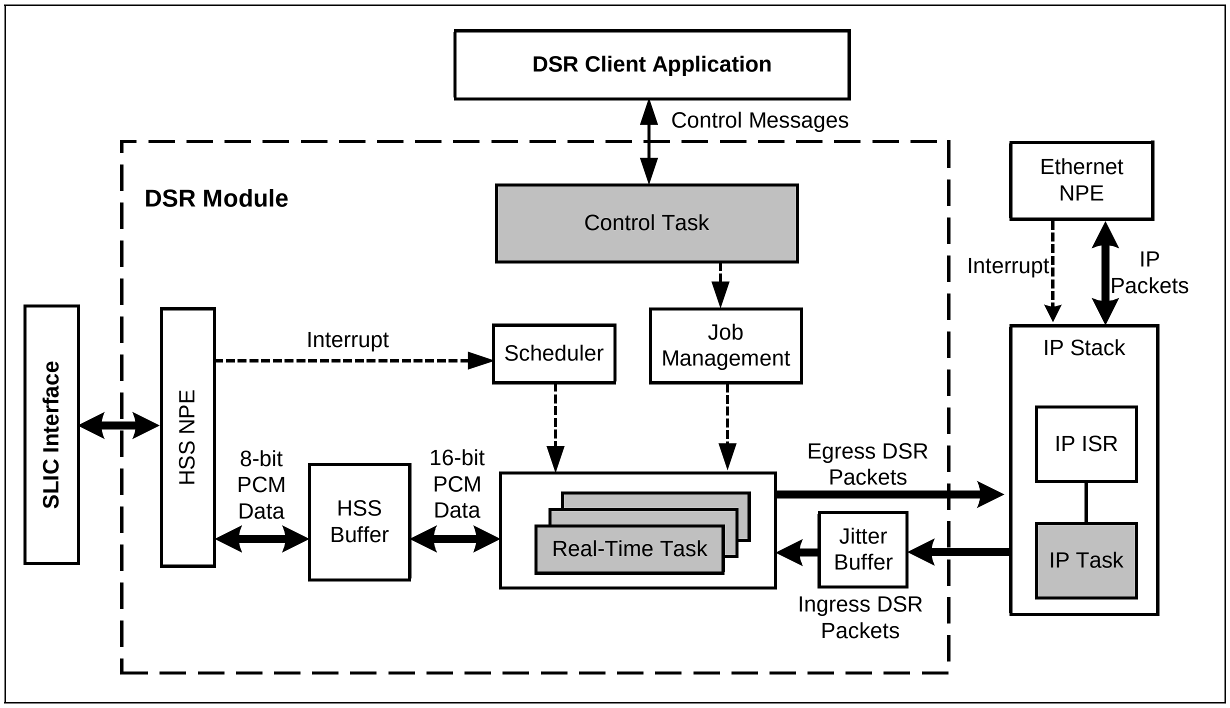


The relation among the messages, data, and tasks — inside and outside DSP — is illustrated by Figure 3. That relationship can be summarized as:

- The control task is driven by the inbound messages from the user application.
- The real-time tasks are synchronized with the data from HSS interface. The HSS NPE signals DSP's scheduler via ISR call-back routine, 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 DSP-encoded, audio packets at the fixed rates essentially synchronized with PCM data.
- DSP-encoded, audio packets arrive at variable rate asynchronously, with the real-time tasks.

It is important to understand that the real-time tasks in DSP are characterized by their hard task deadlines. If a real task cannot finish its processing before the next task period, data will be lost and — consequently — voice quality is seriously degraded. That may happen if the real-time task is preempted for long periods time by ISR, or other tasks, of if the processor is overloaded.

Figure 3. Intel® IXP400 DSP Software Library Release 1.1 Message, Data and Tasks



Run-Time Interfaces 3

Intel® IXP400 Digital Signal Processing (DSP) Software Library Release 1.1 (DSP) is implemented as an independent module executed by its own tasks. User applications do not directly access DSP's internal functions or data.

DSP provides three interfaces for the applications to communicate control information, PCM data, and encapsulated voice packets, respectively, in run-time. See Figure 1 on page 7 and Figure 2 on page 8.

3.1 Control Interface

Applications communicate with DSP primarily through the control interface defined as a set of functions, messages, and macros. There are two message queues in the control interface — for the inbound messages, from applications to DSP and out-bound messages, going in the other direction. (See Figure 4.)

Two interface functions — `xMsgSend()` and `xMsgReceive()` — can be used for the application to send and receive messages to/from the queues, respectively.

DSP spawns a dedicated control task, pending on the inbound message queue, to handle the control messages. DSP is isolated from user applications by message queues to keep DSP control functions from being accessed by multiple tasks of the user application. This is because making the control functions multi-task-safe involves extra complexity and performance penalty.

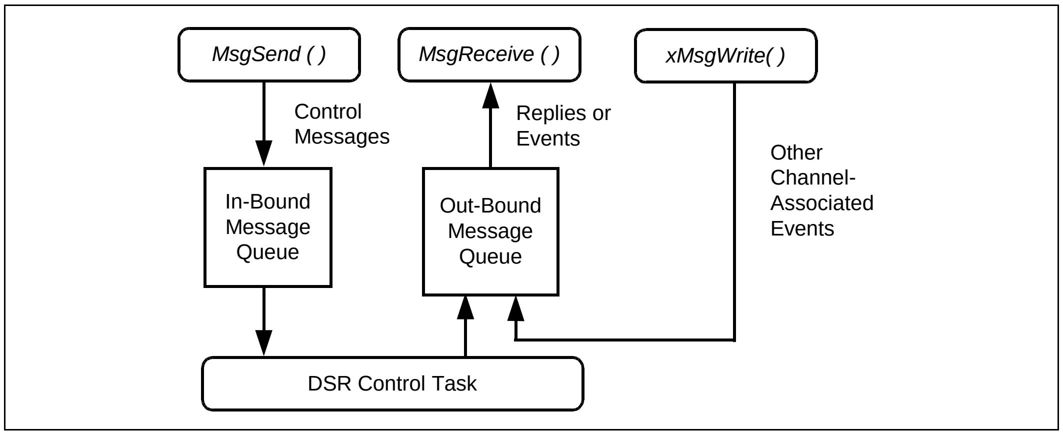
DSP sends the replies or events to the application through the out-bound message queue. The application can retrieve the messages using *`xMsgReceive()`*. The caller's task of *`xMsgReceive()`* will be blocked forever or until time out if the out-bound queue is empty. (In future DSP's, an optional per-channel-basis out-bound queue mechanism may be implemented so that the application can have multiple tasks and each task is associated with a channel.)

A third function of the DSP control interface — `xMsgWrite()` — allows the application to directly post a user-defined message to the DSP's outbound message queue, if necessary. This enables the user application to receive all channel-associated events from DSP, even though some of these events are not controlled by DSP.

For instance, the application may hook a call-back function to the ISR that reports the SLIC interface's on/off hook events. In the call-back function, a user-define message is sent to DSP's outbound message queue to signal the event to the user application.

(In future DSPs, an enhancement may be added to allow the user application to hook a callback that is called whenever a message is put into the out-bound queue.)

Figure 4. Control Interface and Message Queues



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 inbound queue may overflow, if the user application is of higher priority than DSP control task. Alternately, the outbound queue may overflow if the user application has lower priority.

DSP uses copy-based message delivery — the entire message context actually copied from the deliverer to the receptor — rather than a pointer being passed. This avoids dynamically allocating memory for the messages. Since no memory is shared between DSP and the application, the application can reuse the memory of a message for any other purpose, immediately after the message is sent.

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.

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, and size. A 4-byte transaction ID is provided to allow the user application to keep track of the replies or events.

When DSP sends a reply or event message to the user application, it copies the transaction ID from the associated message originated from the user application. For details on the control message format, see the *Intel® IXP400 Digital Signal Processing Software Library Release 1.1 API Reference Manual*.

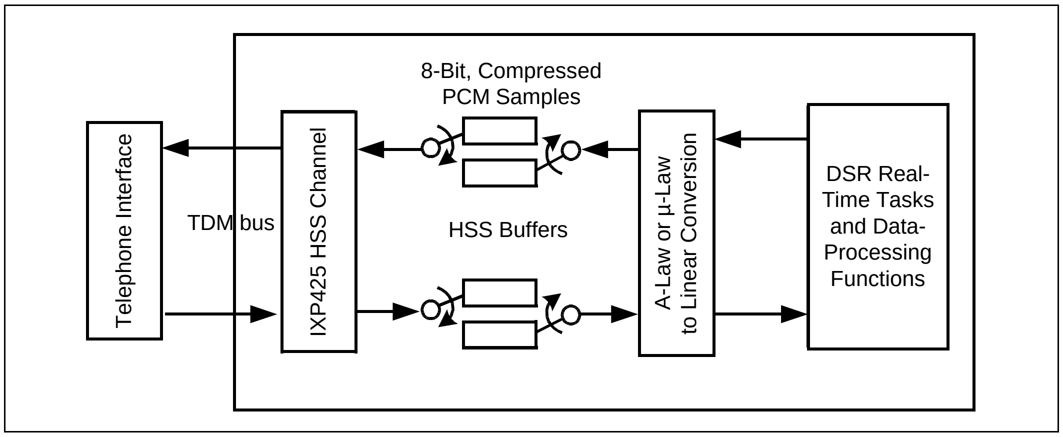
3.2 PCM Data Interface

PCM data represents the audio data stream between DSP and the telephone interface, via the TDM data bus. DSP's PCM data interface relies on the HSS hardware integrated in the IXP42X processor.

In contrast to the data network interface — such as the Ethernet interface — the HSS interface is integrated as part of DSP. 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, through parameters passed to DSP during initialization. From the user application's perspective, the HSS can be viewed as a piece of hardware to be properly configured. Once it is configured and started, there is no further user-application involvement.

Figure 5. PCM Data Interface


8-Bit, Compressed PCM Samples
Telephone Interface
TDM bus
IXP425 HSS Channel
HSS Buffers
A-Law or µ-Law to Linear Conversion
DSR Real-Time Tasks and Data-Processing Functions

The user application configures the HSS by specifying the signal format to be presented on the 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` and `IxHssAccTdmSlotUsage`.

Using this set of information, DSP initializes the HSS interface and starts data transfers. For more details, see the *Intel® IXP400 Product Line Programmer's Guide (Version 1.1).*

Currently, DSP can only handle PCM data in 8-bit, compressed A-law or µ-law format at the rate of 8 K samples per second.

Internally, DSP's real-time tasks are synchronized with HSS data transfer. DSP's scheduler is signaled by the HSS driver (in an interrupt context), each time a 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.

3.3 Packet Interface

Compared to PCM Data Interface, the Packet Interface is a pure software protocol that defines how DSP and the IP packet interface exchange the encoded-audio data packets. There are two functions and a packet format involved in the Audio Packet Interface as shown in Figure 6.

DSP defines the packet format and provides the packet receive function. The user application is responsible for providing the transmit function.

During ingress (packets going from the IP interface to DSP), the IP interface converts each incoming VoIP packet it receives to a DSP data packet. It calls `xPacketReceive()` to deliver the packet to DSP.

The user application needs to decode the incoming IP packets to forward the RTP packet payloads with the proper DSP header format and the extracted RTP time stamp, to the proper DSP channel. This function copies the packet to the jitter buffer without further processing.

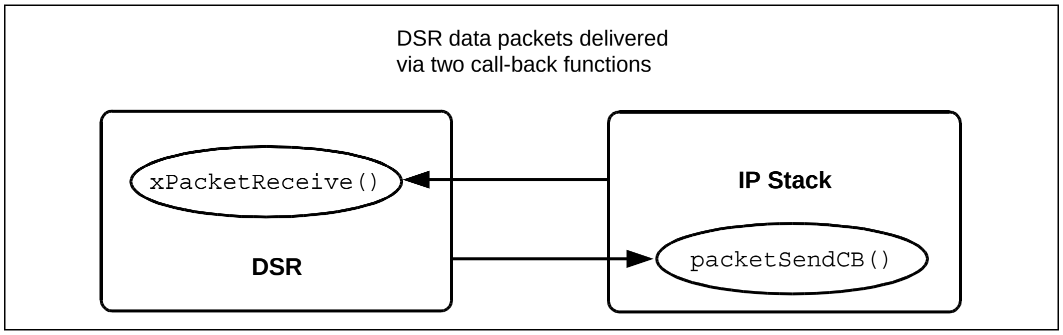
This enables `xPacketReceive()` to be called from an interrupt-service routine context, but re-entry is not allowed. Since the incoming DSP packets are copied by DSP, the caller of the `xPacketReceive()` can free or reuse any memory it may have allocated to buffer the incoming RTP packets upon return from the function.

During egress (packets going from DSP to the IP interface), the application registers a call-back function with DSP. This function is supposed to deliver the DSP data packet to the IP interface and sends it out. DSP always prepares the memory for the packet and fills the packet header information (including local timestamp) and packet payload before it calls the function. This user-provided function should create and encode the RTP header with the time stamp provided in the DSP data packet header.

After returning from the function, DSP will immediately reuse the memory for other purposes. Therefore, it may be necessary for the call-back function to make a copy of the packet. Since the function is called from DSP's real-time tasks — at regular basis each time when a packet is generated — there are two additional requirements to the callback function:

- It must finish as soon as possible without any blocks inside. This allows real-time data to be acquired and processed without data loss in DSP.
- It must be multi-task-safe (i.e., allows re-entry).

Figure 6. Packet Interface



To improve efficiency, another packet-delivery method is being considered in a future DSP release that can avoid copying the packet data. In that method, the packet receiver registers two call-back functions with the deliverer. The packet deliverer calls the first call-back function, to obtain the memory supplied by the receiver for a packet and to deliver a packet. The memory block typically resides in the working memory space of receiver. So it produces the packet directly in the receiver's memory space and calls another call-back function to indicate packet available.

Components, Features, and Parameters 4

An Intel® IXP400 DSP Software Library Release 1.1 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.

This chapter discusses the MPR components and their features and parameters.

4.1 Network Endpoint

The Network Endpoint component is a front-end, data-processing unit connecting the high-speed serial (HSS) interface to the rest of MPR components.

In the Tx direction (from DSP to HSS), the component converts the audio data from 16-bit, linear format to 8-bit, A-law or µ-law format and sends it to the HSS output buffer. In the Rx direction, the component receives the 8-bit, compressed audio data from the HSS buffer; converts it to 16-bit, linear format; and applies high-pass filter (HPF) and echo cancellation (EC). The HPF is of the 3-dB cut-off frequency at 270 Hz.

The application can specify A-law or µ-law of the 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 may be useful for debugging purposes.

In the bypass mode, no other processing can be applied by other MPR components except for the pass-through type CODEC.

Echo cancellation (EC) is the most-significant function in this component. EC cancels the echo generated by the hybrid of local telephone interface and telephone set, so the other party on 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 — `XPARMID_NET_ECTAIL`
- Delay compensation — `XPARMID_NET_DELAYCOMP`

Depending on the hardware circuits and telephone set, the tail length of 4 ~ 8 ms is usually good enough if the telephone set is directly connected to the unit. Since EC is very computation-intensive, a longer tail length results in higher CPU occupancy.

Changing the parameter of EC tail length requires the Network Endpoint component to be reset. This is done by sending the `XMSG_RESET` message.

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 default value of the parameter was determined according to DSP performance on the Intel® IXP425 Residential Gateway Demonstration Platform. (A utility may be provided in future DSPs to automatically determine the delay compensation in customized platforms.)

The Network Endpoint component is started automatically when DSP is initialized. The application can still start or stop it, using the `XMSG_START` or `XMSG_STOP` message for debug and test purpose. Stopping the component stops EC and HPF, but the audio data stream continues.

4.2 Encoder

The primary function of this component is to encode and packetize the audio data from HSS and send that data to the IP interface. The audio CODEC supports G.711 and G.729 on 10-ms frame size. Other features include Automatic Gain Control (AGC), Voice Activity Detector (VAD), and Multiple Frame per Packet (MFPP).

This section discusses these encoder features and their effects on voice quality and system performance.

There are two automatic gain control elements in DSP:

- In the egress side (from HSS to DSP to IP interface) — AGC
- In the ingress side (from IP interface to DSP to HSS) — Automatic Level Control (ALC)

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, so it is recommended that AGC be turned off and ALC turned on. This is the default.

The VAD algorithm can distinguish active speech signal from silence (background noise). During the silence period, the encoder sends much smaller packets containing only the noise parameter at much lower rate. This 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 with G.729 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.

Packing more frames into a packet (i.e., MFPP) is another way to reduce network traffic. The application specifies the number of frames per packet in `XMSG_CODER_START` message when it starts the encoder. 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.

Typically, the user application starts encoder when a call is set up and stops it when the call is torn down.

4.3 Decoder

The Decoder receives the encoded audio packets from the IP interface and converts them to the audio stream and sends it to the HSS interface. Similar to the encoder, the decoder in DSP supports G.711 and G.729 coder types of 10-ms frame size 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 coder, CNG is built into the decoder algorithm 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.

The PLC algorithm uses the previous speech signal to repair the lost frames, but cannot repair any big chuck of consecutive frame lost. Because of the complexity of PLC algorithm, it will increase the processor occupancy during packet loss, when using G.711 coder. But since G.711 is a low-computation coder, the resultant processor occupancy rate is still much lower than that of G.729. 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 set up, 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 is limited by internal buffer size. For G.711, the maximum number of frames per packet is three. For G.729, the maximum number of frames per packet is 24.

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 the 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 correctly to the HSS interface. This delay is dynamically adjusted by the jitter buffer, depending on IP network conditions.

The jitter buffer monitors network conditions by checking the time stamps of the incoming DSP packets against the local clock. (The correct sequencing of audio packets also is done with the help of the time stamp.)

The jitter buffer implements two algorithms for monitoring network conditions, one specified by RFC 1889, the other one a proprietary delay-profiling algorithm. Because the proprietary algorithm provides better tracking and improves voice quality, this algorithm is always enabled.

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

Typically, the user application starts the decoder with the encoder, when a call is set up and stops it when the call is torn down.

4.4 Tone Generator

The tone generator is capable of generating single- or dual-frequency tone- and amplitude-modulated tone. 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. All the tone templates, including DTMF and call progress tones, are pre-defined. (An API may be added in future DSPs to allow the application to create user-defined tone templates.)

Since call-progress tones are country-specific, the application has to set the country code so that the tone generator can select the correct template table.

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

If tones are played while the 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 speech is muted during the whole tone period.

However, some tones have the cadence of a tone-on duration followed by a silence 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 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.

Currently, DSP only supports the overwrite mode. If a continuous tone (e.g., call-progress tone) is played, the user application has to stop it explicitly using the `XMSG_STOP` message.

Tone generator can also generate FSK modem signals compliant to V.23 and Bellcore-202 specifications. 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 is beyond the scope of the current DSP.

4.5 Tone Detector

The tone detector is able to detect single- or dual-frequency tones with a frequency range from 300 ~ 3,500 Hz, using an FFT analyzer. To reliably detect a dual tone, the frequencies of the dual tone signal must be separated by at least 200 Hz. Internally, all the tones to be detected (i.e., DTMF tones and fax tones) are described by a list of templates that contain the criteria of frequencies, energy, SNR, durations, and so on. (An API — to allow the user application to add user-defined tones — may be implemented in the future.)

To detect tones, the user application needs to start the tone detector by sending the `XMSG_START` message and enabling tone-event reporting by setting the parameter `XPARMID_TD_RPT_EVENTS`. DSP will report tone-on and/or tone-off events, depending on the parameter.

Instead of being notified by tone events, the user application may want to receive a DTMF digit string, e.g., a telephone number entered from the telephone set. For this purpose, the user application can use the `XMSG_TD_RCV` message and specify the number of digits to expect and the termination conditions. Tone detector will return the result via the `XMSG_TD_RCV_CMPLT` message, once the digits are collected or the termination conditions are met. (This feature is targeted for future DSPs and is not currently supported.)

Another feature of the tone detector is tone clamping. This features mutes the input audio stream from the HSS interface 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 a 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).

Enabling the look-ahead buffer, however, will increase the latency accordingly.

Programming Guide 5

This chapter discusses the rules and guidelines for building user applications on top of Intel® IXP400 Digital Signal Processing (DSP) Software Library Release 1.1 (DSP).

5.1 Initialization

Because DSP is a stand-alone module or a layer of media processing, it must be initialized properly before the application can interact with it. The initialization involves a few simple steps:

1. Before calling any DSP functions, the user application must first initialize DSP by calling `xDspInit()`.

   At this time, DSP creates its tasks, algorithm, and component instances, message queues, etc. Upon successful initialization, DSP is ready to receive control messages.

2. Download HSS NPE and initialize HSS dependents.

   See the *Intel® IXP400 Product Line Programmer's Guide (Version 1.1)*.

3. Set country code by calling `xSetCountryCode()`.

   This enables the country-specific features. (Currently, that is the call-progress tones.)

4. Initialize PCM data interface by calling `xDspHssInit()`.

   The application must provide the description of the signal formats and time-slot assignment on HSS's TDM bus, as defined by the data structures `IxHssAccConfigParams` and `IxHssAccTdmSlotUsage`.

5. Register the callback function for the encoder to deliver encoded packets to the IP interface by calling `xRegistPktRcvFxn()`.

5.2 Programming Model

A VoIP gateway application may contain several modules such as user interface, IP call stacks, and DSP. 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 and dropping the calls and disconnecting the data path, accordingly.

From control point of view, this procedure can be characterized as the interactions among DSP, IP call stack, and SLIC driver — through asynchronous messages and commands. Such control logic is best implemented by a message-driven state machine model. DSP'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.

All the control messages, commands, and events from IP call stack and SLIC driver are consolidated to the DSP message queue. They are formed as user-defined messages and are posted to DSP's message queue by using `xMsgWrite()`. (In Linux, this can be done in DSP's client driver module.) The task then is pending on the message queue, using `xMsgReceve()`, to handle all the call progress-related messages from all these modules.

Figure 7 shows the general approach of such a state-machine model. In this programming model, a call-progress scenario is represented by a sequence of states. Each state is characterized by:

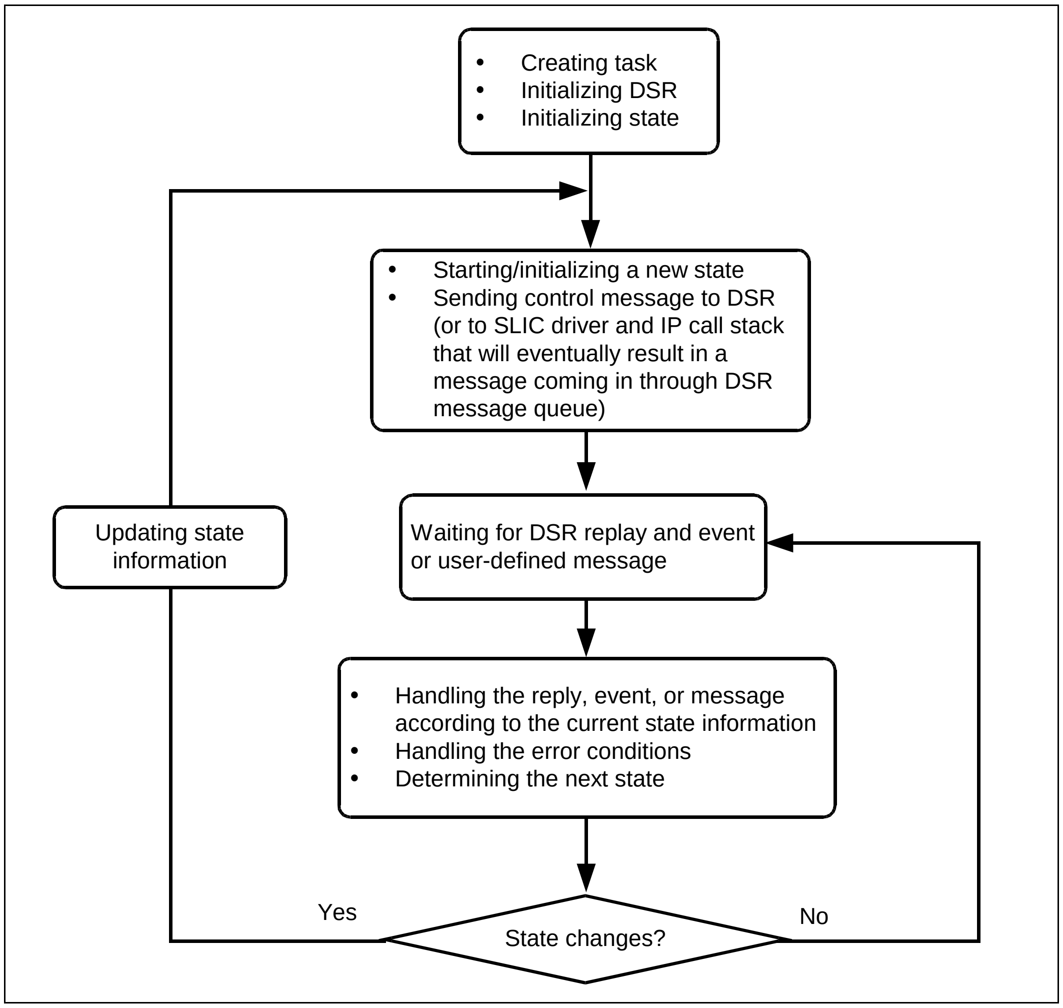
- The actions it takes
- The messages it expects
- The next state it goes to.

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

1. Idle State – Waiting for call setup message from the IP call stack.
2. Ring State – Ringing the local telephone set and waiting for off-hook event.
3. Channel Set-Up State – Sending control messages to DSP to start encoder, decoder, and tone-detector resources and waiting for the acknowledges.
4. Connected State – Acknowledging the IP call stack that a local channel has been set up. Waiting for a disconnect message from the call stack or on-hook event from local telephone set.
5. Tear-Down State – Sending control messages to DSP to stop the resources and waiting for acknowledgements; acknowledging IP call stack that the channel has been torn down; and going back to the Idle State.

The actual state machine will be more complicated, considering all the possible error conditions. For example, if the call is not answered, a time-out message must be handled in Ring State.

Figure 7. Flow Chart of General-State-Machine Approach


• Creating task
• Initializing DSR
• Initializing state
• Starting/initializing a new state
• Sending control message to DSR (or to SLIC driver and IP call stack that will eventually result in a message coming in through DSR message queue)
Updating state information
Waiting for DSR replay and event or user-defined message
• Handling the reply, event, or message according to the current state information
• Handling the error conditions
• Determining the next state
Yes
State changes?
No

The major advantage of such programming model is the high efficiency and good performance. In Linux*, it also helps the DSP to maintain its real-time behavior.

OS-Specific Issues 6

Because of the substantial difference between the VxWorks* and Linux* operating systems, Intel® IXP400 Digital Signal Processing (DSP) Software Library Release 1.1's internal real-time environment and interface are implemented differently. The exposed APIs, however, look identical.

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

6.1 VxWorks*

Implementation of DSP and user applications 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, multi-tasking environment with enormous supporting functionalities
- All the software modules reside under the same memory address space

In the current DSP release for VxWorks, two tasks are spawned and two sets of task properties are reserved for future use, as shown in Table 1. (The higher the number, the lower the priority.) The priorities of DSP's real-time tasks are assigned according to rate-monotonic scheduling (RMS) — the higher-frequency periodic task getting the higher priority.

Table 1. VxWorks* Tasks and Task Properties

Task Name	Priority	Description
Ctr_Task	44	Control task. Pending on inbound message queue. Triggered by incoming control messages.
Task30	43	(Reserved for future G.723 and fax-modem algorithms.) Real-time task. Wake up every 30 ms, synchronously with PCM data.
Task20	42	(Reserved for future GSM algorithms.) Real-time task. Wake up every 20 ms, synchronously with PCM data.
Task10	41	Real-time task. Wake up every 10 ms, synchronously with PCM data. Execute all the DSP algorithms supported in current Intel® IXP400 DSP Software Library Release 1.1.

Although user applications cannot change any properties of DSP tasks, they have to assign their task priorities properly to coexist with DSP. The rules for the user applications are:

- A user's control task — that is not involved in data and packet processing — should not have higher priority than DSP's control task.
- A user's time-critical task may have a higher priority than DSP's real time tasks, only if its execution is predictable and does significantly affects DSP's real-time task — not preempting DSP's real-time tasks for more than 1 ms total, in each 10-ms period. (DSP takes about 5 ms in such period.)

- The applications must not send a burst of control messages to DSP without waiting for the replies. Otherwise, the message queues may overflow.

In future release, a configuration API will be added to allow users to reassign DSP's task priorities especially if the user application also has periodic real-time tasks to coexist with DSP.

6.2 Linux*

Linux will be the choice for DSP, if the cost of the target products is the major consideration. However, some extra development efforts and cautions must be taken. This is because:

- Linux is not a real-time OS and does not support priority-based, preemptive multi-tasking
- User-mode applications cannot directly access DSP interfaces which reside in kernel-mode.
  A shim layer of driver software must be developed to allow the user application to communicate with DSP.

The DSP in Linux is fully in kernel mode. The DSP creates the kernel mode threads shown in Table 2.

Table 2. Linux* Kernel-Mode Threads

Thread Name	Priority	Description
Ctr_Task	kernel	Control thread. Pending on inbound message queue. Triggered by incoming control messages.
Task10	kernel	Wakes up every 10 ms, synchronously with PCM data. Execute all the DSP algorithms supported in current Intel® IXP400 DSP Software Library Release 1.1 (real-time thread).

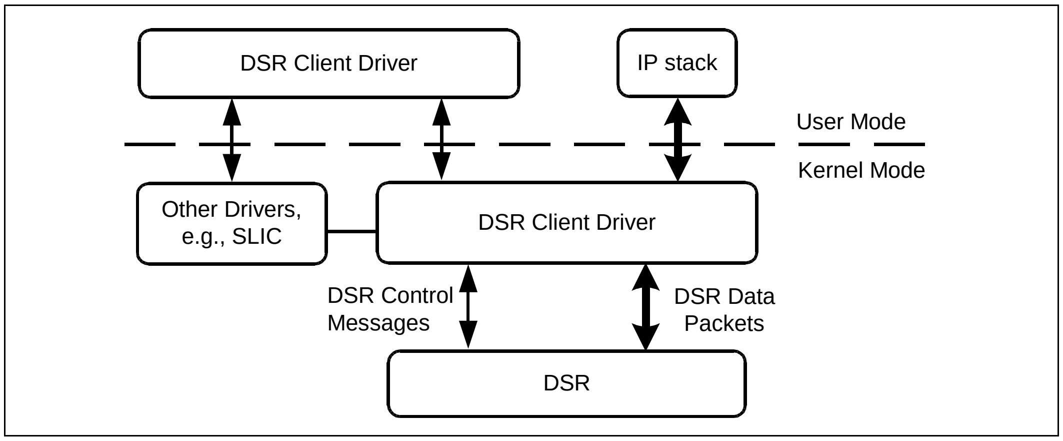
The two threads have the same priorities and do not preempt each other. To enforce real-time behavior of `Task10`, it is important that `Ctr_Task` never takes too much time in any 10-ms period. Although DSP is designed to avoid the burst execution in `Ctr_Task`, it can still be affected by the user applications. The Linux real-time extension from MontaVista* will be employed in future DSPs to enhance the real-time performance.

For the performance and reliability reasons, it is suggested that 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 a DSP client driver module as shown in Figure 8.

As the middleware, the primary responsibility of the driver module is to act as a transport layer between DSP's control interface and the user application and between DSP's packet interface and the IP stack.

The driver module's secondary responsibility is to perform DSP initialization, which can be done as part of driver-module-initialization function. Additionally, the driver may also consolidate the messages and events from DSP, SLIC, and other related modules into the same format and through a single queue to the user applications.

Figure 8. Intel® IXP400 DSP Software Library Release 1.1 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 DSP's outbound queue and automatically pumps the messages to applications, once there is a message in the queue. In passive mode, the driver retrieves the message from DSP's queue, once the user application requests it.

As discussed in "Programming Guide" beginning on page 21, the programming model for user application is still recommended. The applications should not send a burst of control messages to DSP without waiting for the replies, or the real-time behavior of DSP 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 DSP's real-time thread. As a guideline, the total execution time of these other threads should not exceed 1 ms in any 10-ms period.