# Intel® IXP400 Digital Signal Processing (DSP) Software: Voice Over Internet Protocol

## Application Note

**December 2004**

Document Number: 300320-003

# Contents



# Figures

**This page is intentionally left blank.**

# 1.0 Introduction

Intel® IXP400 DSP Software is a software module that performs voice compression, echo cancellation, tone processing, jitter control, etc., required in any IP media gateway or real-time media streaming functions. The DSP software supports G.729a, G.729.b, and G.711 speech codecs; a future release is planned to also support speech codecs G.723.1, G.723.1a, G.722, G.722.1, and G.726. In addition, the DSP software provides FSK signal generation and detection for Caller ID functions, user-defined tone generation and detection, etc.

***Note:*** Higher-layer protocols such as H.323 are not supported by the DSP software.

## 1.1 Scope

This application note is intended to explain how the DSP software components interact with each other to provide the functions required by a typical VoIP application, and how the DSP software APIs are used to implement a VoIP application with multiple threads.

Section 1.0 of this document presents DSP software architecture and features. Section 2.0 describes DSP software operation. Section 3.0 shows the programming model both in Linux and Wind River* VxWorks*. And Section 4.0 presents the detailed functions of the demo code.

Further information can be found in the *Intel® IXP400 Digital Signal Processing (DSP) Software Version 2.3 Programmer's Guide*.

## 1.2 Related Documents

| Document | Document Number |
|---|---|
| *Intel® IXP400 Digital Signal Processing (DSP) Software Version 2.3 API Reference Manual* | 273811 |
| *Intel® IXP400 Digital Signal Processing (DSP) Software Version 2.3 Programmer's Guide* | 252725 |
| *Intel® IXP400 Digital Signal Processing (DSP) Software Version 2.3 Release Notes* | N/A |

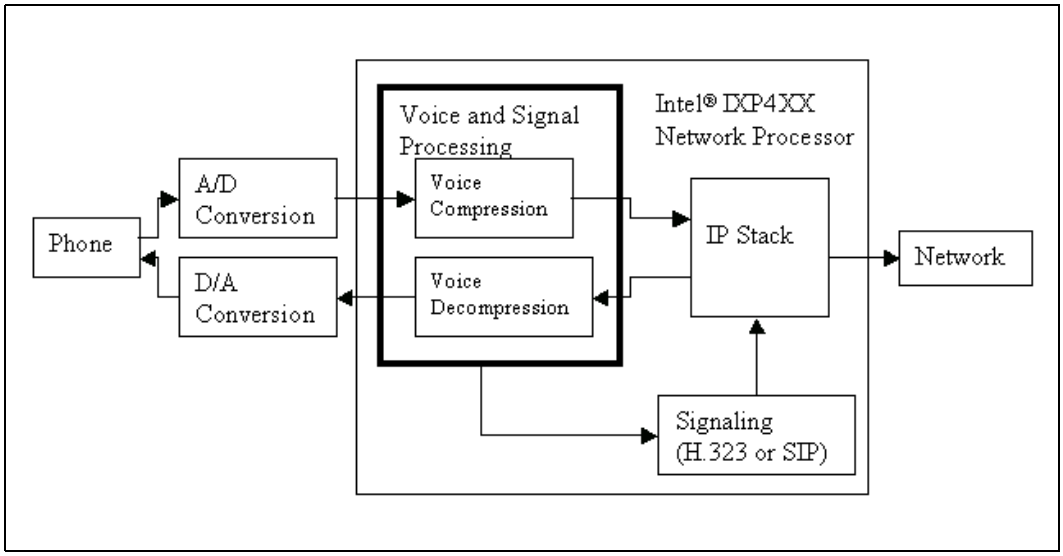
# 2.0 Intel® IXP400 DSP Software Architecture

## 2.1 Overview

A simplified block diagram for a typical voice-over-Internet-protocol (VoIP) application is illustrated in Figure 1. In this diagram, blocks of digitized voice samples from the A/D converter are provided to the voice compression module. The voice compression module compress one voice sample block once every, say, 10, 20, or 30 ms, depending on what type of speech encoder (G.729ab and G.711, G.723, etc.) is in use. The outputs of the voice compression module are blocks of binary bits, representing the voice samples. They are converted into the format of TCP/IP packets by the IP stack and sent out through the network. Similarly, TCP/IP packets received from

the IP stack will be delivered to the voice decompression module. The binary bits are converted into voice samples and sent to the D/A converter. The signaling block handles the call set-up and tear-down functions.

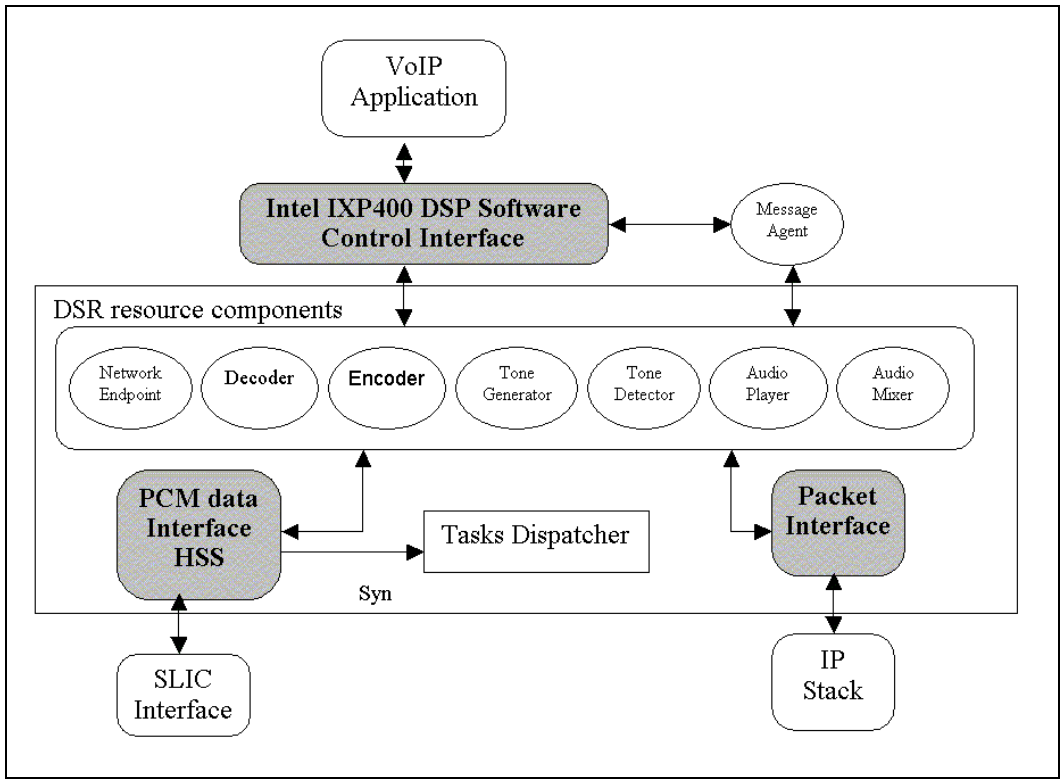
**Figure 1. Simplified VoIP Application Block Diagram**




DSP software is a software module that provides the basic voice and signal processing functionalities as highlighted with a bold frame in Figure 1. It does not cover other functions such as signaling.

As shown in Figure 2, the DSP software consists of eight resource components and three interfaces.

**Figure 2. Intel® IXP400 DSP Software Architecture**




The resource components provide the signal processing functions such as voice compression/decompression, tone generation/detection, etc. The High-Speed Serial (HSS) port provides the interface to the phone. The packet interface is for exchanging packets with the IP stack. And the VoIP application controls the resource components through the control interface. The whole operation of the DSP software is coordinated by the task dispatcher with a synchronization signal coming from the HSS port; see Section 3.0 for further operation detail. The following subsections present the functions and features of the components and interfaces.

## 2.2 Resource Components and Features

The DSP software currently supports the following features:

- G.729a and G.729b
- G.711 µ-law and A-law codec
- G.711 Annex 2. Support for VAD
- Automatic Gain Control and Automatic level control
- Support for multiple frames per packet
- Dynamically switching coder types on the fly
- Dynamically changing the number of frames per packet on the fly
- Packet loss concealment

- Dynamic/Adaptive Jitter Buffer algorithm
- Echo Cancellation (EC)
- Timers
- Flash hook detection
- TDM switch with digital gain control
- DTMF tone generation and detection
- FSK modem signals generation and detection for caller ID
- User-defined tones for tone generation and detection
- Fax tone detection (CNG, CED and V.21PRE)
- Dynamic DTMF tone clamping
- Plays back G.711 and G.729 recorded data
- Mixing multiple audio streams for 3-way call
- User-customizable control API

## 2.3 Control and Data Interface

**Control Interface**

- A set of basic messages provided for an application to control the resource components
- One in-bound message queue for messages from application
- One out-bound message queue for messages to application
- Two message functions to send and receive messages to/from to the application
- One function for application to insert user messages to the out-bound message queue
- Copy-based message delivery

**Packet Interface**

- A packet format defined for DSR to receive or deliver voice packets
- A function provided for IP stack to deliver a voice packet to DSR
- A call-back function provided by application but registered with DSR during initialization to deliver voice packet to the application
- Local time stamp provided in the packet header

**PCM DATA Interface**

- Provide interface between DSP software and the telephone interface via the TDM data bus
- Relies on the HSS hardware integrated in IXP4XX product line and IXC1100 control plane processors
- Handle PCM data in 8-bit compressed A-law or µ-law format at the rate of 8K samples per second

# 3.0 Intel® IXP400 DSP Software Operation

The DSP software module is active after it is initialized and configured (see Section 5.2). Its operation is then coordinated by the internal task dispatcher with the help of the synchronization signal from the PCM data interface.

In the transmitting path, an external A/D converter digitizes the analog signal from the phone and sends the samples in 8-bit compressed A-law or µ-law format through a timeslot in the HSS port. This continuous bit stream is buffered until when the sample buffer has accumulated a block of voice samples of 10 ms long, then a synchronization signal is sent to the DSP task dispatcher. The task dispatcher will start the Network Endpoint component of the DSP software to process the voice sample block.

The Network Endpoint component first decompresses the voice samples in 8-bit compressed A-law or µ-law format into 16-bit linear samples. Then a high pass filter with a 3-dB cut-off frequency at 270 Hz is applied to the samples to remove noise below 270 Hz. Because of the mismatch in the hybrid of the local telephone, the voice sample block from the A/D converter contains some signal coupled from the local receiving path, which, if sent back to remote telephone, will become echo. The echo canceller in the Network Endpoint component, as shown Figure 4, cancels this echo signal by delaying the signal in the local receiving path, filtering it, and subtracting it from the voice sample block from the A/D converter. The voice sample block is now ready for the speech Encoder and Tone Detector to process.

**Figure 3. PCM Data Interface Using HSS Port**

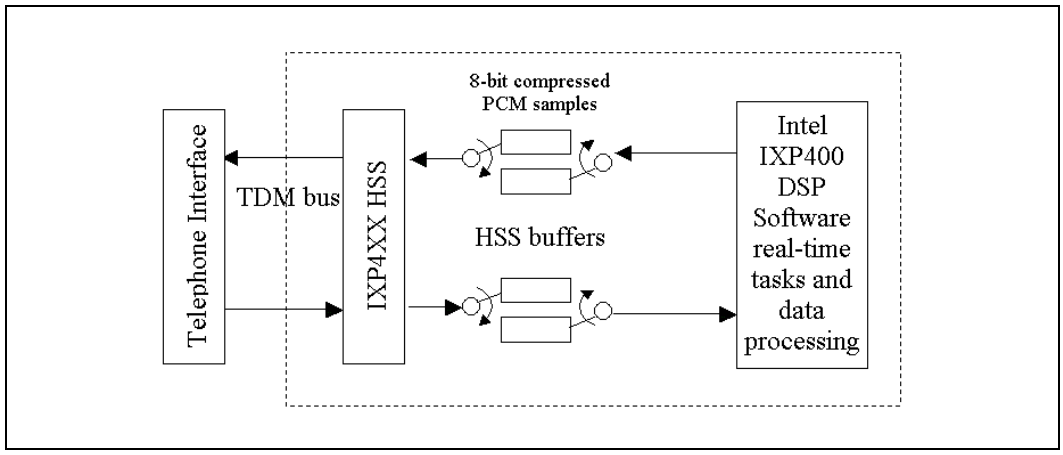
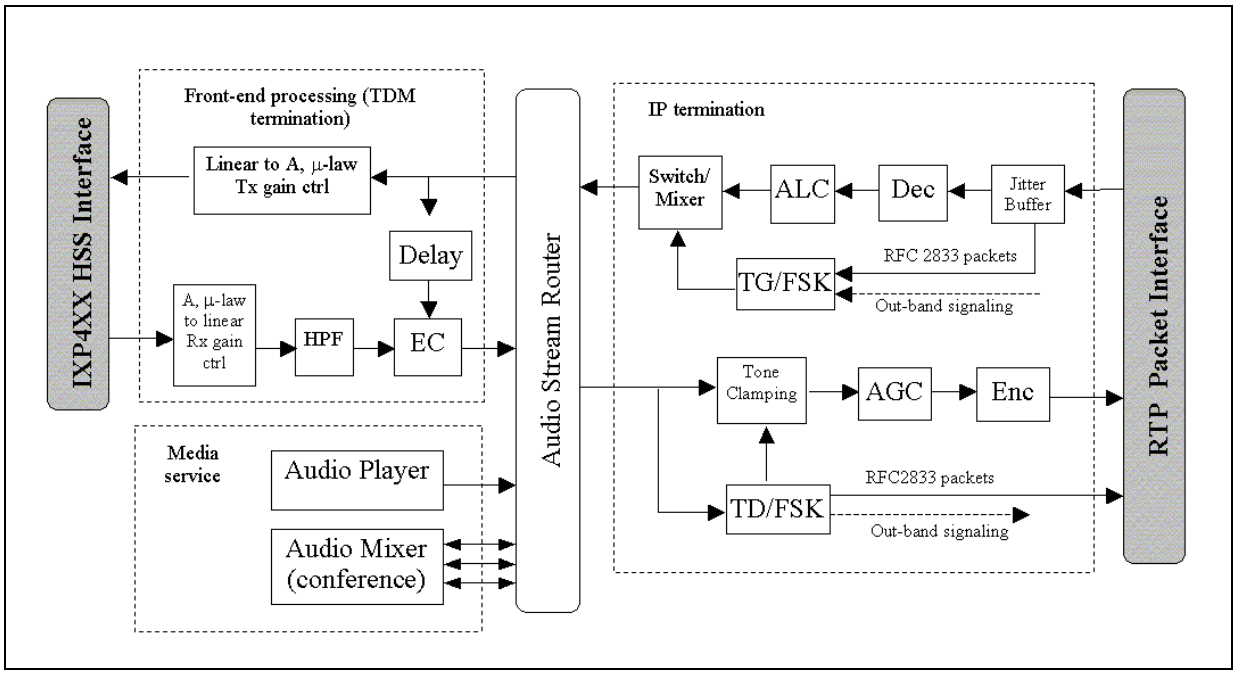


**Figure 4. Data Flow and Data Processing Functions**

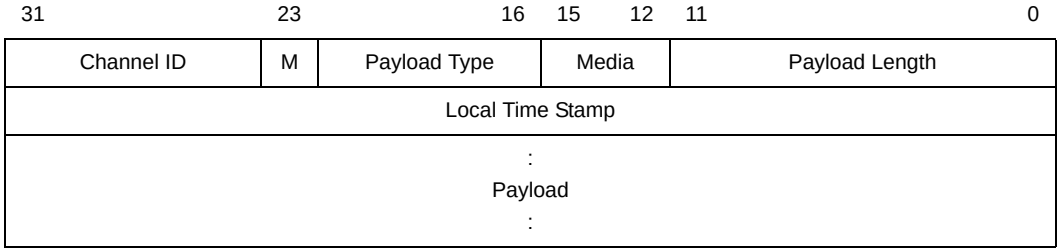



The voice sample block, after its signal level is properly adjusted by the Automatic Gain Control module (AGC), is next processed by the voice Encoder to generate audio data packet to be sent through the IP stack. The encode type is selectable, and determines the compression rate on the compressed voice.

The Tone Detector of the DSP software also uses an FFT analyzer to process the voice sample block to detect if there exist tones in the signal. If there exist predefined or user-defined tones in the signal, then tone events are reported through the out-bound message queue for the VoIP application to process. RFC2833 packets for the detected tones are also sent through the packet interface for IP stack, and the input to the voice encoder is muted (clamping) if desired.

Voice packets from the voice Encoder or RFC2833 packets from the Tone Detector employ a format, as follows. The header has a local time stamp provided by DSP software module for the RTP function in the VoIP application. The Channel ID is needed because the current DSP release can support up to four voice channels.

**Figure 5. Data Packet Format To The IP Stack**



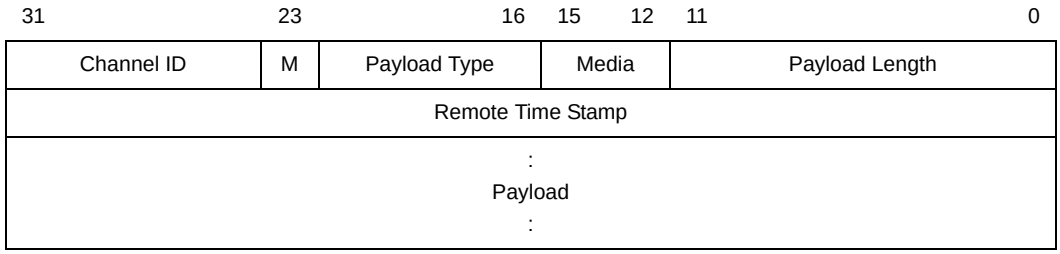
These packets are delivered to the IP stack by calling a call-back function, which is registered by the VoIP application during initialization of the DSP software. The call-back function has a prototype:

```
XStatus_t   xPacketReceive(UINT16 channel, XPacket_t *buffer);
```

where *buffer* is the pointer of the data packet for channel number *channel*. The voice Encoder may deliver one 10 ms voice frame per packet, or multiple voice frames for one packet. These DSP software packets are converted into RTP packets and transmitted though the Internet.

In the receiving path, RTP packets are received by the IP stack and converted into DSP software packet format:

**Figure 6. Data Packet Format to the Intel® IXP400 DSP Software Module**




where the Remote Time Stamp, Payload Type, and Marker bit are directly copied from a RTP packet. These packets are delivered to the Jitter Buffer in the DSP software module by calling the function xPacketReceive(), which is provided by DSP software.

The Jitter Buffer regulates the flow of packets into the voice decoder because packets could be delayed, out-of-order, duplicated, or lost without re-transmission during transmission in the IP network. This is done by dynamically delaying the packets in the Jitter Buffer or indicating that some packets are lost if they never arrive or arrive too late. Lost packets will be repaired by the Packet Loss Algorithm in the Decoder.

Voice packets from the Jitter Buffer are decoded by the voice Decoder into voice samples. The decoder may switch coder type automatically according to the received RTP payload type. The Decoder also automatically handles multiple-frames packets if a received packet contains multiple voice frames.

The Jitter Buffer directs the RFC2833 packets to the Tone Generator, which will generate the corresponding tones specified in the RFC2833 packets. The Tone Generator can also generate single- or dual-frequency tone and amplitude-modulated tone according to the control message sent by the local VoIP application. For example, the local VoIP applications may send out a sequence of control messages to DSP software module to generate Caller ID signal.

The signals from the voice Decoder and the Tone Generator are mixed according to the mode of the tone. If the tone has overwrite-mode, the speech is muted during the whole tone period. If the tone has mix-mode, the tone signal will be added to the speech so that the speech is not suppressed during the tone period. The signal from the Switch/Mixer is then delivered to the Network Endpoint component.

The Network Endpoint component will convert the 16-bit linear samples from Switch/Mixer into 8-bit compressed A-law or µ-law format and send them out through a timeslot in the HSS port. The external D/A converter will undigitize the samples and send them to the local telephone.

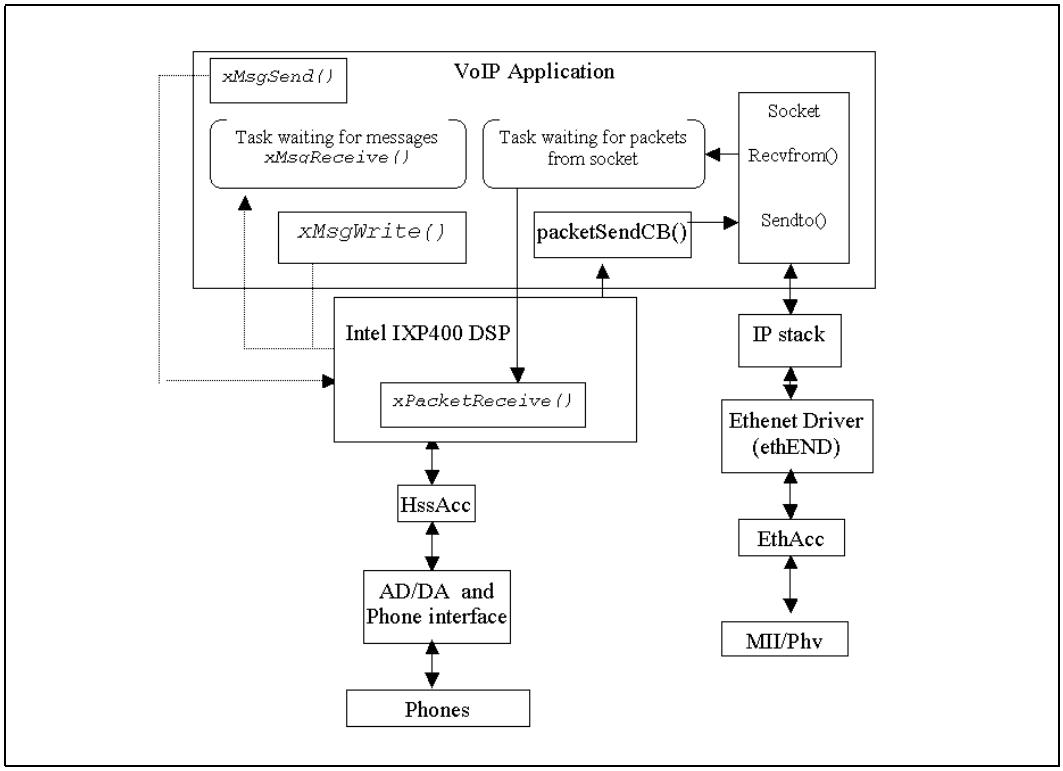
The DSP software also has the capability for conferencing with the help of the Audio Stream Router and the Audio Mixer. The Audio Mixer can add all its inputs together to form its output, and the Audio Stream Router can connect the output of one DSP software resource component to the input of another component. This is done by simply setting the input ID of one DSP software resource component equal to the ID of the output of the other component.

The DSP software has other capabilities: It can play back pre-recorded audio data to local or remote phones using the Audio Player; user-defined tones can be added to the Tone Generator or Detector; high-level, user-defined messages can be used with the help of the Message-agent to make it easier to use the DSP software basic messages to control the DSP module operations.

# 4.0 Program Module

VoIP application development using DSP software in VxWorks is different from in Linux because of the substantial difference of the OS.

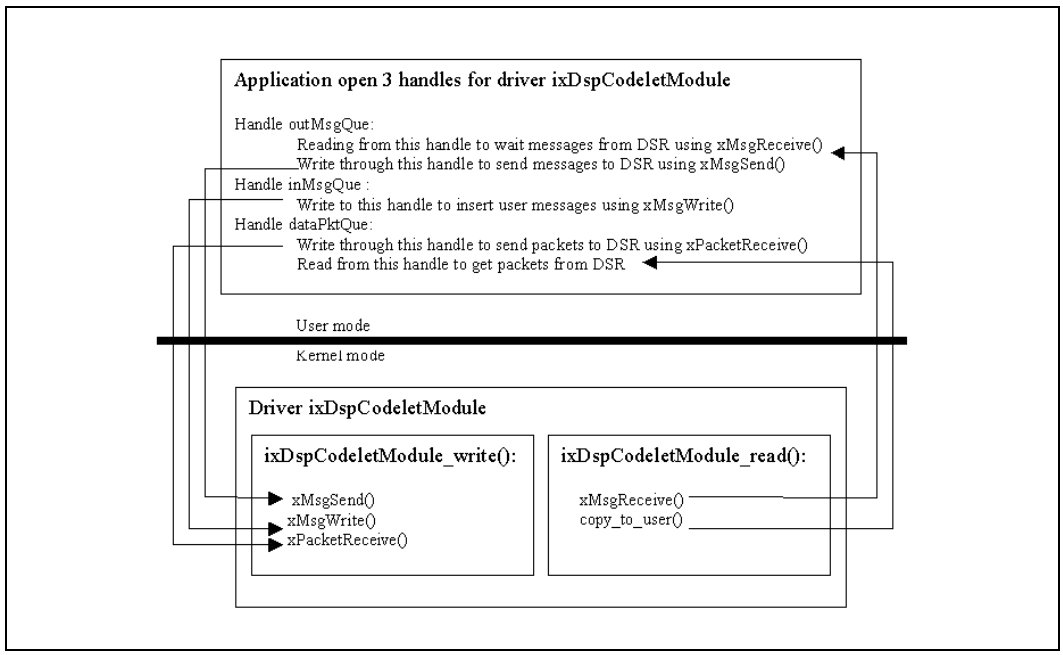
**Figure 7. Intel® IXP400 DSP Software in VxWorks***




The VoIP implementation in VxWorks is simpler because all the software modules reside under the same memory address space. Figure 7 shows how a VoIP application could be implemented in VxWorks, where application has direct access of the control and data interface. This makes it straightforward to exchange control messages and data packets between the application and DSP software. As shown in Figure 7, two tasks are spawned in the application. One uses xMsgReceive() to wait for events and response messages from DSP software module 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 down to DSP software module by calling xPacketReceive(). The DSP software module will directly call a call-back function, e.g., packetSendCB(), which is registered during initialization, to send packets through the socket.

**Figure 8. Intel® IXP400 DSP Software Application in Linux**




The case for Linux is different because the application usually runs in user mode, while the DSP software runs in kernel mode. User-mode applications cannot directly access the DSP software interfaces, which reside in kernel-mode. Hence a driver shim layer software must be developed to allow the user application to communicate with DSP software. To get the events and response messages to the application, a thread in user mode will read through the driver shim layer, which then call xMsgReceive() to wait for messages from the DSP software. Control messages are sent to DSP software module by a write function call to the driver shim layer, which will call xMsgSend() in kernel mode. Another thread in user mode waits for packets from the socket and then use driver write function call to pass the packets to the driver shim layer, which then call xPacketReceive() to pass the packet along to DSP software module. And the third thread will be waiting for packets from DSP software to send through the socket.

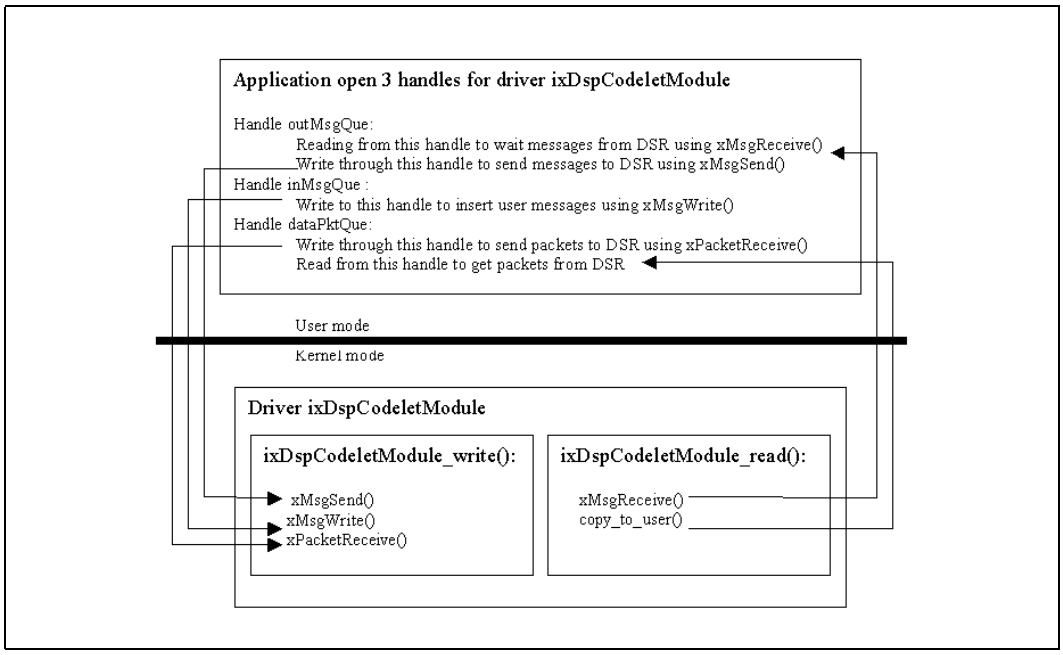
# 5.0 DSR Application Example

This section presents a simplified VoIP example intended to show how an application interfaces with the DSP software module in Linux environment. The application consists of functions both in user mode and kernel mode. A driver shim layer is created to make it easy to interface with the DSP software module. The application sends and receives messages through the reading and writing functions of the driver. Packets will be transmitted or received through sockets. It has a very simple call set-up and tear-down processes.

## 5.1 Driver Shim Layer

A driver shim layer, ixDspCodeletModule, is developed to allow the user application to communicate with DSP software, which runs in kernel mode. In this example, the application will open ixDspCodeletModule three times with three different handles: inMsgQue, outMsgQue, and dataPktQue. Their usage is shown in Figure 9. As it can be seen from the figure, sending message, waiting message, and exchanging data packets are done by reading/writing from/to the handles.

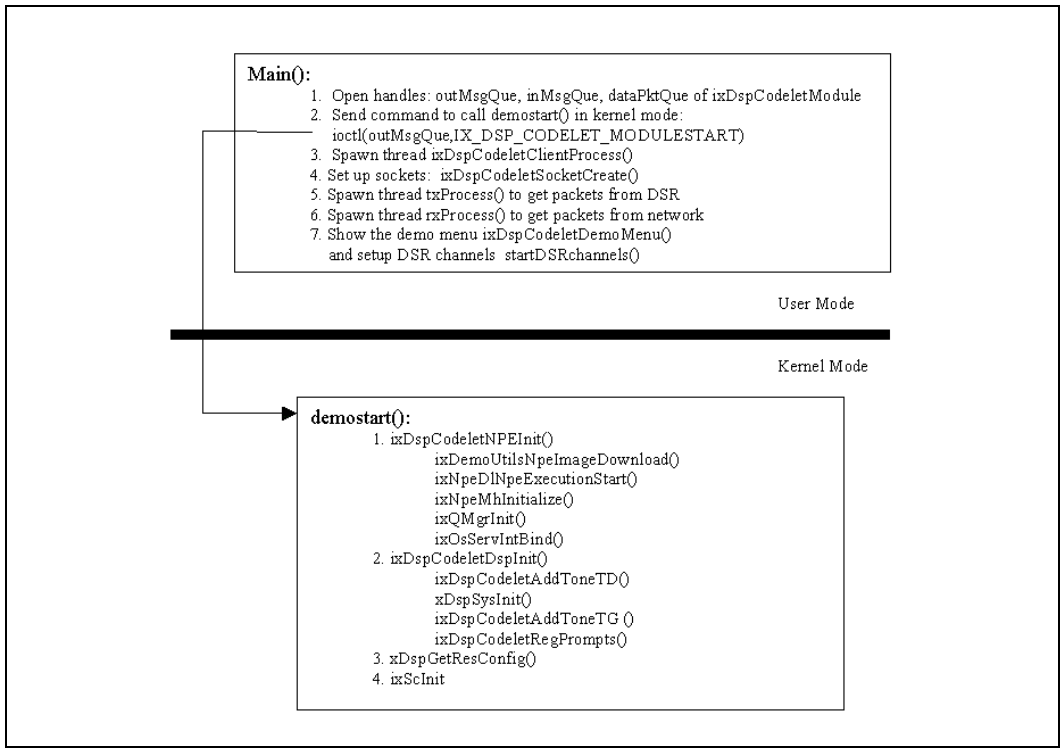
**Figure 9. Driver Shim Layer**




## 5.2 Initialization

When the application starts, main() first opens three handles for the driver ixDspCodeletModule as shown in Section 5.1. It then sends a command IX_DSP_CODELET_MODULESTART through driver handle outMsgQue to call the function demostart() in kernel mode.

**Figure 10. Initialization Steps in Demo Code**




In Kernel mode, function demostart() first calls ixDspCodeletNPEInit() to initialize NPE. ixDspCodeletNPEInit() will downloads the NPE code to NPE-A using function ixDemoUtilsNpeImageDownload(), start the NPE-A execution using ixNpeDlNpeExecutionStart(), initialize the message component using ixNpeMhInitialize(), and initialize the QMgr component using ixQMgrInit() and ixOsServIntBind() if ixp425_eth.o is not going to be used.

Function demostart() next calls ixDspCodeletDspInit() to initialize DSP module. It first calls ixDspCodeletAddToneTD() to add user-defined tone templates to Tone Detector, call xDspSysInit() with a structure XDSPSysConfig_t dspConfig to initialize and configure the DSP resource components, calls ixDspCodeletAddToneTG() to add user-defined tone templates to Tone Generator, and calls ixDspCodeletRegPrompts() to register cache prompts for the Audio Player.

Function demostart() then calls xDspGetResConfig() to get configuration information of the DSP software module.

Function demostart() finally calls ixScInit () to initializes the Slic driver and call ixScHookXCallbackRegister() to register call-back function for on/off-hook event.

After the DSR is initialized and configured, main() then spawns a thread ixDspCodeletClientProcess() to wait for messages from DSP software module. The thread ixDspCodeletClientProcess() reads through handle outMsgQue and returns only if there are messages sent back by the driver shim layer, which directly calls DSP software function xMsgReceive() to wait for messages from the DSP software components.

The main() function also calls ixDspCodeletSocketCreate() to create and set up sockets for packets transmission through the IP network. It also spawns two threads, txProcess() for packet transmitting and rxProcess() for packet receving.

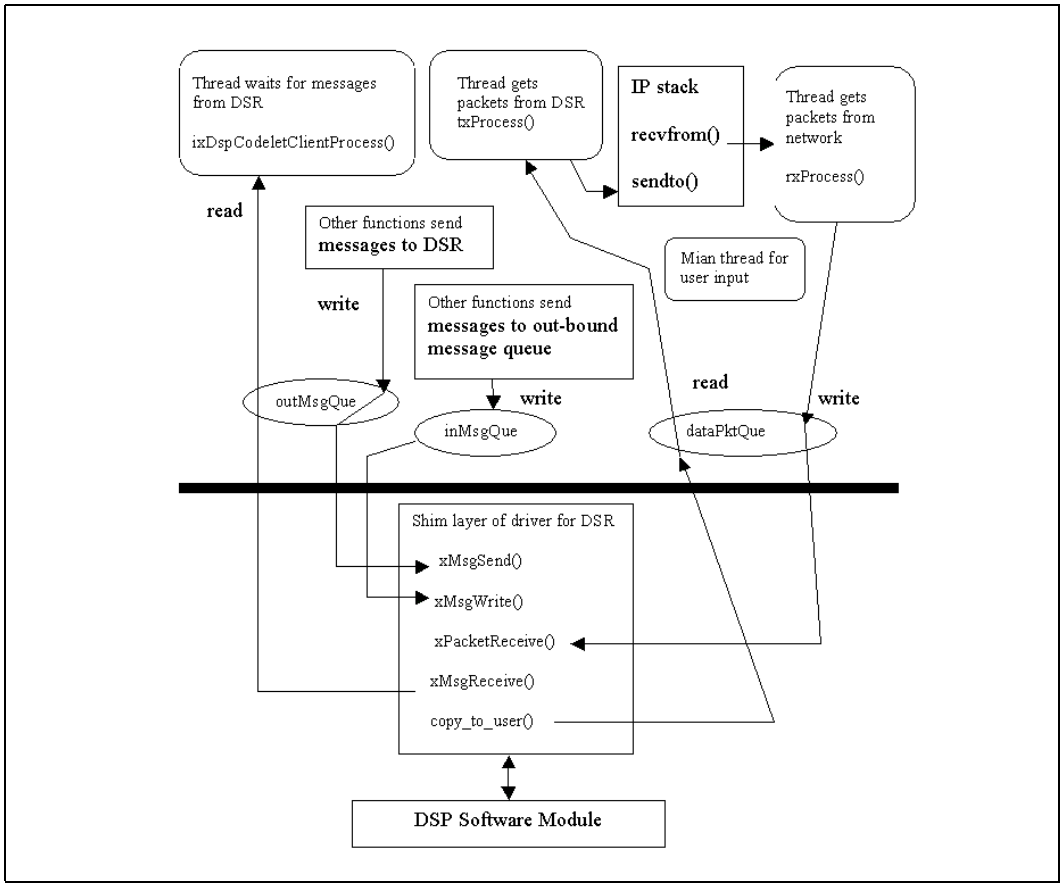
The txProcess() read from the handle dataPktQue to get packets from the DSP software module for transmission through the network. The read operation will be blocked in the driver shim layer if there is no packet from the DSP software. When there are packets coming from the DSP software, socket function sendto() is called to send the packets out after some processing.

The rxProcess() calls socket function recvfrom() to wait for packets from IP network. When packets arrive, rxProcess() will write the packets through handle dataPktQue to pass the packets to the driver shim layer, which then calls xPacketReceive() to pass the packets to the DSP software Jitter Buffer.

The main() function finally calls ixDspCodeletDemoMenu() to show the demo menu and wait for and process the user selections.

After initialization, the threads running in the demo code are shown in Figure 11.

**Figure 11. Thread Running in the Demo Code**



## 5.3 Program Flow in Call Set Up Process

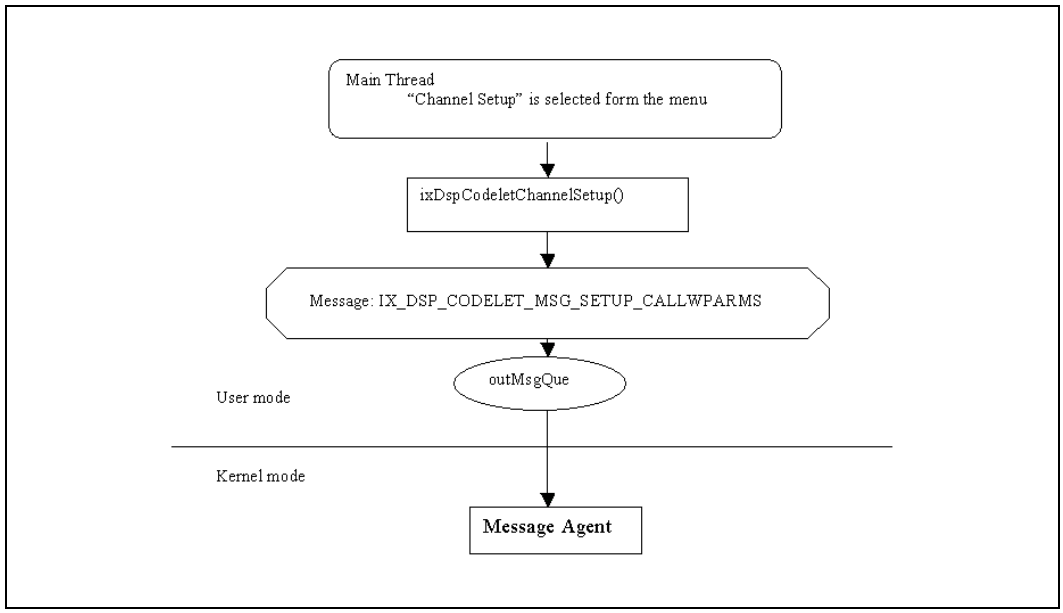
This section presents the call set up process in the original demo code in the DSP software to explain how application exchange messages with the DSP software module.

- After initialization, the main thread displays the demo menu and waits for user to make a selection.
- When "Channel Setup" is selected from the menu, array ixDspCodeletMenu[] is searched and function ixDspCodeletChannelSetup() is called.

  Because in this demo example, the thread ixDspCodeletClientProcess() process messages from DSR depending on what demo selection is made, function ixDspCodeletChannelSetup() first passes the test category value IX_DSP_CODELET_CATEGORY_TEST to the Slic driver by calling the function ixDspCodeletSetHookEventCategory() so that when the driver reports on-hook and off-hook events, the message will use this category value as part of the transaction ID. ixDspCodeletClientProcess() uses the transaction ID to decide which function is called to process the messages from DSP software module.

  For this test case, ixDspCodeletClientProcess() will simply direct the messages from the DSP software to function ixDspCodeletPrtMsg().
- Function ixDspCodeletChannelSetup() then creates a transaction ID by using the test category value IX_DSP_CODELET_CATEGORY_TEST and the channel number.
- The transaction ID is then used in IX_DSP_CODELET_MAKE_MSGHDR_SETUP_CALLWPARMS() to create a user-defined message with message type IX_DSP_CODELET_MSG_SETUP_CALLWPARMS.
- The message is then sent to the DSP software by writing to driver handler outMsgQue.

**Figure 12. Call Setup Message Sent To Message Agent**




- Because the resource type field in the message header is Message Agent XMPR_MA, the DSP software module will direct the message of type IX_DSP_CODELET_MSG_SETUP_CALLWPARMS above to the Message Agent.

- The Message Agent will call ixDspCodeletMsgDecoder(), which is registered during DSP software initialization.

  ixDspCodeletMsgDecoder() checks the message type, and uses the table msgCodeTable[] to switch to user-defined decoder function msgDecSetupCallwParms() for message of type IX_DSP_CODELET_MSG_SETUP_CALLWPARMS.

  Each time Message Agent calls ixDspCodeletMsgDecoder() for a new message, sequenceNo is set to 1. It then increases sequenceNo by 1 each time ixDspCodeletMsgDecoder() is called.

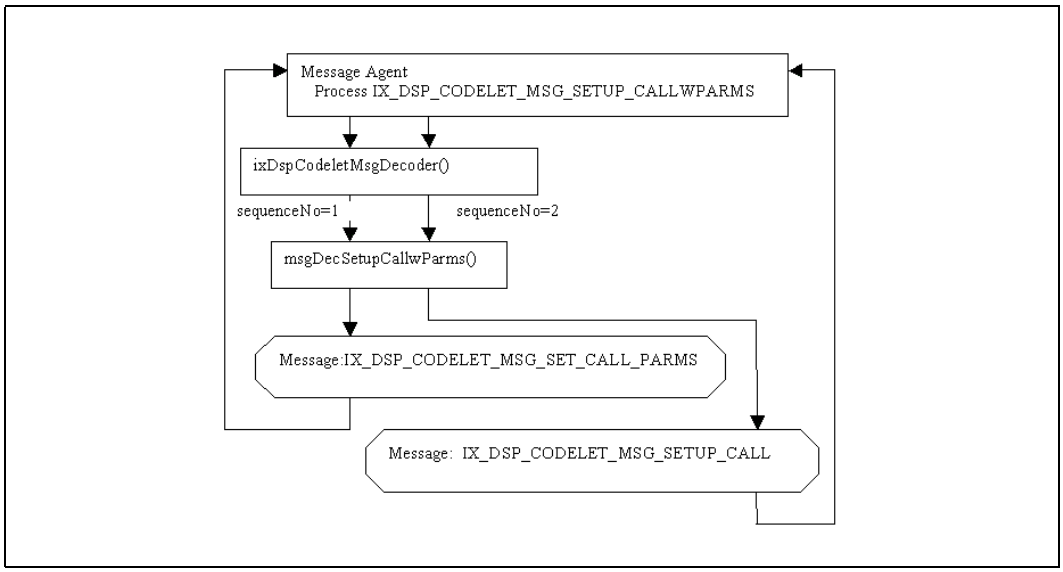
- ixDspCodeletMsgDecoder() passes sequenceNo to msgDecSetupCallwParms(). msgDecSetupCallwParms() then use sequenceNo to decide which step to take.

  In this particular case, when msgDecSetupCallwParms() is called for the first time, namely, sequenceNo=1, msgDecSetupCallwParms() will create a new user-defined message of type IX_DSP_CODELET_MSG_SET_CALL_PARMS intended for Message Agent using IX_DSP_CODELET_MAKE_MSGHDR_SET_CALL_PARMS. This message is sent to the Message Agent when msgDecSetupCallwParms() and ixDspCodeletMsgDecoder() returns.

- Because msgDecSetupCallwParms() returns a non-zero value in the step above, ixDspCodeletMsgDecoder() is called again for the original message IX_DSP_CODELET_MSG_SETUP_CALLWPARMS with sequenceNo=2. msgDecSetupCallwParms() is called again too and it will create another Message of type IX_DSP_CODELET_MSG_SETUP_CALL using IX_DSP_CODELET_MAKE_MSG_SETUP_CALL.

  This time msgDecSetupCallwParms() returns a zero value, the same as with ixDspCodeletMsgDecoder(), to indicate that it already finishes all its steps for message IX_DSP_CODELET_MSG_SETUP_CALLWPARMS. Message IX_DSP_CODELET_MAKE_MSG_SETUP_CALL is sent to the Message Agent when ixDspCodeletMsgDecoder() returns.

**Figure 13. Call Setup Decoded into Two New Macro Messages**

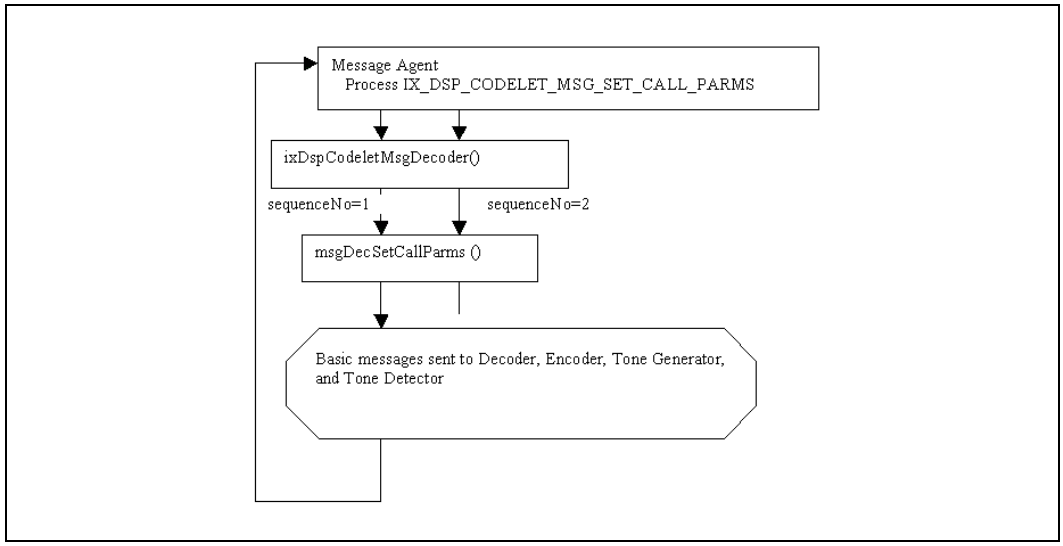



Now the Message Agent finishes processing the user-defined message IX_DSP_CODELET_MSG_SETUP_CALLWPARMS. But it now has two new user-defined messages waiting for processing, which are created during processing the message IX_DSP_CODELET_MSG_SETUP_CALLWPARMS.

- ixDspCodeletMsgDecoder() is hence called again with sequenceNo reset to 1 to process message IX_DSP_CODELET_MSG_SET_CALL_PARMS. ixDspCodeletMsgDecoder() uses the table msgCodeTable[] to switch to user-defined decoder function msgDecSetCallParms() for this message.
- Each time msgDecSetCallParms() is called, it will create a basic message for a particular DSP software resource component, and then return, causing the basic message to be sent to the particular DSP software component.

  In this particular example, msgDecSetCallParms() will be called four times to set up Decoder, Encoder, Tone Generator, and Tone Decoder. The last time msgDecSetCallParms() is called, it returns a value zero to indicate to the Message Agent that the processing of message IX_DSP_CODELET_MSG_SET_CALL_PARMS is done.

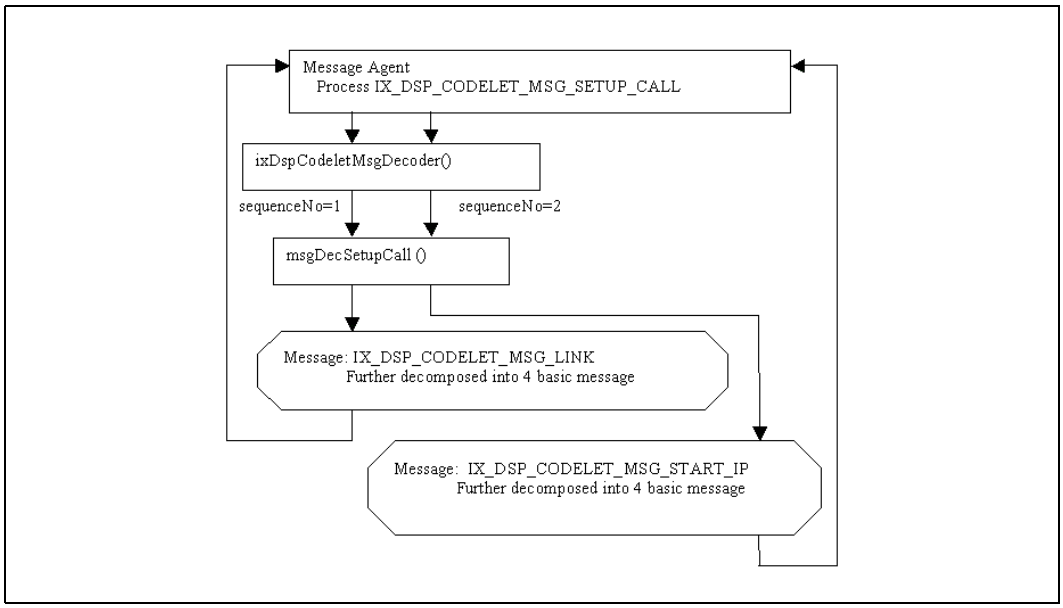
**Figure 14. Decoding Macro Message into Basic Messages For Parameters Setup**




- ixDspCodeletMsgDecoder() is called again with sequenceNo reset to 1 to process the remaining user-define message IX IX_DSP_CODELET_MSG_SETUP_CALL. The user-defined decoder function msgDecSetupCall() will be called twice for this case. It will create 2 more new user-defined macro messages IX_DSP_CODELET_MSG_LINK and IX_DSP_CODELET_MSG_START_IP.
- Message IX_DSP_CODELET_MSG_LINK will cause msgDecLink() to be called. This user-defined message is decoded into up to four basic massages in decoder function msgDecLink() to connect L-ports (the input port of components) with T-ports (the output ports of components).
- Message IX_DSP_CODELET_MSG_START_IP causes msgDecStartStopIP()to be called. This message will be decoded into 0 ~ 4 DSP messages that start the ENC, DEC and TD respectively. And TG is always stopped.

The call-setup process is now completed.

**Figure 15. Decoding Macro Message into Basic Messages For Link Setup**




## 5.3.1 Program Flow in Call Setup Using Sockets

This section presents the very simple call-setup process in the demo code modified for using sockets.

After the initialization, the main thread displays the demo menu and waits for user to make a selection. When "IP Call Menu" is selected from the menu, array ixDspCodeletMenu[] is searched and function IPcallMenu () is called.

IPcallMenu () first sets IPcallTestFlag to 1 so that ixDspCodeletClientProcess() will direct all messages from the DSP software module to function ixDspCodeletIPcallSM(), which is a very simple state machine to handle the call process.

IPcallMenu() then passes the test category value IX_DSP_CODELET_IPCALL_TEST to the Slic driver so that when the driver reports on-hook and off-hook events, the message will use this category value as part of the transaction ID.

IPcallMenu() also initialize the array ixDspCodeletDemoChl[] to indicate that the phones are in on-hook state.

When a phone is picked up, the Slic codec driver will call the call-back function ixDspCodeletSlicHookCB() to inform that the phone is off-hook. ixDspCodeletSlicHookCB() then calls the DSP software complementary function xFlashHookDetect() to report an off-hook event XEVT_NET_HOOK_STATE through the out-bound message queue. This event is directed by ixDspCodeletClientProcess() to function ixDspCodeletIPcallSM().

When ixDspCodeletIPcallSM() finds out it is an off-hook event IX_SC_OFF_HOOK, it calls playOneTone() to send a message to the Tone Generator to play a dial tone in the phone. The phone state in ixDspCodeletDemoChl[] is updated to state POTS_OFFHOOK, and the calling number variable is cleaned.

When the user presses a key in the phone keypad, the Tone Detector will send out a XEVT_CODE_TD_TONEON event. ixDspCodeletIPcallSM() calls stopPlayingTone() to send a message to the Tone Generator to stop playing the dial tone. The number pressed in the keypad is stored in the calling number variable in ixDspCodeletDemoChl[], and the phone state in ixDspCodeletDemoChl[] is updated into state POTS_ACTIVE.

When the # key is entered form the keypad, the calling number entered is checked. Depending on what number is input, the remote IP address is obtained from the number entered or from the directory. If it is a valid remote IP address, the function ixDspCodeletSendMeg() is called to send out an IP packet through the IP network to the remote phone to request a connection RPC_IP_CONNECT_REQ. The packet include information of the local phone IP address, the channel number of the local phone, the remote phone IP address, the desired channel of the remote phone. The local phone is now waiting for voice packets from the remote phone.

When the remote phone receives the connection request packet RPC_IP_CONNECT_REQ, the IP address and the channel number are stored, and function ixDspCodeletRingReq() is called to send a XEVT_USER_RING_REQ event into the out-bound message queue. ixDspCodeletIPcallSM() call ixDspCodeletNormalRing() to make the remote phone ring. The remote phone state is set to POTS_RINGING.

When the remote phone is picked up, ixDspCodeletIPcallSM() calls ixDspCodeletStopRing() to stop the ringing, the remote phone state is set to POTS_ACTIVE.

The call is now set up completely.

## 5.4 Running the Demo Code

Procedures to build and run the demo code in the DSP software are presented in the *Intel® IXP400 Digital Signal Processing (DSP) Software Version 2.3 Release Notes*. The demo can run on the Intel® IXDP425 / IXCDP1100 Development Platform and the Coyote* Gateway Reference Design. Use the following steps to run the demo code:

1. Load the kernel

   load -r -v -b 0x01600000 zImage

2. Run the kernel

   exec 0x01600000

3. Login as root

   root

4. Copy the following files to the NFS directory:

   dsr.o
   IxDspCodeletApp
   csr.o
   csr_codelets_demoUtils.o
   csr_codelets_dspEng.o

5. Execute the following commands:

   insmod csr.o
   insmod csr_codelets_demoUtils.o
   insmod dsr.o
   insmod csr_codelets_dspEng.o
   mknod /dev/ixDspCodeletModule c 253 0
   ./IxDspCodeletApp

6. Make the following selections:

   Choose test plt – (1:Richfield, 2:Coyote) – 2
   Set country code (1:US, 81:Japan, 86:China) – 1
   Please select test item – 1 (1 for channel set up)
   Enter Channel Number – 0
   Enter the coder type – 3

7. Phones can now be picked up to have a conversation.

If the demo code using sockets is used, then the following procedure should be followed:

1. Load the kernel

   load -r -v -b 0x01600000 zImage

2. Run the kernel

   exec 0x01600000

3. Login as root

   root

4. Copy the following files to the NFS directory:

   dsr.o
   IxDspCodeletApp
   csr.o
   insmod ixp425_eth.o
   csr_codelets_demoUtils.o
   csr_codelets_dspEng.o

5. Execute the following commands:

   cd /tmp
   insmod csr.o
   insmod ixp425_eth.o
   insmod csr_codelets_demoUtils.o
   insmod dsr.o
   insmod csr_codelets_dspEng.o
   mknod /dev/ixDspCodeletModule c 253 0
   ./IxDspCodeletApp

6. Make the following selections:

   Choose test plt – (1:Richfield, 2:Coyote) – 2
   Set country code (1:US, 81:Japan, 86:China) – 1
   Please select test item – 1 (12 for IP Call Menu)

7. Connect IXP1 (J4 on the Coyote board) to a hub if call is to be made from one board to another.

8. Set up the IXP1 IP address, e.g.:

   Set one board: ifconfig ixp1 18.17.17.111
   Set another board: ifconfig ixp1 18.17.17.112

9. Pick up a phone with 18.17.17.111, and then:

   Dial 18*17*17*112*0# if the phone is connected to J1 on the board
   Or dial: 1120# (packets will go through PCI Ethernet card)
   Dial 18*17*17*112*1# if the phone is connected to J2 on the board
   Or dial: 1121# (packets will go through PCI Ethernet card)

10. Answer another phone to have a conversation.

# Appendix A Example Code Using Sockets

```
IX_STATUS ixDspCodeletSocketCreate(void)
{
        struct sockaddr_in sockAddrIn;

        /* Open the Rx and Tx sockets and save the descripters */
        rxSocketDescripter = socket(AF_INET, SOCK_DGRAM, 0);
        if (rxSocketDescripter == ERROR)
        {
           printf("%s: ERROR - socket open failed for DSP data reception.\n",__FUNCTION__);
           return IX_FAIL;
        }

        txSocketDescripter = socket(AF_INET, SOCK_DGRAM, 0);
        if (txSocketDescripter == ERROR)
        {
           printf("%s: ERROR - socket open failed for DSP data transmission.\n", __FUNCTION__);
           return IX_FAIL;
        }

        localUdpPort = htons(COMMON_PORT);
        remoteUdpPort = htons(COMMON_PORT);

        bzero((char *)&sockAddrIn, sizeof(sockAddrIn));
        sockAddrIn.sin_family = AF_INET;
        //sockAddrIn.sin_len = sizeof(struct sockaddr_in);
        sockAddrIn.sin_addr.s_addr = INADDR_ANY;
        sockAddrIn.sin_port = localUdpPort;

        /* Only need to bind address to the receive socket */
        if (bind(rxSocketDescripter, (struct sockaddr *)(&sockAddrIn),sizeof(sockAddrIn) ) == ERROR)
        {
           printf("%s: ERROR - receive socket bind failed.\n", __FUNCTION__);
           return IX_FAIL;
        }

        return IX_SUCCESS;
}



int ixDspCodeletIPcallSM(XMsgRef_t pMsg)
{
   int chl;
   IxDspCodeletChannel *pChl;
   UINT32 code, data1, data2;

   chl = IX_DSP_CODELET_TRANS_GET_CHAN(pMsg);
   pChl = ixDspCodeletDemoChl + chl;

   printf("\n------DSR msg for station %d\n",chl-1);
   switch (pChl->state)
   {
        case POTS_ONHOOK:
                printf("curretn state: POTS_ONHOOK\n");
                break;
```

```
        case POTS_OFFHOOK:
                printf("curretn state: POTS_OFFHOOK\n");
                break;
        case POTS_RINGING:
                printf("curretn state: POTS_RINGING\n");
                break;
        case POTS_ACTIVE:
                printf("curretn state: POTS_ACTIVE\n");
                break;
}


switch(pMsg->type)
{
        case XMSG_EVENT:
                printf("get an event \n");
                XMSG_FIELD_EVENT(pMsg, code, data1, data2);
                switch (code)
                {
                        case XEVT_NET_HOOK_STATE:
                                printf("XEVT_SLIC_HOOK \n");
                                if(data1==IX_SC_OFF_HOOK)
                                {
                                        printf("IX_SC_OFF_HOOK \n");
                                        if(pChl->state==POTS_ONHOOK)
                                        {
                                                playOneTone(chl, NTT_TID_DT);
                                                pChl->state=POTS_OFFHOOK;
                                           printf("new state: POTS_OFFHOOK\n");
                                                pChl->fpp=0;
                                                memset(pChl->cidData,0,20);
                                        printf("phone %d (%d) is picked up\n",chl-1,
                                xDspCodeletSocketChanConfigInfo[chl-1].localPhoneNumber);
                                        }
                                        else if(pChl->state==POTS_RINGING)
                                        {
                                                ixDspCodeletStopRing(chl);
                                                pChl->state=POTS_ACTIVE;
                                           printf("new state: POTS_ACTIVE\n");
                                                printf("phone %d (%d) is picked up\n",chl-1,
                                ixDspCodeletSocketChanConfigInfo[chl-1].localPhoneNumber);
                                        }
                                }
                                else if(data1==IX_SC_ON_HOOK)
                                {
                                        stopPlayingTone(chl);
                                        ixDspCodeletSendMeg(RPC_RELEASE, NULL, 0, chl-1);

                                        pChl->state=POTS_ONHOOK;
                                   printf("new state: POTS_ONHOOK\n");
                                        pChl->fpp=0;
                                        memset(pChl->cidData,0,20);
                                        printf("phone %d (%d) hang up\n",chl-1,
                                ixDspCodeletSocketChanConfigInfo[chl-1].localPhoneNumber);
                                }
                        break;

                        case XEVT_CODE_TD_TONEON:
                                printf("XEVT_CODE_TD_TONEON \n");
                        if( (pChl->state==POTS_OFFHOOK)||(pChl->state==POTS_ACTIVE) )
```

```
                {
                        stopPlayingTone(chl);

                        pChl->state=POTS_ACTIVE;
                        printf("new state: POTS_ACTIVE\n");
                        if((RFC_TID_DTMF_0<=data1)&&(data1<=RFC_TID_DTMF_9))
                        {
                                pChl->cidData[pChl->fpp]='0'+data1;
                                pChl->fpp=pChl->fpp+1;
                                printf("dialed number: %s\n",pChl->cidData);
                        }
                        else if (data1==RFC_TID_DTMF_STAR)
                        {
                                pChl->cidData[pChl->fpp]='.';
                                pChl->fpp=pChl->fpp+1;
                                printf("dialed number: %s\n",pChl->cidData);
                        }
                        else if (data1==RFC_TID_DTMF_POUND)
                        {
                        printf("# is entered\n");
                        convertDialedNumber(chl, pChl->cidData, pChl->fpp);

                                pChl->fpp=0;
                                memset(pChl->cidData,0,20);
                                showCallSetup();
                        }
                }
break;

case XEVT_USER_RING_REQ:
                printf("XEVT_USER_RING_REQ \n");
                if(pChl->state==POTS_ONHOOK)
                {
                   printf("make local phone ring \n");
                        ixDspCodeletNormalRing(chl);
                        pChl->state=POTS_RINGING;
                   printf("new state: POTS_RINGING\n");
                }
break;

case RPC_RELEASE:
                printf("RPC_RELEASE \n");
                if(pChl->state==POTS_RINGING)
                {
                        ixDspCodeletStopRing(chl);
                        pChl->state=POTS_ONHOOK;
                   printf("new state: POTS_ONHOOK\n");
                }
                else if (pChl->state==POTS_ACTIVE)
                {
                                playOneTone(chl, NTT_TID_DT);
                                pChl->state=POTS_OFFHOOK;
                     printf("new state: POTS_OFFHOOK\n");
                                pChl->fpp=0;
                                memset(pChl->cidData,0,20);
printf("call to phone %d (%d) is over.  Dial new number\n",chl-1,
ixDspCodeletSocketChanConfigInfo[chl-1].localPhoneNumber);

                }
break;
```

```
                }
        break;
    }
    return IX_SUCCESS;
}


void playOneTone(int dspChannel, UINT8 tone)
{
        XMsgTGPlay_t tGenMsg;
        UINT8 *toneId;
        INT32 trans;

    trans = IX_DSP_CODELET_MAKE_TRANS(IX_DSP_CODELET_IPCALL_TEST, dspChannel);

        XMSG_FIELD_TG_PLAY(&tGenMsg, toneId);
        toneId[0] = tone;
        XMSG_MAKE_TG_PLAY(&tGenMsg, trans, dspChannel, 1);
        xMsgSend(&tGenMsg);
}

void stopPlayingTone(int dspChannel)
{
        XMsgStop_t stopMsg;
    UINT32 trans;

    trans = IX_DSP_CODELET_MAKE_TRANS(IX_DSP_CODELET_IPCALL_TEST, dspChannel);

        XMSG_MAKE_STOP(&stopMsg, trans, XMPR_TNGEN, dspChannel);
        xMsgSend(&stopMsg);
}


IX_STATUS ixDspCodeletSocketReceive(void)
{
    struct sockaddr_in remoteSockAddr;
    char *buffer;
    int maxRead;

    int receivedBytes = -1;
    int remoteSockAddrSize = 0; /* This variable has two purposes, to
                                                        * indicate size of remoteSockAddr to the
                                                        * recvfrom function, and to store the size
                                                        * of the remoteSockAddr returned from the
                                                        * recvfrom function
                                                        */

    /* creat a buffer to receive data from the socket */
    maxRead = IX_DSP_CODELET_SOCKET_MAX_READ_FROM_SOCKET_SIZE;
    buffer = malloc (maxRead);

    /* will keep waiting */
    while(1)
    {
        remoteSockAddrSize = sizeof(remoteSockAddr);

        receivedBytes =     recvfrom(rxSocketDescripter,buffer, maxRead,
                            0,     /* Flags */
                            (struct sockaddr *)&remoteSockAddr,
                                    &remoteSockAddrSize);
```

```
        IX_ASSERT(receivedBytes < maxRead);

        if (receivedBytes == ERROR)
        {
                printf("error socket Rx\n");
                continue;
        }

        ixDspCodeletSocketProcessRxPkt(buffer,receivedBytes, &remoteSockAddr);
    }

    free (buffer);
    return IX_SUCCESS;
}

XStatus_t ixDspCodeletSocketSend(char *buffer, int size, unsigned ipAddr, UINT16 port)
{
    struct sockaddr_in remoteSockAddr;
    unsigned bytesTransmitted = 0;

    bzero((char *)&remoteSockAddr, sizeof(remoteSockAddr));
    remoteSockAddr.sin_family = AF_INET;
    //remoteSockAddr.sin_len = sizeof(remoteSockAddr);
    remoteSockAddr.sin_addr.s_addr = ipAddr;
    remoteSockAddr.sin_port = port;

    /* Send the data to the socket */
    bytesTransmitted =      sendto(txSocketDescripter,
                                   buffer,
                                   size,
                                   0, /* flags */
                                   (struct sockaddr *)(&remoteSockAddr),
                                   sizeof(remoteSockAddr) );

    if (bytesTransmitted == ERROR)
    {
        printf("socket Tx error 1\n");
        return XERROR;
    }
    else if (bytesTransmitted != size)
    {
        printf("socket Tx error 2\n");
        return XERROR;
    }

    return XSUCC;
}

void IPcallMenu()
{
    int i;
    int selectedItem=0;

    // for testing without the call state machine
    //startDSRchannels();
    //return;

    IPcallTestFlag=1;
```

```
    ixDspCodeletSetHookEventCategory(IX_DSP_CODELET_IPCALL_TEST);

    // initial channel state
    //for(i=0; i<=ixDspCodeletNumHssChannels; i++)
    for(i=0; i<=IX_DSR_MAX_CHANNELS; i++)
    {
            ixDspCodeletDemoChl[i].state=POTS_ONHOOK;
            ixDspCodeletDemoChl[i].fpp=0;
    }



    do
    {
      /* print the test menu */
      {
          printf( "\n----------------------------------------\n"
                  "-      IP Phone Test Menu          -\n"
                  "----------------------------------------\n");

          printf("1 - %s\n", "setup a call");
          printf("2 - %s\n", "chat");
          printf("3 - %s\n", "get current call setup");
          printf("4 - %s\n", "setup local IP address");
          printf("5 - %s\n", "exit to main menu");

          printf("Please select test item - ");
       }
       selectedItem = ixDspCodeletGetNum(NULL);

          switch (selectedItem)
          {
                      case 1:
                                  setupCallingIPaddr();
                      break;

                      case 2:
                                  chatProcess();
                      break;

                      case 3:
                                  showCallSetup();
                      break;

                      case 4:
                                  setupLocalIP();
                      break;
          }
    }while(selectedItem != 5);

    IPcallTestFlag=0;
}

void ixDspCodeletRingReq(int chl)
{
    XMsgEvent_t  evt;
    UINT32 trans;

    printf("send a command to ring the local phone %d\n",chl-1);
    /* create a user-defined event */
```

```
    trans = IX_DSP_CODELET_MAKE_TRANS(IX_DSP_CODELET_IPCALL_TEST, chl);
    XMSG_MAKE_HEAD(&evt,
                        trans,
                        XMPR_USER,
                        chl,
                        sizeof(XMsgEvent_t),
                        XMSG_EVENT,
                        0);

    evt.code = XEVT_USER_RING_REQ;

    /* send the user-defined event via DSR's message queue */
    //xMsgWrite(&evt);
    ixDspCodeletIPcallSM(&evt);
}

void ixDspCodeletCallRelease(int chl)
{
    XMsgEvent_t evt;
    UINT32 trans;

    printf("send a command to release the local phone %d\n",chl-1);
    /* create a user-defined event */

    trans = IX_DSP_CODELET_MAKE_TRANS(IX_DSP_CODELET_IPCALL_TEST, chl);
    XMSG_MAKE_HEAD(&evt,
                        trans,
                        XMPR_USER,
                        chl,
                        sizeof(XMsgEvent_t),
                        XMSG_EVENT,
                        0);

    evt.code = RPC_RELEASE;

    /* send the user-defined event via DSR's message queue */
    //xMsgWrite(&evt);

    ixDspCodeletIPcallSM(&evt);

}
```

# Appendix B Example Driver Code

```
ssize_t ixDspCodeletModule_write(struct file *fp, const char *buf, size_t num, loff_t *off)
{
   int rc;

   //modified for using socket-->
   XPacket_t *p;
   //<--modified for using socket

   switch ((int)fp->private_data)
   {
   case IX_DSP_CODELET_SETOUTMSGQUEUE:
      rc = xMsgSend((void *)buf);
      if(rc!=XSUCC)
      {
         printf("ERROR - xMsgSend() fails.\n");
         return -ESPIPE;
      }
      break;
   case IX_DSP_CODELET_SETINMSGQUEUE:
      rc = xMsgWrite((void *)buf);
      if(rc!=XSUCC)
      {
         printf("ERROR - xMsgSend() fails.\n");
         return -ESPIPE;
      }
      break;

   case IX_DSP_CODELET_SETPLAYFILE:
                  num =  ixDspCodeletPlayPrepareSendData(fp, (const char *)buf,num,1);
                  break;

   //modified for using socket-->
   case IX_DSP_CODELET_DATA_PACKET:
         if (buf == NULL)
         {
            printk("KERNEL warning DSP WRITE received invalid parameters\n");
            return -ESPIPE;
         }

         if (copy_from_user(dspRxBuffer, buf, num))
         {
                  printk("KERNEL warning DSP WRITE couldn't access user space\n");
         }
         else
         {
                  p=(XPacket_t *) buf;
                  xPacketReceive(p->channelID, p);
         }
         break;
         //<--modified for using socket

   default:
      return -ESPIPE;
   }

   return(num);
```

```
}

ssize_t ixDspCodeletModule_read(struct file *file, char *buffer, size_t size, loff_t *ppos)
{
   //modified for using socket-->
   XPacket_t *p;
   //<--modified for using socket

   size_t length = 0;

   switch ((int)file->private_data)
   {
   case IX_DSP_CODELET_SETOUTMSGQUEUE:
      length = xMsgReceive((XMsgRef_t)buffer, 0,  XWAIT_FOREVER );
      break;

   case IX_DSP_CODELET_SETINMSGQUEUE:
      length = xMsgRead((XMsgRef_t)buffer,  XWAIT_FOREVER );
      break;

   // modified for using socket-->
   case IX_DSP_CODELET_DATA_PACKET:
         while (dspTxBuffer_pool<=0)
         {
                  MSG("putting process with pid %u to sleep\n", current->pid);
                  /* go to sleep, but wake up on signals */
                  interruptible_sleep_on(&schar_wq);
                  if (signal_pending(current))
                  {
                           MSG("pid %u got signal\n", (unsigned)current->pid);
                           /* tell vfs about the signal */
                           return -EINTR;
                  }
         }

         /* copy the data from the buffer */
         p=(XPacket_t *)(dspTxBuffer+buf_out_ind*BUFFER_LENGTH);
         length=p->payloadLen+sizeof(XPacketHeader_t);

         if (copy_to_user(buffer, p, length))
                  return -EFAULT;

         buf_out_ind=(buf_out_ind+1)&(NUMBER_OF_BUFFER-1);

         dspTxBuffer_pool -= length;

         MSG("read %u bytes, %ld bytes in queue\n", (unsigned)length, dspTxBuffer_pool);
         MSG("buf_out_ind= %u\n",(unsigned)buf_out_ind);

         break;
         //<--modified for using socket

   default:
      break;
   }

   return(length);
}

int ixDspCodeletModule_ioctl(struct inode *inode, struct file *file,
```

```
                    unsigned int functionId, unsigned long arg)
{
   int rc;
   IxDspCodeletReadParm *readParm;
   IxDspCodeletWriteParm *writeParm;

   switch(functionId)
   {
   case IX_DSP_CODELET_MODULESTART: {
         int CntCd = (arg >> 16);
         TstPlatform = (arg&3);
      demostart(TstPlatform,CntCd);
      break;
   }

   case IX_DSP_CODELET_SETOUTMSGQUEUE:
      (int)file->private_data =  IX_DSP_CODELET_SETOUTMSGQUEUE;
      xDspGetResConfig((void *)arg);
      break;

   case IX_DSP_CODELET_SETINMSGQUEUE:
      (int)file->private_data = IX_DSP_CODELET_SETINMSGQUEUE;
      break;

   case IX_DSP_CODELET_SETPLAYFILE:
         (int)file->private_data = IX_DSP_CODELET_SETPLAYFILE;
         ixDspCodeletPlayPrepareSendData(file, (const char *)arg, sizeof(XCachePromptDesc_t),0);
         break;

   case IX_DSP_CODELET_READPARM:
      readParm = (IxDspCodeletReadParm *)arg;
      readParm->result = xDspParmRead(readParm->res, readParm->inst,
            readParm->parmId, readParm->pParmVal);
      break;

   case IX_DSP_CODELET_WRITEPARM:
      writeParm = (IxDspCodeletWriteParm *)arg;
      writeParm->result = xDspParmWrite(writeParm->res, writeParm->inst,
         writeParm->parmId, writeParm->parmVal, writeParm->trans);
      break;

   // modified for using socket
   case IX_DSP_CODELET_DATA_PACKET:
         printk("set up IX_DSP_CODELET_DATA_PACKET\n");

         (int)file->private_data = IX_DSP_CODELET_DATA_PACKET;
         dspTxBuffer = (char *) kmalloc(BUFFER_LENGTH*NUMBER_OF_BUFFER,GFP_KERNEL);
         dspRxBuffer = (char *) kmalloc(sizeof (XPacket_t),GFP_KERNEL);

         if(dspTxBuffer==NULL)
                  printk("dspTxBuffer is NULL\n");
         else
                  printk("dspTxBuffer is allocated\n");

         if(dspRxBuffer==NULL)
                  printk("dspRxBuffer is NULL\n");
         else
                  printk("dspRxBuffer is allocated\n");

         dspTxBuffer_pool = 0;
```

```
        buf_in_ind = buf_out_ind=0;

        printk("done setting up IX_DSP_CODELET_DATA_PACKET\n");

        break;
        //<--modified for using socket

/* ..........................................................*/
/* These are JP Caller id ioctls.....................*/

    case IX_DSP_CODELET_GETHOOKSTATE:
       rc = ixDspCodeletGetHookState(*(int *)arg);
       *(int *)arg = rc;
       break;

    case IX_DSP_CODELET_POLARITYINV:
       rc = ixDspCodeletPolarityInv(*(int *)arg);
       *(int *)arg = rc;
       break;

    case IX_DSP_CODELET_SHORTRING:
       ixDspCodeletShortRing(*(int *)arg);
       break;

    case IX_DSP_CODELET_NORMALRING:
       ixDspCodeletNormalRing(*(int *)arg);
       break;

    case IX_DSP_CODELET_STOPRING:
       ixDspCodeletStopRing(*(int *)arg);
       break;

    case IX_DSP_CODELET_HOOKEVENT:
       ixDspCodeletSetHookEventCategory(*(int *)arg);
       break;

    case IX_DSP_CODELET_GETRESCONFIG:
       xDspGetResConfig((void *)arg);
       break;

    case IX_DSP_CODELET_PLAYSTART: {
        int i;
        for(i=0; i<IX_DSP_CODELET_MAX_PROMPTS; i++)
              if((CachePrompts[i].pDesc) && (CachePrompts[i].pFd == file)) {
                XMediaHandle_t Handle;
                if((Handle = (xDspRegCachePrompt(CachePrompts[i].pDesc))))
                        CachePrompts[i].handle = Handle;
                *(int *)arg = Handle;
                break;
               }
            break;
           }
     case IX_DSP_CODELET_PLAYSYNC: {
     int i;
           IxDspCodeletPlySyncPrm_t *playSyncPrm;
           playSyncPrm = (IxDspCodeletPlySyncPrm_t *)arg;
           for(i=0; i<IX_DSP_CODELET_MAX_PROMPTS; i++)
                    if(playSyncPrm->handle == CachePrompts[i].handle)
                              if(playSyncPrm->pBuffer) {
                              int err;
```

```
                                    if(CachePrompts[i].pBuffer) {
            err = copy_from_user(CachePrompts[i].pBuffer, playSyncPrm->pBuffer, playSyncPrm->size);
                                            if(!err) return err;
                                    }
                                }
                            }
        break;

        case IX_DSP_CODELET_UNHOOKTRS:
          ixDspCodeletUnHookTransmission(*(int *)arg);
          break;

        default:
          printf("Invalid IOCTL is passed to driver %x \n", functionId);
          break;
       }

      return IX_SUCCESS;
}
```