White Paper

Intel® EP80579 Integrated Processor with Intel® QuickAssist Technology

Integrating a Voice Application with the TDM Infrastructure of the Intel® EP80579 Integrated Processor with Intel® QuickAssist Technology

July 2008

Document Number: 320053-001US

Abstract

This paper describes the main challenges when integrating a Linux* user space voice application with the TDM infrastructure of the Intel® EP80579 Integrated Processor with Intel® QuickAssist Technology, examining decisions that are needed when adding features. As an example, the choices made when integrating Asterisk® with this integrated processor are detailed.

The authors present a brief overview of the processor's TDM Infrastructure interface for Linux user space voice applications.

They describe the different options available for adding FXS, FXO and E1/T1 features, along with the choices made during Asterisk integration. The authors show how an open source DSP software library can be used to add an additional feature to the voice application.

The authors conclude with a summary of recommendations on the design decisions for integrating voice applications with a particular architecture.

Contents

Introduction

VoIP is becoming more and more prevalent across voice bearing networks and infrastructure devices from the core network to the end user. The main driving force of this is that VoIP providers can take advantage of IP networks to offer extremely cost effective call rates.

However, there is still a need to integrate legacy interfaces into VoIP and IP PBXs to be backward compatible with existing PSTN infrastructure and provide fail safe telephony services in the event of power outages. These legacy interfaces play a major part in IP PBXs for the SMB market with multiple use cases ranging from analog to IP PBX connections, to E1/T1 digital trunks from the IP PBX to the ISP.

The TDM Infrastructure of the Intel® EP80579 Integrated Processor with Intel® QuickAssist Technology offers IP PBX products an ideal opportunity to incorporate legacy interfaces while benefiting from a simpler design with a reduced amount of components and Intel® QuickAssist technology acceleration. This reduces product BOM cost, form factor, platform integration and software complexity, and increases the availability of processor cycles for applications.

The processor's TDM Infrastructure provides a flexible interface and as such, engineers writing or porting a voice application utilizing this interface have a variety of design choices. This whitepaper is designed to offer assistance and make recommendations for these choices and design trade-offs. It discusses at a high level the options available and the type of voice applications they are suitable for. By way of example, the choices made when integrating the Asterisk PBX application are highlighted.

The authors assume that the reader has a solid background in telephony and Linux*. Further related documentation including a detailed description of the processor's TDM Infrastructure interface is listed in the references section.

TDM Infrastructure of the Processor

The processor's TDM Infrastructure for voice allows TDM connectivity through three High Speed Serial (HSS) ports. Each of these HSS ports can be independently configured to be E1, T1 or H-MVIP (4 E1s or 4 T1s). In total, they can support a maximum of 128 voice channels.

These ports can be used to connect external telephony circuitry and equipment. External circuitry containing Subscriber Loop Interface Circuit (SLIC/Codec) and Data Access Arrangement (DAA) can be connected to support FXS and FXO lines respectively. Additionally, external circuitry containing a framer can be connected to support digital lines.

There are two mezzanine cards that can be connected to the Intel® EP80579 Integrated Processor Development Platform (Customer Reference Board):

- The Intel® EPAVM80579 Analog Voice Mezzanine Card has 4 SLIC/Codec chips and a DAA which in total support 4 FXS lines and 1 FXO line.
- The Intel® EPTEM80579 T1/E1 Mezzanine Card has a quad span framer and supports 4 E1/T1 links.

Linux User Space Interface

The processor's TDM Infrastructure can be accessed by Linux user space voice applications through a set of software components. Together with the TDM setup driver that initializes underlying components on loading, two drivers and a library provide an API to Linux user space applications for the initialization, configuration and operation of the processor's TDM Infrastructure.

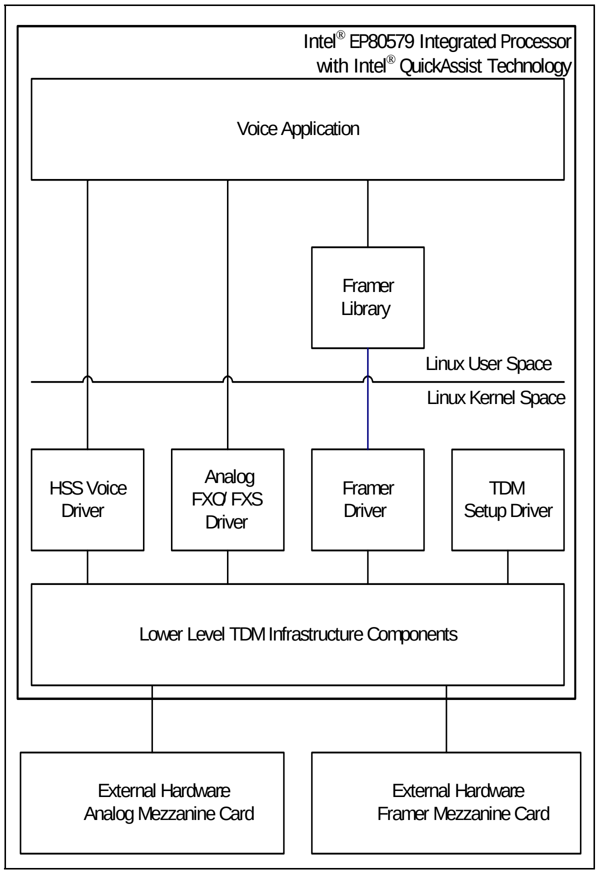
The analog FXO/FXS driver provides a control path interface to the FXO/FXS interfaces on analog voice mezzanine cards. It enables the user to send commands to and receive events from the SLIC/Codec and DAA chips, for example, ring a POTS phone connected to the SLIC/Codec or detect that there is an incoming call from a Central Office (CO) connected to the DAA.

The HSS voice driver provides a control and data path interface to the HSS ports. It enables the user to configure the voice channels as well as transmit and receive voice samples.

The framer library provides a control path interface to the framer devices on T1/E1 mezzanine cards. It enables the user to send commands to and receive events from the framer devices, for example, set the signalling bits on a particular timeslot or detect loss of sync. This software component resides in Linux user space.

Figure 1 shows the primary TDM Infrastructure components of the processor.

Figure 1. TDM Infrastructure Overview of the Processor


Intel® EP80579 Integrated Processor with Intel® QuickAssist Technology
Voice Application
Framer Library
Linux User Space
Linux Kernel Space
HSS Voice Driver
Analog FXO/ FXS Driver
Framer Driver
TDM Setup Driver
Lower Level TDM Infrastructure Components
External Hardware Analog Mezzanine Card
External Hardware Framer Mezzanine Card

Programming Models

Different programming models are discussed below for the key TDM features. They refer to the use of the software provided by Intel for operation of the HSS ports and analog voice mezzanine cards and the T1/E1 mezzanine cards.

FXS

Voice applications that need to integrate the FXS functionality use the analog FXO/FXS and HSS voice drivers Linux user space API.

Control Path

The analog FXO/FXS driver is responsible for the control path. The voice application opens a single file descriptor for this driver. This file descriptor can then be used to monitor and control all the FXS lines. The voice application initializes the analog voice mezzanine card during setup. As part of this process, the SLIC/ Codec chips are configured with the necessary PCM parameters.

While the system is running, the voice application needs to monitor the events reported by the analog FXO/FXS driver to setup and tear down calls. This can be done by creating a thread that calls the **read()** function in blocking mode on the file descriptor. This function blocks until an event occurs. It then returns this event which is processed by the voice application. Alternatively, the voice application can poll for events using the **read()** function provided by the analog FXO/FXS driver in non-blocking mode. To ensure that events are processed in a timely fashion without introducing unnecessary overhead, it is recommended that voice applications use the analog FXO/FXS driver's blocking **read()** function.

The analog FXO/FXS driver is capable of playing tones that are needed to indicate call state and call progress over FXS lines, for example the busy tone. The use of this feature is recommended since it offloads the tone generation to the SLIC/Codec chips and removes the need to implement a tone generator in the voice application.

To release the analog voice mezzanine card, the file descriptor should be closed.

Data Path

The HSS voice driver is responsible for the data path for voice traffic. It is also responsible for configuring the HSS ports. The voice application needs to open at least one file descriptor in order to configure an HSS port. This can be done independently of other HSS ports using extendible predefined configurations that are provided for the most commonly used configurations. The voice application can then add channels that allocate the resources needed. When adding a channel, the interval of the voice codec, combined with the voice packet size, determines how often voice samples are ready for reception by the voice application. The following formula is used to calculate the voice packet size that should be set for a known codec and a desired read rate:

Voice packet size in bytes = (Codec sampling frequency in Hertz) * (Desired read rate in seconds) * (Codec sample size in bytes)

For example, the G.711 codec samples at 8 kHz and each sample is coded as 8 bits. Therefore, if the desired read rate is 10 ms, the voice packet size needs to be set to 80 bytes.

When adding channels, the voice application must define channel IDs to identify each voice channel. If the voice application is the sole user of the HSS port, it may choose to add all channels during initializing. If the HSS port is to be shared with another voice or data application, channels can be added and removed as needed. This is to ensure that unused HSS channels are available to all applications.

When the voice application brings a channel up, data flow is enabled on the channel. Channels in the "up" state must have their data paths serviced. Care must be taken to service the data path on time, otherwise, a latency or loss of voice samples occurs. As such, it is important that there is minimal delay between bringing a channel up and starting data path servicing on it. If a large number of channels need to be brought up, a temporary thread may be created to read and discard voice samples on the channels currently brought up before proper servicing can occur.

Figure 2 shows that in scenario (a) one channel is being added and enabled. There is insignificant latency between enabling the channel and reading from it, hence there is insignificant latency in the voice path. In scenario (b) all channels are being added and enabled. A thread is created to read voice samples from the channels as they are enabled. This ensures that no latency occurs on the channels that are created earliest.

Figure 2. Adding and Enabling Voice Channels

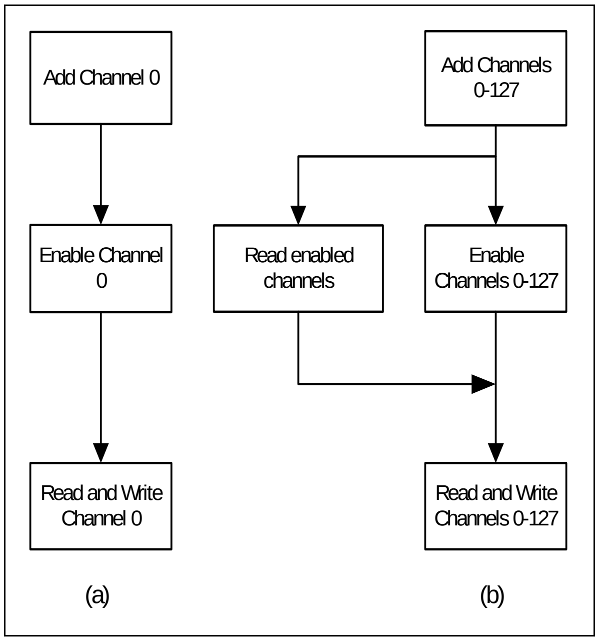
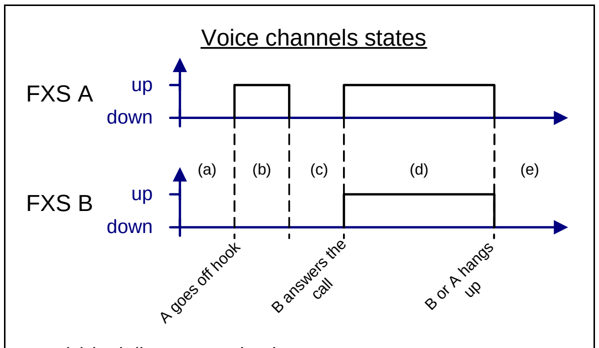


Figure 3 shows an example of when channels should be brought up and down. In this example FXS A calls FXS B.

Figure 3. FXS to FXS Call



(a) both lines are on hook
(b) A has gone off hook to dial an extension, the channel is up to retrieve the digits
(c) A has finished dialling, the PBX is ringing line B
(d) B has answered the call by going off hook, the call is up
(e) Either B or A has hung up

When configuring voice channels, the voice application can decide to associate one or more voice channels with a file descriptor. When the HSS voice driver **read()** function is called, the voice application passes in a file descriptor as part of the parameters. So, the choice of associating channels to file descriptors is important and is discussed further below.

When the HSS voice driver **read()** function is called in blocking mode, the voice application is returned voice samples as soon as they are available for all the channels associated with the file descriptor.

When the HSS voice driver **read()** function is called in non-blocking mode, if voice samples are available for any of the channels associated with the file descriptor, they are returned, otherwise nothing is returned.

In general, it is recommended that voice applications use blocking mode, otherwise they need to implement their own timing system to ensure that they service the data path on time. Using blocking mode means that as soon as data is available on the required channels, it is returned to the voice application.

As part of using blocking mode, the voice application may find it convenient to use the **poll()** method to monitor multiple file descriptors and service channels without blocking other channels.

Some voice applications use only one thread for servicing the data path on all channels. For this type of application, it is recommended that they associate all the channels with a single file descriptor, provided that the read rate is the same on all channels. If more than one read rate is needed, a different file descriptor should be opened for each read rate with the **poll()** method used to prevent the slower read rate channels from introducing latency on the faster read rate channels.

Some voice applications use multiple threads for servicing the data path on channels. In the case where one thread is started for each new call, it is convenient that the application uses one file descriptor per channel. In an alternative case where one thread is used to service channels with the same read rate, it is better that the voice application uses a different file descriptor for each read rate.

When a voice application gets an event to indicate that the call should be finished, it will stop servicing the data path, that is, stop bridging the call. As soon as the voice application stops servicing the data path, the appropriate channels should be brought down. Otherwise, data continues to be received and this may lead to a receive overflow.

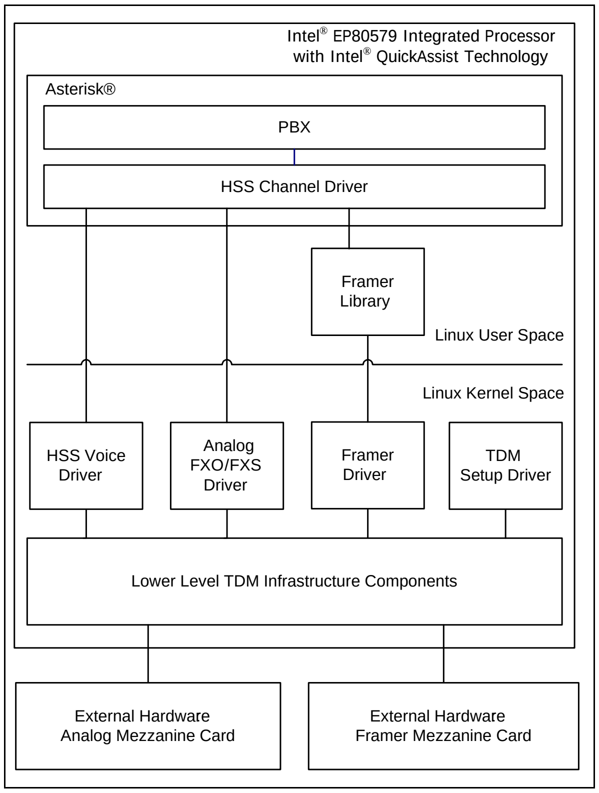
All channels should be removed to deallocate their resources, and finally the HSS ports should be un-initialized. If the HSS ports are also in use by another application, they cannot be brought down. This is the normal scenario where more than one application shares an HSS port.

Programming Model Chosen for integrating with Asterisk®

To integrate FXS functionality with Asterisk a channel driver has been developed that on one side interfaces with Asterisk, and on the other side interfaces with the processor's TDM Infrastructure. On the Asterisk side, the channel driver interfaces with the Asterisk channel API, which is used by all VoIP protocols and technologies that interface to Asterisk. On the processor's TDM Infrastructure side the channel driver interfaces with the analog FXO/FXS driver, HSS voice driver and framer library.

Figure 4 shows the location of this driver in the Asterisk architecture.

Figure 4. TDM Infrastructure with HSS Channel Driver Overview of the Processor


Intel® EP80579 Integrated Processor with Intel® QuickAssist Technology
Asterisk®
PBX
HSS Channel Driver
Framer Library
Linux User Space
Linux Kernel Space
HSS Voice Driver
Analog FXO/FXS Driver
Framer Driver
TDM Setup Driver
Lower Level TDM Infrastructure Components
External Hardware Analog Mezzanine Card
External Hardware Framer Mezzanine Card

To integrate the FXS functionality, the HSS channel driver uses the analog FXO/FXS and HSS voice drivers only.

When the HSS channel driver module is loaded, it initializes the HSS ports and adds as many voice channels as configured FXS lines. Since Asterisk is a multi-threaded application that creates a thread for each new call, the HSS channel driver uses a different file descriptor for each voice channel. All channels are configured with a 10 ms read rate. For the control path a single file descriptor is opened to send commands to and receive events from the analog FXO/FXS driver. To process these events, the HSS channel driver starts a dedicated thread that uses the analog FXO/FXS driver's blocking **read()** function to retrieve events asynchronously.

Channels are brought up and down in accordance with Figure 3.

When a call is established, the newly created thread is responsible for bridging the call. To do this, the thread uses the **poll()** method on both legs of the call to monitor when data is available. As soon as one side has voice samples available, the thread reads them and writes them to the other side of the call.

When the call is over, this thread stops bridging the call and brings down the voice channels. Finally, this thread is destroyed by Asterisk.

When the HSS channel driver is unloaded, the voice channels are removed and the HSS ports in use are released.

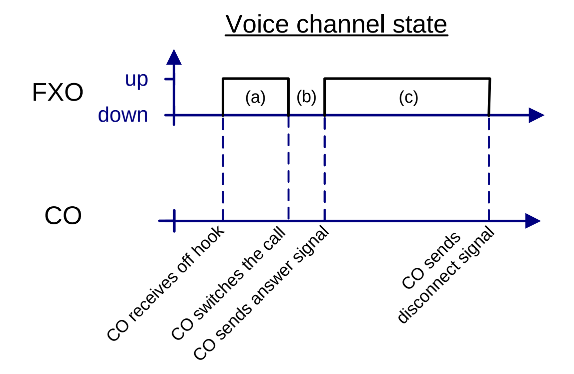
FXO

Using FXO is very similar to using FXS. Different commands and events are sent to and received from the analog FXO/FXS driver, but the methods in sending and receiving them are similar.

There may be some differences in sending commands and handling events depending on whether the telephony system plugged into FXO provides answer and disconnect supervision events, and the analog FXO/FXS driver can detect these events or not.

In the case where these events are supported, a similar model for bringing channels up and down can be used as FXS, since the application is notified when the call is answered and disconnected using events. This is shown in Figure 5 where a call is placed from an FXO interface to a CO. Note that the CO is external equipment that is connected to the FXO interface.

Figure 5. FXO call with Answer and Disconnect Supervision

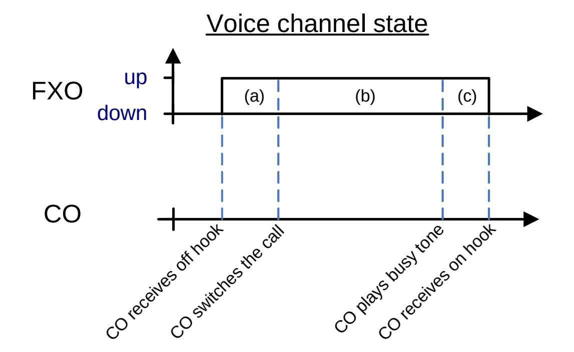


(a) FXO goes off hook and sends the DTMF digits of the extension to call
(b) The channel is brought down while the CO switches the call
(c) FXO received answer signal, the call is up
(d) FXO received disconnect signal and hangs up the call

In the case where answer and disconnect supervision events are not supported by the DAA or CO, it is possible to detect tones in the voice samples to simulate these events. When this occurs, a different model for bringing channels up and down is needed, so that the tones can be detected.

Figure 6 shows an example of a call from an FXO interface to a CO providing tone-based answer and disconnect supervision only.

Figure 6. FXO Call with Tone-based Disconnect Supervision



(a) FXO goes off hook and sends the DTMF digits of the extension to call
(b) The channel stays up to receive and send voice samples
(c) FXO detects busy tone and hangs up the call

Programming Model Chosen for integrating with Asterisk®

The HSS channel driver does not support answer supervision events, and this has the side effect that all outgoing calls through an FXO interface are reported in the CDR as answered.

The disconnection event is simulated through the use of the busy tone. As such, the method used when integrating Asterisk is the one shown in Figure 6.

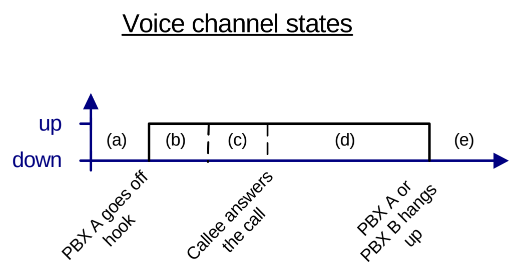
E1/T1

An E1/T1 line can be used as a digital trunk to carry multiple calls on a single link. It can be used to connect a PBX to channel bank equipment to allow multiple POTS lines to be used, or it can also be used to connect a PBX to a CO to provide PSTN connectivity. In addition, an E1/T1 line can be used to connect two PBXs together.

When voice applications want to integrate E1/T1 functionality, they can use the framer library and HSS voice driver Linux user space API.

The data path for E1/T1 differs slightly from the one used for FXS, because the tone generation cannot be offloaded to the framer device. The voice application is now responsible for generating tones over the digital line. Figure 7 shows a call over a digital line between two PBXs, A and B. Both PBXs are configured to support a voice channel to communicate over a digital link. The figure shows how this channel should be brought up and down at both PBXs, when a call is made from PBX A to PBX B.

Figure 7. Call Over a Digital Line



(a) both lines are on hook. Channel is down
(b) PBX A goes off hook and sends the DTMF digits of the extension to call. PBX B retrieves these digits.
(c) PBX B plays the ringback tone to PBX A while ringing the callee
(d) PBX B goes offhook and answers the call
(e) Either PBX A or PBX B hangs up

As the T1/E1 mezzanine card is being used instead of the analog voice mezzanine card, the framer library is needed instead of the analog FXO/FXS driver. The framer library is in user space so the voice application must link to it at compilation/build time.

Each HSS port that is connected to a T1/E1 mezzanine card must be configured and initialized. Events are reported separately for each port through a framer library function that can be used in blocking or non-blocking mode. The voice application may use a thread per port to call the function in blocking mode, to retrieve framer events asynchronously. Alternatively, the voice application can, in a single thread, periodically check for events on each individual port using calls to this function in non-blocking mode. It is recommended that a thread for each HSS port configured for a T1/E1 mezzanine card is used so that events are received and processed without delay.

The voice application can choose to use CAS or CCS signalling over the E1/T1. The framer library provides functions to allow signalling bits to be transmitted and received. The voice application is free to implement whichever signalling protocol they choose.

Programming Model Chosen for integrating with Asterisk®

The model used for the data path is the one presented in Figure 7. All required voice channels are added when the HSS channel driver is initialized, and brought up when a call begins.

To monitor framer events, a new thread is started for each connected T1/E1 mezzanine card, using a blocking call to the framer library function.

To perform signalling over E1/T1, the HSS channel driver implements the commonly used E&M protocol.

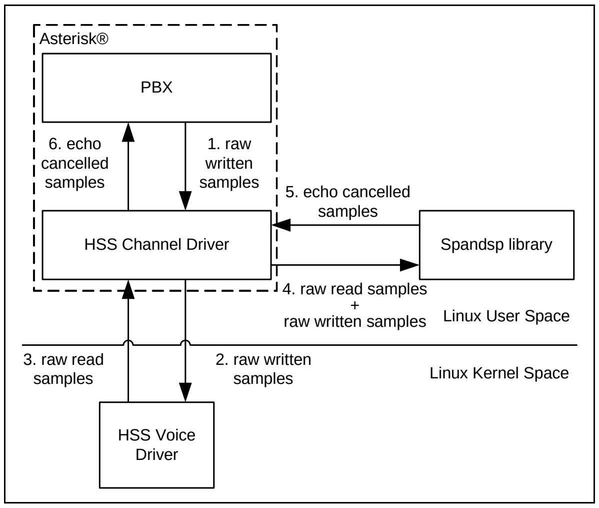
Adding Features

One decision that may face an engineer integrating a voice application with the processor's TDM Infrastructure is whether to write or use a ready made library for additional features.

Since Asterisk is fully-featured, there was little functionality that needed to be added when integrating the HSS channel driver. A feature that was added was echo cancellation as it is not supported by Asterisk. Since writing an echo canceller is a complicated task, it was decided to use the SpanDSP* open source library echo canceller.

To use the SpanDSP library, the HSS channel driver keeps a copy of the voice samples it writes to the voice driver. These samples, along with the samples read from the voice driver, are passed into SpanDSP to perform echo cancellation. SpanDSP returns the echo cancelled voice samples and these are passed to Asterisk for call bridging as per normal. This process is demonstrated in Figure 8.

Figure 8. Echo Cancellation using Spandsp Library



Integrating the SpanDSP library is an example of where open source software libraries can be used effectively to reduce time to market.

Conclusions

Voice applications that integrate traditional TDM interfaces differ in their architecture and in the nature of the interfaces they support. This whitepaper shows that the TDM Infrastructure of the Intel® EP80579 Integrated Processor with Intel® QuickAssist Technology provides a flexible interface and as such supports different voice application programming models.

When porting a voice application, it should be possible to use the processor's TDM Infrastructure interface in a way that removes the need for re-architecting the existing voice application.

When writing a new application, the different programming models should be considered as references that will influence the design of the voice application.

One key issue to be considered when choosing a programming model is the scalability to a large number of channels. When using a large number of channels, a single threaded application may be more suitable than a multi-threaded application, as it will reduce the number of user to kernel space transitions and any overhead due to thread management. Care must be taken to ensure that the codec intervals of the channels are the same, otherwise, voice samples for some channels may not be available on time.

Since voice processing is time sensitive, it is recommended that the blocking modes for receiving events and data are used. This is to ensure that the voice application is notified as soon as an event occurs or there is data ready to be received. Using the **poll()** method is also highly recommended, as it can allow multiple channels to be monitored at the same time. For real time processing voice channels should not be brought up until the voice application is ready to service them.

The Intel® EP80579 Integrated Processor with Intel® QuickAssist Technology provides a high performance Intel® x86 core with integrated I/O and acceleration technology to take advantage of a robust and ever growing ecosystem of software development. VoIP, analog voice, and converged access appliances have seen their capabilities expand by leaps, while their development complexity and costs shrink. The Intel® EP80579 Integrated Processor with Intel® QuickAssist Technology is proud to play a key role in the new era of VoIP communications.

References

- Intel® EP80579 Software for IP Telephony Applications on Intel® QuickAssist Technology Programmer's Guide
- Intel® EP80579 Software for IP Telephony Applications on Intel® QuickAssist Technology Linux* Device Driver API Reference Manual
- Intel® EP80579 Software for IP Telephony Applications on Intel® QuickAssist Technology TDM I/O Access API Reference Manual

Please contact your Intel Sales Representative for more information on these documents.