Voice API for Host Media Processing

Programming Guide

September 2004

05-2332-001

Publication Date: September 2004

Document Number: 05-2332-001

Intel Converged Communications, Inc.
1515 Route 10
Parsippany, NJ 07054

For **Technical Support**, visit the Intel Telecom Support Resources website at:
http://developer.intel.com/design/telecom/support

For **Products and Services Information**, visit the Intel Telecom Products website at:
http://www.intel.com/design/network/products/telecom

For **Sales Offices** and other contact information, visit the Where to Buy Intel Telecom Products page at:
http://www.intel.com/buy/wtb/wtb1028.htm

Contents

Figures

Tables

Revision History

This revision history summarizes the changes made in each published version of this document.

Document No.	Publication Date	Description of Revisions
05-2332-001	September 2004	Initial version of document.

About This Publication

The following topics provide information about this publication:

Purpose

This guide provides instructions for developing applications on Linux* and Windows* operating systems using the voice API that is supplied with the Intel® NetStructure™ Host Media Processing (HMP) product. This document is a companion guide to the *Voice API for Host Media Processing Library Reference*, which describes the voice functions, data structures, events, and error codes.

Applicability

This document is published for Intel® NetStructure™ Host Media Processing Software Release 1.2 for Linux* operating system.

This document may also be applicable to later software releases (including service packs and feature packs) on Linux or Windows. Check the Release Guide for your software release to determine whether this document is supported.

Intended Audience

This guide is intended for software developers who will access the voice software. They may include any of the following:

- Distributors
- System Integrators
- Toolkit Developers
- Independent Software Vendors (ISVs)
- Value Added Resellers (VARs)
- Original Equipment Manufacturers (OEMs)

How to Use This Publication

This publication assumes that you are familiar with the Linux or Windows operating systems and the C programming language.

The information in this guide is organized as follows:

- Chapter 1, "Product Description" introduces the key features of the voice library and provides a brief description of each feature.
- Chapter 2, "Programming Models" provides a brief overview of supported programming models.
- Chapter 3, "Device Handling" discusses topics related to devices such as device naming concepts.
- Chapter 4, "Event Handling" provides information on functions used to handle events.
- Chapter 5, "Error Handling" provides information on handling errors in your application.
- Chapter 6, "Application Development Guidelines" provides programming guidelines and techniques for developing an application using the voice library.
- Chapter 7, "Call Progress Analysis" describes components of call progress analysis and discusses how to use call progress analysis.
- Chapter 8, "Recording and Playback" discusses playback and recording features, such as encoding algorithms, and play and record API functions.
- Chapter 9, "Volume Control" explains how to control volume of playback recordings through API functions and data structures.
- Chapter 10, "Global Tone Detection and Generation, and Cadenced Tone Generation" describes these tone detection and tone generation features in detail.
- Chapter 11, "Building Applications" discusses compiling and linking requirements such as include files and library files.

A glossary and index are provided for your reference.

Related Information

Refer to the following sources for more information:

- For details on all functions and data structures in the voice library, see the *Voice API for Host Media Processing Library Reference*.
- For details on the Standard Runtime Library (SRL), supported programming models, and programming guidelines for building all applications, *Standard Runtime Library API Programming Guide*. The Standard Runtime Library is a device-independent library that consists of event management functions and standard attribute functions.
- For details on all functions and data structures in the Standard Runtime Library (SRL) library, see the *Standard Runtime Library API Library Reference*.
- For information on the software release, system requirements, features, and documentation, see the Release Guide for the software release you are using.

- For details on known problems and late-breaking updates or corrections to the release documentation, see the Release Update.

  Be sure to check the Release Update for the software release you are using for any updates or corrections to this publication. Release Updates are available on the Telecom Support Resources website at *http://resource.intel.com/telecom/support/documentation/releases/index.htm*
- For guidelines on building applications using Global Call software (a common signaling interface for network-enabled applications, regardless of the signaling protocol needed to connect to the local telephone network), see the *Global Call API for Host Media Processing Programming Guide*.
- For details on all functions and data structures in the Global Call library, see the *Global Call API for Host Media Processing Library Reference*.
- For Global Call IP-specific information, see the *Global Call IP for Host Media Processing Technology Guide*.
- For details on all functions and data structures in the IP Media Library, see the *IP Media Library API for Host Media Processing Library Reference*.
- For guidelines on building applications using the IP Media software, see the *IP Media Library API for Host Media Processing Programming Guide*.

Product Description 1

This chapter provides information on key voice library features and capability. The following topics are covered:

1.1 Overview

The voice software is a building block for creating computer telephony applications. It offers a comprehensive set of features such as dual-tone multifrequency (DTMF) detection, tone signaling, call progress analysis, playing and recording that supports a number of encoding methods, and much more.

The voice software consists of a C language library of functions, device drivers, and firmware.

The voice library is well integrated with other technology libraries provided by Intel such as fax, conferencing, and continuous speech processing. This architecture enables you to add new capability to your voice application over time.

For a list of voice features by product, see the Release Guide for your system release.

1.2 R4 API

The term R4 API (“System Software Release 4 Application Programming Interface”) describes the direct interface used for creating computer telephony application programs. The R4 API is a rich set of proprietary APIs for building computer telephony applications on Intel telecom products, including Intel® NetStructure™ Host Media Processing (HMP) software. These APIs encompass technologies that include voice, conferencing, fax, and speech. This document describes the voice API.

In this document, the term voice API is used to refer to the R4 voice API.

1.3 Host Media Processing

Intel® NetStructure™ Host Media Processing (HMP) software performs media processing tasks on general-purpose servers based on Intel architecture without the need for specialized hardware. When installed on a system, HMP performs like a virtual DM3 board to the customer application, but all media processing takes place on the host processor. In this document, the term "board" represents the virtual DM3 board.

DM3 boards refer to products that are based on the next-generation DM3 mediastream architecture. These products succeed the original Springware products (also known as earlier-generation products).

1.4 Call Progress Analysis

Call progress analysis monitors the progress of an outbound call after it is dialed into the Public Switched Telephone Network (PSTN).

See Chapter 7, "Call Progress Analysis" for detailed information about this feature.

1.5 Tone Generation and Detection Features

In addition to DTMF and MF tone detection and generation, the following signaling features are provided by the voice library:

- Global Tone Detection (GTD)
- Global Tone Generation (GTG)
- Cadenced Tone Generation

1.5.1 Global Tone Detection (GTD)

Global tone detection allows you to define single- or dual-frequency tones for detection on a channel-by-channel basis. Global tone detection and GTD tones are also known as **user-defined tone detection** and **user-defined tones**.

Use global tone detection to detect single- or dual-frequency tones outside the standard DTMF range of 0-9, a-d, *, and #. The characteristics of a tone can be defined and tone detection can be enabled using GTD functions and data structures provided in the voice library.

See Chapter 10, "Global Tone Detection and Generation, and Cadenced Tone Generation" for detailed information about global tone detection.

1.5.2 Global Tone Generation (GTG)

Global tone generation allows you to define a single- or dual-frequency tone in a tone generation template and to play the tone on a specified channel.

See Chapter 10, “Global Tone Detection and Generation, and Cadenced Tone Generation” for detailed information about global tone generation.

1.5.3 Cadenced Tone Generation

Cadenced tone generation is an enhancement to global tone generation. It allows you to generate a tone with up to 4 single- or dual-tone elements, each with its own on/off duration, which creates the signal pattern or cadence. You can define your own custom cadenced tone or take advantage of the built-in set of standard PBX call progress signals, such as dial tone, ringback, and busy.

See Chapter 10, “Global Tone Detection and Generation, and Cadenced Tone Generation” for detailed information about cadenced tone generation.

1.6 Play and Record Features

The following play and record features are provided by the voice library:

- Play and Record Functions
- Volume Control
- Streaming to Board

1.6.1 Play and Record Functions

The voice library includes several functions and data structures for recording and playing audio data. These allow you to digitize and store human voice; then retrieve, convert, and play this digital information.

For more information about play and record features, see Chapter 8, “Recording and Playback”. This chapter also includes information about voice encoding methods supported; see Section 8.5, “Voice Encoding Methods”, on page 49. For detailed information about play and record functions, see the *Voice API Library Reference*.

1.6.2 Volume Control

The volume control feature allows you to control the volume of a message being played on a channel, for example, by entering a DTMF tone.

Se Chapter 9, “Volume Control” for more information about this feature.

1.6.3 Streaming to Board

The streaming to board feature allows you to stream data to a network interface in real time. Unlike the standard voice play feature (store and forward), data can be streamed in real time with little delay as the amount of initial data required to start the stream is configurable. The streaming to

board feature is essential for applications such as text-to-speech, distributed prompt servers, and IP gateways.

For more information about this feature, see Chapter 8, "Recording and Playback".

1.7 TDM Bus Routing

A time division multiplexing (TDM) bus is a technique for transmitting a number of separate digitized signals simultaneously over a communication medium. TDM bus includes the CT Bus. On HMP, no physical TDM bus exists but its functionality is implemented in the software.

For information on TDM bus routing functions, see the *Voice API Library Reference*.

Programming Models 2

This chapter briefly discusses the Standard Runtime Library and supported programming models:

2.1 Standard Runtime Library

The Standard Runtime Library (SRL) provides a set of common system functions that are device independent and are applicable to all Intel® telecom devices. The SRL consists of a data structure, event management functions, device management functions (called standard attribute functions), and device mapper functions. You can use the SRL to simplify application development, such as by writing common event handlers to be used by all devices.

When developing voice processing applications, refer to the Standard Runtime Library documentation in tandem with the voice library documentation. For more information on the Standard Runtime Library, see the *Standard Runtime Library API Library Reference* and *Standard Runtime Library API Programming Guide*.

2.2 Asynchronous Programming Models

Asynchronous programming enables a single program to control multiple voice channels within a single process. This allows the development of complex applications where multiple tasks must be coordinated simultaneously.

The asynchronous programming model uses functions that do not block thread execution; that is, the function continues processing under the hood. A Standard Runtime Library (SRL) event later indicates function completion.

Generally, if you are building applications that use any significant density, you should use the asynchronous programming model to develop field solutions.

For complete information on asynchronous programming models, see the *Standard Runtime Library API Programming Guide*.

2.3 Synchronous Programming Model

The synchronous programming model uses functions that block application execution until the function completes. This model requires that each channel be controlled from a separate process. This allows you to assign distinct applications to different channels dynamically in real time.

Synchronous programming models allow you to scale an application by simply instantiating more threads or processes (one per channel). This programming model may be easy to encode and manage but it relies on the system to manage scalability. Applying the synchronous programming model can consume large amounts of system overhead, which reduces the achievable densities and negatively impacts timely servicing of both hardware and software interrupts. Using this model, a developer can only solve system performance issues by adding memory or increasing CPU speed or both. The synchronous programming models may be useful for testing or very low-density solutions.

For complete information on synchronous programming models, see the *Standard Runtime Library API Programming Guide*.

Device Handling 3

This chapter describes the concept of a voice device and how voice devices are named and used.

3.1 Device Concepts

The following concepts are key to understanding devices and device handling:

device
: A device is a computer component controlled through a software device driver. A resource board, such as a voice resource, fax resource, and conferencing resource, and network interface board, contains one or more logical board devices. Each channel or time slot on the board is also considered a device.

device channel
: A device channel refers to a data path that processes one incoming or outgoing call at a time (equivalent to the terminal equipment terminating a phone line).

device name
: A device name is a literal reference to a device, used to gain access to the device via an xx_open( ) function, where “xx” is the prefix defining the device to be opened. For example, “dx” is the prefix for voice device and “fx” for fax device.

device handle
: A device handle is a numerical reference to a device, obtained when a device is opened using xx_open( ), where “xx” is the prefix defining the device to be opened. The device handle is used for all operations on that device.

virtual boards
: The device driver views a single voice board with more than four channels as multiple emulated D/4x boards. These emulated boards are called virtual boards. For example, a system with 44 voice channels consists of 11 virtual boards.

3.2 Voice Device Names

The software assigns a device name to each device or each component on a board. A voice device is named **dxxxBn**, where **n** is the device number assigned in sequential order down the list of sorted voice boards. A device corresponds to a grouping of two or four voice channels.

For example, a system running Intel® NetStructure™ Host Media Processing (HMP) software with 44 voice channels has 11 virtual board devices, where each device consists of four channels. Examples of board device names for voice boards are dxxxB1 and dxxxB2.

A device name can be appended with a channel or component identifier. A voice channel device is named **dxxxBnCy**, where **y** corresponds to one of the voice channels. Examples of channel device names for voice boards are dxxxB1C1 and dxxxB1C2.

Use the Standard Runtime Library device mapper functions to retrieve information on all devices in a system.

For complete information on device handling, see the *Standard Runtime Library API Programming Guide*.

Event Handling 4

This chapter provides information on functions used to retrieve and handle events. Topics include:

4.1 Overview of Event Handling

An event indicates that a specific activity has occurred on a channel. The voice driver reports channel activity to the application program in the form of events, which allows the program to identify and respond to a specific occurrence on a channel. Events provide feedback on the progress and completion of functions and indicate the occurrence of other channel activities. Voice library events are defined in the *dxxxlib.h* header file.

For a list of events that may be returned by the voice software, see the *Voice API Library Reference*.

4.2 Event Management Functions

Event management functions are used to retrieve and handle events being sent to the application from the firmware. These functions are contained in the Standard Runtime Library (SRL) and defined in *srllib.h*. The SRL provides a set of common system functions that are device independent and are applicable to all Intel® telecom devices. For more information on event management and event handling, see the *Standard Runtime Library API Programming Guide*.

Event management functions include:

- **sr_enbhdlr( )**
- **sr_dishdlr( )**
- **sr_getevtdev( )**
- **sr_getevttype( )**
- **sr_getevtlen( )**
- **sr_getevtdatap( )**

For details on SRL functions, see the *Standard Runtime Library API Library Reference*.

The event management functions retrieve and handle voice device termination events for functions that run in asynchronous mode, such as **dx_dial( )** and **dx_play( )**. For complete function reference information, see the *Voice API Library Reference*.

Each of the event management functions applicable to the voice boards are listed in the following tables. Table 1 lists values that are required by event management functions. Table 2 list values that are returned for event management functions that are used with voice devices.

Table 1. Voice Device Inputs for Event Management Functions

Event Management Function	Voice Device Input	Valid Value	Related Voice Functions
sr_enbhdlr( ) Enable event handler	evt_type	TDX_PLAY	**dx_play( )**
		TDX_PLAYTONE	**dx_playtone( )**
		TDX_RECORD	**dx_rec( )**
		TDX_GETDIG	**dx_getdig( )**
		TDX_DIAL	**dx_dial( )**
		TDX_CALLP	**dx_dial( )**
		TDX_ERROR	All asynchronous functions
sr_dishdlr( ) Disable event handler	evt_type	As above	As above

Table 2. Voice Device Returns from Event Management Functions

Event Management Function	Return Description	Returned Value	Related Voice Functions
sr_getevtdev( ) Get device handle	device	voice device handle	
sr_getevttype( ) Get event type	event type	TDX_PLAY	**dx_play( )**
		TDX_PLAYTONE	**dx_playtone( )**
		TDX_RECORD	**dx_rec( )**
		TDX_GETDIG	**dx_getdig( )**
		TDX_DIAL	**dx_dial( )**
		TDX_CALLP	**dx_dial( )**
		TDX_CST	**dx_setevtmsk( )**
		TDX_ERROR	All asynchronous functions
sr_getevtlen( ) Get event data length	event length	sizeof (DX_CST)	
sr_getevtdatap( ) Get pointer to event data	event data	pointer to DX_CST structure	

Error Handling 5

This chapter discusses how to handle errors that can occur when running an application.

All voice library functions return a value to indicate success or failure of the function. A return value of zero or a non-negative number indicates success. A return value of -1 indicates failure.

If a voice library function fails, call the standard attribute functions **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to determine the reason for failure. For more information on these functions, see the *Standard Runtime Library API Library Reference*.

If an extended attribute function fails, two types of errors can be generated. An extended attribute function that returns a pointer will produce a pointer to the ASCIIZ string "Unknown device" if it fails. An extended attribute function that does not return a pointer will produce a value of AT_FAILURE if it fails. Extended attribute functions for the voice library are prefaced with "ATDX_".

Notes: ***1.*** The **dx_open( )** and **dx_close( )** functions are exceptions to the above error handling rules. On Linux, if these functions fail, the return code is -1, and the specific error is found in the **errno** variable contained in *errno.h*. On Windows, if these functions fail, the return code is -1. Use **dx_fileerrno( )** to obtain the system error value.

2. If **ATDV_LASTERR( )** returns the EDX_SYSTEM error code, an operating system error has occurred. On Linux, check the global variable **errno** contained in *errno.h*. On Windows, use **dx_fileerrno( )** to obtain the system error value.

For a list of errors that can be returned by a voice library function, see the *Voice API Library Reference*. You can also look up the error codes in the *dxxxlib.h file*.

Application Development Guidelines 6

This chapter provides programming guidelines and techniques for developing an application using the voice library. The following topics are discussed:

6.1 General Considerations

The following considerations apply to all applications written using the voice API:

- Busy and Idle States
- Setting Termination Conditions for I/O Functions
- Setting Termination Conditions for Digits
- Clearing Structures Before Use

See feature chapters for programming guidelines specific to a feature, such as call progress analysis, recording and playback, and so on.

6.1.1 Busy and Idle States

The operation of some library functions are dependent on the state of the device when the function call is made. A device is in an idle state when it is not being used, and in a busy state when it is dialing, stopped, being configured, or being used for other I/O functions. Idle represents a single state; busy represents the set of states that a device may be in when it is not idle. State-dependent functions do not make a distinction between the individual states represented by the term busy. They only distinguish between idle and busy states.

For more information on categories of functions and their description, see the *Voice API Library Reference*.

6.1.2 Setting Termination Conditions for I/O Functions

When an I/O function is issued, you must pass a set of termination conditions as one of the function parameters. Termination conditions are events monitored during the I/O process that will cause an I/O function to terminate. When the termination condition is met, a termination reason is returned by **ATDX_TERMMSK( )**. If the I/O function is running in synchronous mode, the **ATDX_TERMMSK( )** function returns a termination reason after the I/O function has completed. If the I/O function is running in asynchronous mode, the **ATDX_TERMMSK( )** function returns a

termination reason after the function termination event has arrived. I/O functions can terminate under several conditions as described later in this section.

You can predict events that will occur during I/O (such as a digit being received or the call being disconnected) and set termination conditions accordingly. The flow of control in a voice application is based on the termination condition. Setting these conditions properly allows you to build voice applications that can anticipate a caller's actions.

To set the termination conditions, values are placed in fields of a DV_TPT structure. If you set more than one termination condition, the first one that occurs will terminate the I/O function. The DV_TPT structures can be configured as a linked list or array, with each DV_TPT specifying a single terminating condition. For more information on the DV_TPT structure, which is defined in *srllib.h*, see the *Voice API Library Reference*.

The termination conditions are described in the following paragraphs.

byte transfer count
: This termination condition applies when playing or recording a file with **dx_play( )** or **dx_rec( )**. The maximum number of bytes is set in the DX_IOTT structure. This condition will cause termination if the maximum number of bytes is used before one of the termination conditions specified in the DV_TPT occurs. For information about setting the number of bytes in the DX_IOTT, see the *Voice API Library Reference*.

dx_stopch( ) occurred
: The **dx_stopch( )** function will terminate any I/O function, except **dx_dial( )** (with call progress analysis disabled), and stop the device. See the **dx_stopch( )** function description for more detailed information about this function.

end of file reached
: This termination condition applies when playing a file. This condition will cause termination if -1 has been specified in the io_length field of the DX_IOTT, and no other termination condition has occurred before the end of the file is reached. For information about setting the DX_IOTT, see the *Voice API Library Reference*. When this termination condition is met, a TM_EOD termination reason is returned from **ATDX_TERMMSK( )**.

maximum delay between digits (DX_IDDTIME)
: This termination condition monitors the length of time between the digits being received. A specific length of time can be placed in the tp_length field of a DV_TPT. If the time between receiving digits is more than this period of time, the function terminates. The amount of time can be specified in 100 msec units (default) or 10 msec units. 10 msec units can be specified in the tp_flags field of the DV_TPT. When this termination condition is met, a TM_IDDTIME termination reason is returned from **ATDX_TERMMSK( )**.

  This termination condition is only supported by the **dx_getdig( )** function.

maximum digits received (DX_MAXDTMF)
: This termination condition counts the number of digits in the channel's digit buffer. If the buffer is not empty before the I/O function is called, the digits that are present in the buffer when the function is initiated are counted as well. The maximum number of digits to receive is set by placing a number from 1 to 31 in the tp_length field of a DV_TPT. This value specifies the number of digits allowed in the buffer before termination. When this termination condition is met, a TM_MAXDTMF termination reason is returned from **ATDX_TERMMSK( )**.

maximum length of silence (DX_MAXSIL)

This termination condition is enabled by setting the tp_length field of a DV_TPT. The specified value is the length of time that continuous silence will be detected before it terminates the I/O function. The amount of time can be specified in 100 msec units (default) or 10 msec units. 10 msec units can be specified in the tp_flags field of the DV_TPT. When this termination condition is met, a TM_MAXSIL termination reason is returned from **ATDX_TERMMSK( )**.

specific digit received (DX_DIGMASK)

Digits received during an I/O function are collected in a channel's digit buffer. If the buffer is not empty before an I/O function executes, the digits in the buffer are treated as being received during the I/O execution. This termination condition is enabled by specifying a digit bit mask in the tp_length field of a DV_TPT structure. If any digit specified in the bit mask appears in the digit buffer, the I/O function will terminate. When this termination condition is met, a TM_DIGIT termination reason is returned from **ATDX_TERMMSK( )**.

Using more than one DX_IOTT structure for detecting different digits is not supported. Instead, use one DX_IOTT structure, set DX_DIGMASK in the tp_termno field, and bitwise-OR "DM_1 | DM_2" in the tp_length field.

maximum function time (DX_MAXTIME)

A time limit may be placed on the execution of an I/O function. The tp_length field of a DV_TPT can be set to a specific length of time in 100 msec units. The I/O function will terminate when it executes longer than this period of time. The amount of time can be specified in 100 msec units (default) or 10 msec units. 10 msec units can be specified in the tp_flags field of the DV_TPT. When this termination condition is met, a TM_MAXTIME termination reason is returned from **ATDX_TERMMSK( )**.

DX_MAXTIME is not supported by tone generation functions such as **dx_playtone( )** and **dx_playtoneEx( )**.

user-defined digit received (DX_DIGTYPE)

User-defined digits received during an I/O function are collected in a channel's digit buffer. If the buffer is not empty before an I/O function executes, the digits in the buffer are treated as being received during the I/O execution. This termination condition is enabled by specifying the digit and digit type in the tp_length field of a DV_TPT structure. If any digit specified in the bit mask appears in the digit buffer, the I/O function will terminate. When this termination condition is met, a TM_DIGIT termination reason is returned from **ATDX_TERMMSK( )**.

user-defined tone on/off event detected (DX_TONE)

This termination condition is used with global tone detection. Before specifying a user-defined tone as a termination condition, the tone must first be defined using the GTD **dx_bld...( )** functions, and tone detection on the channel must be enabled using the **dx_addtone( )** or **dx_enbtone( )** function. To set tone on/off to be a termination condition, specify DX_TONE in the tp_termno field of the DV_TPT. You must also specify DX_TONEON or DX_TONEOFF in the tp_data field. When this termination condition is met, a TM_TONE termination reason is returned from **ATDX_TERMMSK( )**.

6.1.3 Setting Termination Conditions for Digits

To specify a timeout for **dx_getdig( )** if the first digit is not received within a specified time period, use the DX_MAXTIME termination condition in the DV_TPT structure.

To specify an additional timeout if subsequent digits are not received, use the DX_IDDTIME (interdigit delay) termination condition and the TF_FIRST flag in the DV_TPT structure. The TF_FIRST flag specifies that the timer will start after the first digit is received; otherwise the timer starts when the **dx_getdig( )** function is called.

6.1.4 Clearing Structures Before Use

Two library functions are provided to clear structures. **dx_clrcap( )** clears DX_CAP structures and **dx_clrtpt( )** clears DV_TPT structures. See the *Voice API Library Reference* for details.

It is good practice to clear the field values of any structure before using the structure in a function call. Doing so will help prevent unintentional settings or terminations.

6.2 Additional Considerations

The following information provides programming guidelines and considerations for developing voice applications:

- Multithreading and Multiprocessing
- Device Initialization Hint
- Tone Detection Considerations

6.2.1 Multithreading and Multiprocessing

The voice API supports multithreading and multiprocessing, with some restrictions on the latter. The restrictions on multiprocessing are outlined below.

First, one particular channel can only be opened in one process at a time. There can, however, be multiple processes accessing different sets of channels. In other words, ensure that each process is provided with a unique set of devices to manipulate.

Second, if a channel was opened in process A and then closed, process B is then allowed to open the same channel. However, since closing a channel is an asynchronous operation, there is a small gap between the time when the xx_close( ) function returns in process A and the time when process B is allowed to open the same channel. If process B opens the channel too early, things could go wrong. For this reason, this type of sequence should be avoided.

Third, there is a restriction on use of firmware tones in case of multiprocessing in the voice API. Multiple processes that define tones (GTD or GTG) do not share tone definitions in the firmware. That is, if you define tone A in process #1 for channel dxxxB1C1 on DM3 board X and the same tone A in process #2 for channel dxxxB1C1 on the same DM3 board X, two firmware tones are consumed on board X. In other words, the same tone defined from different processes is not shared in the firmware; hence this limits the number of tones that can be created overall.

6.2.2 Device Initialization Hint

The xx_open( ) functions for the voice (dx), Global Call (gc), and fax (fx) APIs are asynchronous. This should usually have no impact on an application, except in cases where a subsequent function calls on a device that is still initializing, that is, is in the process of opening. In such cases, the initialization must be finished before the follow-up function can work. The function won't return an error, but it is blocked until the device is initialized.

For instance, if your application calls **dx_open( )** and **dx_getfeaturelist( )**, the **dx_getfeaturelist( )** function is blocked until the initialization of the device is completed internally, even though **dx_open( )** has already returned success. In other words, the initialization (**dx_open( )**) may appear to be complete, but, in truth, it is still going on in parallel.

With some applications, this may cause slow device-initialization performance. You can avoid this problem in one of several ways, depending on the type of application:

- In multithreaded applications, you can reorganize the way the application opens and then configures devices. The recommendation is to do as many xx_open( ) functions as possible (grouping the devices) in one thread arranging them in a loop before proceeding with the next function. For example, you would have one loop through the grouping of devices do all the xx_open( ) functions first, and then start a second loop through the devices to configure them, instead of doing one single loop where an xx_open( ) is immediately followed by other API functions on the same device. With this method, by the time all xx_open( ) commands are completed, the first channel will be initialized, so you won't experience problems.

  This change is not necessary for all applications, but if you experience poor initialization performance, you can gain back speed by using this hint.
- Develop your application using a single thread per span or a single thread per board. This way, device initialization can still be done in a loop, and by the time the subsequent function is called on the first device, initialization on that device has completed.

6.2.3 Tone Detection Considerations

In certain scenarios in a voice and conferencing application, HMP may detect a DTMF tone multiple times. Tone detection is performed separately on voice and conference devices. If the tone duration is long enough and the device is switched from voice to conference, then the tone may be detected twice as the algorithm detects the tone on the rising edge (for tone clamping purposes). To resolve this issue, you can add a delay before calling **dx_clrdigbuf( )**.

Call Progress Analysis 7

This chapter provides detailed information about the call progress analysis feature. The following topics are discussed:

7.1 Call Progress Analysis Overview

Call progress analysis monitors the progress of an outbound call after it is dialed into the Public Switched Telephone Network (PSTN).

By using call progress analysis (CPA) you can determine for example:

- whether the line is answered and, in many cases, how the line is answered
- whether the line rings but is not answered
- whether the line is busy
- whether there is a problem in completing the call

The outcome of the call is returned to the application when call progress analysis has completed.

Caution: If your application also uses the Global Call API, see the Global Call documentation set for call progress analysis considerations specific to Global Call. The Global Call API is a common signaling interface for network-enabled applications, regardless of the signaling protocol needed to connect to the local telephone network. Call progress analysis support varies with the protocol used.

7.2 Call Progress and Call Analysis Terminology

A distinction is made between activity that occurs before a call is connected and after a call is connected. The following terms are used:

call progress (pre-connect)
: This term refers to activity to determine the status of a call connection, such as busy, no ringback, no dial tone, and can also include the frequency detection of Special Information Tones (SIT), such as operator intercept. This activity occurs before a call is connected.

call analysis (post-connect)
: This term refers to activity to determine the destination party's media type, such as voice detection, answering machine detection, fax tone detection, modem, and so on. This activity occurs after a call is connected.

call progress analysis
: This term refers to the feature set that encompasses both call progress and call analysis.

7.3 Call Progress Analysis Components

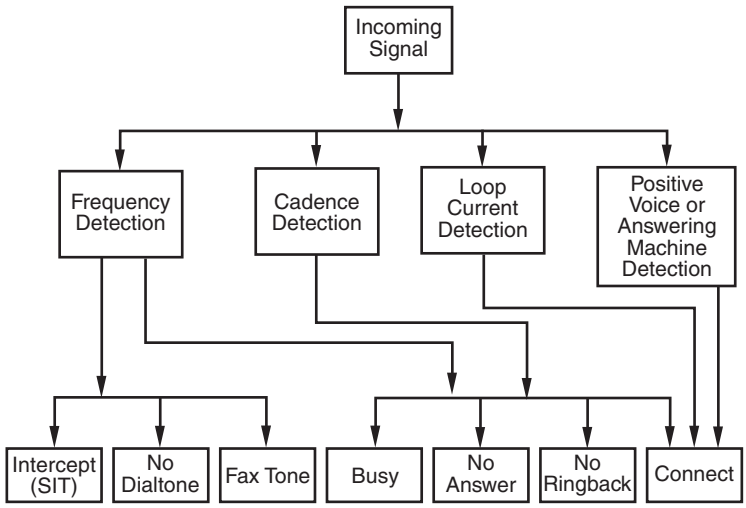
Call progress analysis uses the following techniques or components to determine the progress of a call as applicable:

- cadence detection (pre-connect part of call progress analysis)
- frequency detection (pre-connect part of call progress analysis)
- loop current detection (pre-connect part of call progress analysis)
- positive voice detection (post-connect part of call progress analysis)
- positive answering machine detection (post-connect part of call progress analysis)
- fax tone detection (post-connect part of call progress analysis)

Figure 1 illustrates the components of call progress analysis. These components can all operate simultaneously.

Note: Note that loop current detection is not applicable to HMP.

Figure 1. Call Progress Analysis Components



7.4 Using Call Progress Analysis

The following topics provide information on how to use call progress analysis:

- Call Progress Analysis Rules
- Overview of Steps to Initiate Call Progress Analysis
- Setting Up Call Progress Analysis Parameters in DX_CAP
- Executing a Dial Function
- Determining the Outcome of a Call
- Obtaining Additional Call Outcome Information

7.4.1 Call Progress Analysis Rules

The following rules apply to the use of call progress analysis using HMP voice devices:

- For IP ports using IP protocols:
  - Pre-connect is typically provided by the protocol via the Global Call API or via a native IP protocol API.
  - The **dx_dial( )** function is not recommended to be used for pre-connect. In general this function is not required to be used since pre-connect call progress is typically not provided inband in the media stream. *However, the **dx_dial( )** function may be used if inband pre-connect call progress is available.*
  - The **dx_dial( )** function may be used for post-connect call analysis.

Table 3 provides information on call progress analysis scenarios supported with the **dx_dial( )** function.

Table 3. Call Progress Analysis Support with dx_dial( )

CPA Feature	dx_dial( ) support on HMP
Busy	Yes
No ringback	Yes
SIT frequency detection	Yes
No answer	Yes
Cadence break	Yes
Loop current detection	No
Dial tone detection	No
Fax tone detection	Yes
Positive Voice Detection (PVD)	Yes
Positive Answering Machine Detection (PAMD)	Yes

7.4.2 Overview of Steps to Initiate Call Progress Analysis

Review the information in Section 7.4.1, "Call Progress Analysis Rules", on page 33. If you choose to use the voice API for call progress analysis, perform the following procedure to initiate an outbound call with call progress analysis:

1. Set up the call analysis parameter structure (DX_CAP), which contains parameters to control the operation of call progress analysis, such as positive voice detection and positive answering machine detection.
2. Call **dx_dial( )** to start call progress analysis during the desired phase of the call.
3. Use the **ATDX_CPTERM( )** extended attribute function to determine the outcome of the call.
4. Obtain additional termination information as desired using extended attribute functions.

Each of these steps is described in more detail next. For a full description of the functions and data structures described in this chapter, see the *Voice API Library Reference*.

7.4.3 Setting Up Call Progress Analysis Parameters in DX_CAP

The call progress analysis parameters structure, DX_CAP, is used by **dx_dial( )**. It contains parameters to control the operation of call progress analysis features, such as positive voice detection (PVD) and positive answering machine detection (PAMD). To customize the parameters for your environment, you must set up the DX_CAP structure before calling a dial function.

To set up the DX_CAP structure for call progress analysis:

1. Execute the **dx_clrcap( )** function to clear the DX_CAP and initialize the parameters to 0. The value 0 indicates that the default value will be used for that particular parameter. **dx_dial( )** can

also be set to run with default call progress analysis parameter values, by specifying a NULL pointer to the DX_CAP structure.

2. Set a DX_CAP parameter to another value if you do not want to use the default value. The ca_intflg field (intercept mode flag) of DX_CAP enables and disables the following call progress analysis components: SIT frequency detection, positive voice detection (PVD), and positive answering machine detection (PAMD). Use one of the following values for the ca_intflg field:
   - DX_OPTDIS. Disables Special Information Tone (SIT) frequency detection, PAMD, and PVD. This setting provides call progress without SIT frequency detection.
   - DX_OPTNOCON. Enables SIT frequency detection and returns an "intercept" immediately after detecting a valid frequency. This setting provides call progress with SIT frequency detection.
   - DX_PVDENABLE. Enables PVD and fax tone detection. Provides PVD call analysis only (no call progress).
   - DX_PVDOPTNOCON. Enables PVD, DX_OPTNOCON, and fax tone detection. This setting provides call progress with SIT frequency detection and PVD call analysis.
   - DX_PAMDENABLE. Enables PAMD, PVD, and fax tone detection. This setting provides PAMD and PVD call analysis only (no call progress).
   - DX_PAMDOPTEN. Enables PAMD, PVD, DX_OPTNOCON, and fax tone detection. This setting provides full call progress and call analysis.

   Note: DX_OPTEN and DX_PVDOPTEN are obsolete. Use DX_OPTNOCON and DX_PVDOPTNOCON instead.

7.4.4 Executing a Dial Function

To use call progress analysis, call **dx_dial( )** with the **mode** function argument set to DX_CALLP. Termination of dialing with call progress analysis is indicated differently depending on whether the function is running asynchronously or synchronously.

If running asynchronously, use Standard Runtime Library (SRL) event management functions to determine when dialing with call progress analysis is complete (TDX_CALLP termination event).

If running synchronously, wait for the function to return a value greater than 0 to indicate successful completion.

Notes: ***1.*** **dx_dial( )** cannot be used to start an outbound call; instead use the Global Call API.

2. To issue **dx_dial( )** without dialing digits, specify " " in the **dialstrp** argument.

7.4.5 Determining the Outcome of a Call

In asynchronous mode, once **dx_dial( )** with call progress analysis has terminated, use the extended attribute function **ATDX_CPTERM( )** to determine the outcome of the call. (In synchronous mode, **dx_dial( )** returns the outcome of the call.) **ATDX_CPTERM( )** will return one of the following call progress analysis termination results:

CR_BUSY
Called line was busy.

CR_CEPT
Called line received operator intercept (SIT).

CR_CNCT
Called line was connected. Use **ATDX_CONNTYPE( )** to return the connection type for a completed call.

CR_ERROR
Call progress analysis error occurred. Use **ATDX_CPERROR( )** to return the type of error.

CR_FAXTONE
Called line was answered by fax machine or modem.

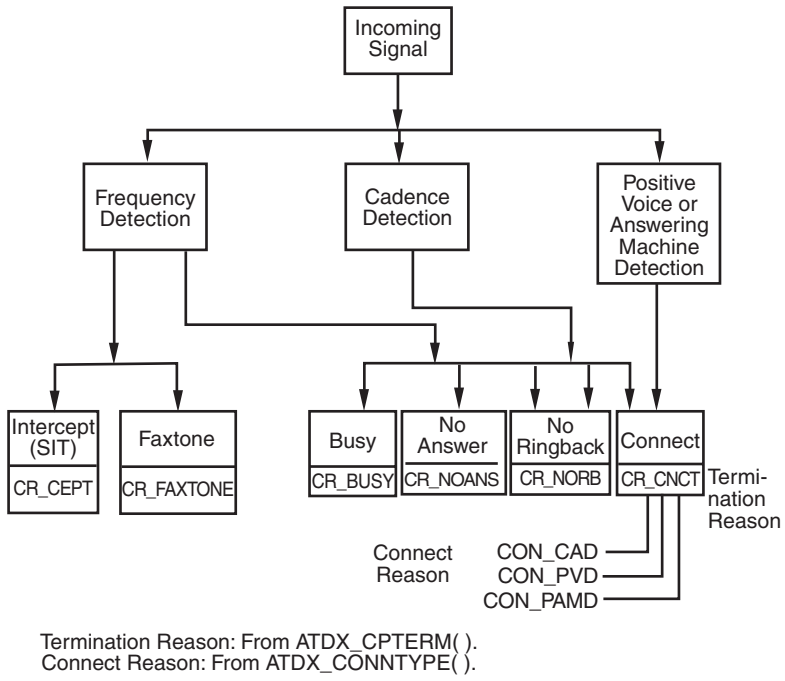
CR_NOANS
Called line did not answer.

CR_NORB
No ringback on called line.

CR_STOPD
Call progress analysis stopped due to **dx_stopch( )**.

Figure 2 illustrates the possible outcomes of call progress analysis.

Figure 2. Call Outcomes for Call Progress Analysis



Termination Reason: From ATDX_CPTERM( ).
Connect Reason: From ATDX_CONNTYPE( ).

7.4.6 Obtaining Additional Call Outcome Information

To obtain additional call progress analysis information, use the following extended attribute functions:

ATDX_CPERROR( )
Returns call analysis error.

ATDX_CPTERM( )
Returns last call analysis termination reason.

ATDX_CONNTYPE( )
Returns connection type.

See each function reference description in the *Voice API Library Reference* for more information.

7.5 Call Progress Analysis Tone Detection

The following topics discuss tone detection used in call progress analysis:

- Tone Detection Overview
- Types of Tones
- Ringback Detection
- Busy Tone Detection
- Fax or Modem Tone Detection
- SIT Frequency Detection

7.5.1 Tone Detection Overview

Call progress analysis uses a combination of cadence detection and frequency detection to identify certain signals during the course of an outgoing call. Cadence detection identifies repeating patterns of sound and silence, and frequency detection determines the pitch of the signal. Together, the cadence and frequency of a signal make up its “tone definition”.

7.5.2 Types of Tones

Tone definitions are used to identify several kinds of signals.

The following defined tones and tone identifiers are provided by the voice library.

TID_BUSY1
Busy signal

TID_BUSY2
Alternate busy signal

TID_DIAL_INTL
International dial tone

TID_DIAL_LCL
Local dial tone

TID_FAX1
CNG (calling) fax tone or modem tone

TID_FAX2
CED (called station) fax tone or modem tone

TID_RNGBK1
Ringback

TID_RNGBK2
Ringback

The tone identifiers are used as input to function calls to change the tone definitions. For more information, see Section 7.8, "Modifying Default Call Progress Analysis Tone Definitions", on page 42.

7.5.3 Ringback Detection

Call progress analysis uses the tone definition for ringback to identify the first ringback signal of an outgoing call. At the end of the first ringback (that is, normally, at the beginning of the second ringback), a timer goes into effect. The system continues to identify ringback signals (but does not count them). If a break occurs in the ringback cadence, the call is assumed to have been answered, and call progress analysis terminates with the reason CR_CNCT (connect); the connection type returned by the **ATDX_CONNTYPE( )** function will be CON_CAD (cadence break).

However, if the timer expires before a connect is detected, then the call is deemed unanswered, and call progress analysis terminates with the reason CR_NOANS.

To enable ringback detection, turn on SIT frequency detection in the DX_CAP ca_intflg field. For details, see Section 7.4.3, "Setting Up Call Progress Analysis Parameters in DX_CAP", on page 34.

The following DX_CAP fields govern ringback behavior:

ca_cnosig
Continuous No Signal: the maximum length of silence (no signal) allowed immediately after the ca_stdely period (in 10 msec units). If this duration is exceeded, call progress analysis is terminated with the reason CR_NORB (no ringback detected). Default value: 4000 (40 seconds).

ca_noanswer
No Answer: the length of time to wait after the first ringback before deciding that the call is not answered (in 10 msec units). If this duration is exceeded, call progress analysis is terminated with the reason CR_NOANS (no answer). Default value: 3000 (30 seconds).

7.5.4 Busy Tone Detection

Call progress analysis specifies two busy tones: TID_BUSY1 and TID_BUSY2. If either of them is detected while frequency detection and cadence detection are active, then call progress is terminated with the reason CR_BUSY.

To enable busy tone detection, turn on SIT frequency detection in the DX_CAP ca_intflg field. For details, see Section 7.4.3, "Setting Up Call Progress Analysis Parameters in DX_CAP", on page 34.

7.5.5 Fax or Modem Tone Detection

Call progress analysis specifies two tones: TID_FAX1 and TID_FAX2. If either of these tones is detected while frequency detection and cadence detection are active, then call progress is terminated with the reason CR_FAXTONE.

To enable fax or modem tone detection, use the ca_intflg field of the DX_CAP structure. For details, see Section 7.4.3, "Setting Up Call Progress Analysis Parameters in DX_CAP", on page 34.

7.5.6 SIT Frequency Detection

Special Information Tone (SIT) frequency detection is a component of call progress analysis. SIT sequences are defined as standard tone IDs.

To enable SIT frequency detection, use the ca_intflg field of the DX_CAP structure. For more information, see Section 7.4.3, "Setting Up Call Progress Analysis Parameters in DX_CAP", on page 34.

Table 4 provides default tone definitions for SIT sequences. Note the following:

- The values in the "Freq." column represent minimum and maximum values in Hz.
- Time refers to minimum and maximum on time in 10 msec units; the maximum off time between each segment is 5 (or 50 msec).
- The repeat count is 1 for all SIT segments.
- A single tone proxy for the dual tone (also called twin tone) exists for each of the three segments in a SIT sequence. The default definition for the minimum value and maximum value (in Hz) is 0. For more information on this tone, see Section 7.8.4, "Rules for Using a Single Tone Proxy for a Dual Tone", on page 44.
- N/A means "not applicable."

Table 4. Special Information Tone Sequences

SIT		1st Segment		2nd Segment		3rd Segment	
Tone ID	Description	Freq.	Time	Freq.	Time	Freq.	Time
TID_SIT_NC	No Circuit Found	950/1001	32/45	1400/1450	32/45	1740/1850	N/A
TID_SIT_IC	Operator Intercept	874/955	15/30	1310/1430	15/30	1740/1850	N/A
TID_SIT_VC	Vacant Circuit	950/1001	32/45	1310/1430	15/30	1740/1850	N/A
TID_SIT_RO	Reorder (system busy)	874/955	15/30	1400/1450	32/45	1740/1850	N/A

7.6 Media Tone Detection

Media tone detection in call progress analysis is discussed in the following topics:

- Positive Voice Detection (PVD)
- Positive Answering Machine Detection(PAMD)

7.6.1 Positive Voice Detection (PVD)

Positive voice detection (PVD) can detect when a call has been answered by determining whether an audio signal is present that has the characteristics of a live or recorded human voice. This provides a very precise method for identifying when a connect occurs.

The ca_intflg field in DX_CAP enables/disables PVD. For information on enabling PVD, see Section 7.4.3, “Setting Up Call Progress Analysis Parameters in DX_CAP”, on page 34.

PVD is especially useful in those situations where no other method of answer supervision is available, and where the cadence is not clearly broken for cadence detection to identify a connect (for example, when the nonsilence of the cadence is immediately followed by the nonsilence of speech).

If the **ATDX_CONNTYPE**( ) function returns CON_PVD, the connect was due to positive voice detection.

7.6.2 Positive Answering Machine Detection(PAMD)

Whenever PAMD is enabled, positive voice detection (PVD) is also enabled.

The ca_intflg field in DX_CAP enables/disables PAMD and PVD. For information on enabling PAMD, see Section 7.4.3, "Setting Up Call Progress Analysis Parameters in DX_CAP", on page 34.

When enabled, detection of an answering machine will result in the termination of call analysis with the reason CR_CNCT (connected); the connection type returned by the **ATDX_CONNTYPE**( ) function will be CON_PAMD.

The following DX_CAP fields govern positive answering machine detection:

ca_pamd_spdval

PAMD Speed Value: To distinguish between a greeting by a live human and one by an answering machine, use one of the following settings:

- PAMD_FULL – look at the greeting (long method). The long method looks at the full greeting to determine whether it came from a human or a machine. Using PAMD_FULL gives a very accurate determination; however, in situations where a fast decision is more important than accuracy, PAMD_QUICK might be preferred.
- PAMD_QUICK – look at connect only (quick method). The quick method examines only the events surrounding the connect time and makes a rapid judgment as to whether or not an answering machine is involved.
- PAMD_ACCU – look at the greeting (long method) and use the most accuracy for detecting an answering machine. This setting provides the most accurate evaluation. It detects live voice as accurately as PAMD_FULL but is more accurate than PAMD_FULL (although slightly slower) in detecting an answering machine. Use the setting PAMD_ACCU when accuracy is more important than speed.

Default value: PAMD_ACCU

The recommended setting for the call analysis parameter structure (DX_CAP) ca_pamd_spdval field is PAMD_ACCU.

ca_pamd_failtime

maximum time to wait for positive answering machine detection or positive voice detection after a cadence break. Default Value: 400 (in 10 msec units).

7.7 Default Call Progress Analysis Tone Definitions

Table 5 provides the range of values for default tone definitions. These default tone definitions are used in call progress analysis. Amplitudes are given in dBm, frequencies in Hz, and duration in 10 msec units. A dash in a table cell means not applicable.

Notes: ***1.*** Voice API functions are provided to manipulate the tone definitions in this table (see Section 7.8, "Modifying Default Call Progress Analysis Tone Definitions", on page 42). However, not all the functionality provided by these tones is available through the voice API. You may need to use the Global Call API to access the functionality, for example, in the case of disconnect tone detection.

2. An On Time maximum value of 0 indicates that this is a continuous tone. For example, TID_DIAL_LCL has an On Time range of 10 to 0. This means that the tone is on for 100 msecs.

The minimum requirement for detecting a tone is that it must be continuous for at least 100 msecs (10 in 10 msec units) after it is detected.

3. A single tone proxy for a dual tone (twin tone) can help improve the accuracy of dual tone detection in some cases. For more information, see Section 7.8.4, "Rules for Using a Single Tone Proxy for a Dual Tone", on page 44.

Table 5. Default Call Progress Analysis Tone Definitions

Tone ID	Freq1 (in Hz)	Freq2 (in Hz)	On Time (in 10 msec)	Off Time (in 10 msec)	Reps	Twin Tone Freq (Hz)
TID_BUSY1	450 - 510	590 - 650	30 - 100	30 - 100	2	0
TID_BUSY2	450 - 510	590 - 650	10 - 40	10 - 40	2	0
TID_DIAL_LCL	300 - 380	400 - 480	10 - 0	-	1	0
TID_DIAL_INTL	300 - 380	400 - 480	100 - 0	-	1	300 - 480
TID_DISCONNECT	360 - 410	430 - 440	30 - 60	30 - 60	1	360 - 440
TID_FAX1	1050 - 1150	-	10 - 60	-	1	-
TID_FAX2	2000 - 2300	-	10 - 0	-	1	-
TID_RNGBK1	350 - 550	350 - 550	75 - 300	0 - 800	1	350 - 550
TID_RNGBK2 (segment 0)	350 - 550	350 - 550	20 - 100	20 - 100	1	350 - 550
TID_RNGBK2 (segment 1)	350 - 550	350 - 550	20 - 100	100 - 600	1	350 - 550

7.8 Modifying Default Call Progress Analysis Tone Definitions

Call progress analysis tones are maintained in the firmware. More information on tone definitions is provided in the following topics:

- API Functions for Manipulating Tone Definitions
- TONE_DATA Data Structure
- Rules for Modifying a Tone Definition
- Rules for Using a Single Tone Proxy for a Dual Tone
- Steps to Modify a Tone Definition

7.8.1 API Functions for Manipulating Tone Definitions

The following voice API functions are used to manipulate the default tone definitions shown in Table 5, "Default Call Progress Analysis Tone Definitions", on page 42 and Table 4, "Special Information Tone Sequences", on page 40:

dx_querytone( )
gets tone information for a specific call progress tone

dx_deletetone( )
deletes a specific call progress tone

dx_createtone( )
creates a new tone definition for a specific call progress tone

7.8.2 TONE_DATA Data Structure

The TONE_DATA structure contains tone information for a specific call progress tone. This structure contains a nested array of TONE_SEG substructures. A maximum of six TONE_SEG substructures can be specified. The TONE_DATA structure specifies the following key information:

TONE_SEG.structver
Specifies the version of the TONE_SEG structure. Used to ensure that an application is binary compatible with future changes to this data structure.

TONE_SEG.tn_dflag
Specifies whether the tone is dual tone or single tone. Values are 1 for dual tone and 0 for single tone.

TONE_SEG.tn1_min
Specifies the minimum frequency in Hz for tone 1.

TONE_SEG.tn1_max
Specifies the maximum frequency in Hz for tone 1.

TONE_SEG.tn2_min
Specifies the minimum frequency in Hz for tone 2.

TONE_SEG.tn2_max
Specifies the maximum frequency in Hz for tone 2.

TONE_SEG.tn_twinmin
Specifies the minimum frequency in Hz of the single tone proxy for the dual tone.

TONE_SEG.tn_twinmax
Specifies the maximum frequency in Hz of the single tone proxy for the dual tone.

TONE_SEG.tnon_min
Specifies the debounce minimum ON time in 10 msec units.

TONE_SEG.tnon_max
Specifies the debounce maximum ON time in 10 msec units.

TONE_SEG.tnoff_min
Specifies the debounce minimum OFF time in 10 msec units.

TONE_SEG.tnoff_max
Specifies the debounce maximum OFF time in 10 msec units.

TONE_DATA.structver
Specifies the version of the TONE_DATA structure. Used to ensure that an application is binary compatible with future changes to this data structure.

TONE_DATA.tn_rep_cnt
Specifies the debounce rep count.

TONE_DATA.numofseg
Specifies the number of segments for a multi-segment tone.

7.8.3 Rules for Modifying a Tone Definition

Consider the following rules and guidelines for modifying default tone definitions using the voice API library:

- You must issue **dx_querytone( )**, **dx_deletetone( )**, and **dx_createtone( )** in this order, one tone at a time, for each tone definition to be modified.
- Attempting to create a new tone definition before deleting the current call progress tone will result in an EDX_TNQUERYDELETE error.
- When **dx_querytone( )**, **dx_deletetone( )**, or **dx_createtone( )** is issued in asynchronous mode and is immediately followed by another similar call prior to completion of the previous call on the same device, the subsequent call will fail with device busy.
- Only default call progress analysis tones and SIT sequences are supported for these three functions. For a list of these tones, see Table 4, “Special Information Tone Sequences”, on page 40 and Table 5, “Default Call Progress Analysis Tone Definitions”, on page 42.
- These three voice API functions are provided to manipulate the call progress analysis tone definitions. However, not all the functionality provided by these tones is available through the voice API. You may need to use the Global Call API to access the functionality, for example, in the case of disconnect tone detection.
- If the application deletes all the default call progress analysis tones in a particular set (where a set is defined as busy tones, dial tones, ringback tones, fax tones, disconnect tone, and special information tones), the set itself is deleted from the board and call progress analysis cannot be performed successfully. Therefore, you must have at least one tone defined in each tone set in order for call progress analysis to perform successfully.

7.8.4 Rules for Using a Single Tone Proxy for a Dual Tone

A single tone proxy (also called a twin tone) acts as a proxy for a dual tone. A single tone proxy can be defined when you run into difficulty detecting a dual tone. This situation can arise when the two frequencies of the dual tone are close together, are very short tones, or are even multiples of each other. In these cases, the dual tone might be detected as a single tone. A single tone proxy can help improve the detection of the dual tone by providing an additional tone definition.

The TONE_SEG.tn_twinmin field defines the minimum frequency of the tone and TONE_SEG.tn_twinmax field defines the maximum frequency of the tone.

Consider the following guidelines when creating a single tone proxy:

- It is recommended that you add at least 60 Hz to the top of the dual tone range and subtract at least 60 Hz from the bottom of the dual tone range. For example:

  Freq1 (Hz): 400 - 500

  Freq 2 (Hz): 600 - 700

  Twin tone freq (Hz): 340 - 760

- Before using the TONE_DATA structure in a function call, set any unused fields in the structure to zero to prevent possible corruption of data in the allocated memory space. This guideline is applicable to unused fields in any data structure.

7.8.5 Steps to Modify a Tone Definition

To modify a default tone definition using the voice API library, follow these steps:

Note: This procedure assumes that you have already opened the board device handle in your application.

1. Get the tone information for the call progress tone to be modified using **dx_querytone( )**. After the function completes successfully, the relevant tone information is contained in the TONE_DATA structure.
2. Delete the current call progress tone using **dx_deletetone( )** before creating a new tone definition.
3. Create a new tone definition for the call progress tone using **dx_createtone( )**. Specify the new tone information in the TONE_DATA structure.
4. Repeat steps 1-3 in this order **for each tone** to be modified.

7.9 Call Progress Analysis Errors

If **ATDX_CPTERM( )** returns CR_ERROR, you can use **ATDX_CPERROR( )** to determine the call progress analysis error that occurred. For details on these functions, see the *Voice API Library Reference*.

Recording and Playback 8

This chapter discusses playback and recording features supported by the voice library. The following topics are discussed:

8.1 Overview of Recording and Playback

The primary voice processing operations provided by a voice board include:

- recording: digitizing and storing human voice
- playback: retrieving, converting, and playing the stored, digital information to reconstruct the human voice.

The following features related to voice recording and playback operation are documented in other chapters in this document:

- Controlling when a playback or recording terminates using I/O termination conditions is documented in Section 6.1.2, "Setting Termination Conditions for I/O Functions", on page 25.
- Controlling the volume when messages are played back is documented in Chapter 9, "Volume Control".

8.2 Digital Recording and Playback

In digital speech recording, the voice board converts the human voice from a continuous sound wave, or analog signal, into a digital representation. The voice board does this by frequently sampling the amplitude of the sound wave at individual points in the speech signal.

The accuracy, and thus the quality, of the digital recording is affected by:

- the sampling rate (number of samples per second), also called digitization rate
- the precision, or resolution, of each sample (the amount of data that is used to represent 1 sample).

If the samples are taken at a greater frequency, the digital representation will be more accurate and the voice quality will be greater. Likewise, if more bits are used to represent the sample (higher resolution), the sample will be more accurate and the voice quality will be greater.

In digital speech playback, the voice board reconstructs the speech signal by converting the digitized voice back into analog voltages. If the voice data is played back at the same rate at which it was recorded, an approximation of the original speech will result.

8.3 Play and Record Functions

The C language function library includes several functions for recording and playing audio data, such as **dx_rec( )**, **dx_reciottdata( )**, **dx_play( )**, and **dx_playiottdata( )**. Recording takes audio data from a specified channel and encodes it for storage in memory, in a file on disk, or on a custom device. Playing decodes the stored audio data and plays it on the specified channel. The storage location is one factor in determining which record and play functions should be used. The storage location affects the access speed for retrieving and storing audio data.

One or more of the following data structures are used in conjunction with certain play and record functions: DV_TPT to specify a termination condition for the function, DX_IOTT to identify a source or destination for the data, and DX_XPB to specify the file format, data format, sampling rate, and resolution.

For detailed information about play and record functions, which are also known as I/O functions, see the *Voice API Library Reference*.

8.4 Play and Record Convenience Functions

Several convenience functions are provided to make it easier to implement play and record functionality in an application. Some examples are: **dx_playf( )**, **dx_playvox( )**, **dx_playwav( )**, **dx_recf( )**, and **dx_recvox( )**. These functions are specific cases of the **dx_play( )** and **dx_rec( )** functions and run in synchronous mode.

For example, **dx_playf( )** performs a playback from a single file by specifying the filename. The same operation can be done using **dx_play( )** and specifying a DX_IOTT structure with only one entry for that file. Using **dx_playf( )** is more convenient for a single file playback because you do not have to set up a DX_IOTT structure for the one file and the application does not need to open the file. **dx_recf( )** provides the same single file convenience for the **dx_rec( )** function.

For a complete list of I/O convenience functions and function reference information, see the *Voice API Library Reference*.

8.5 Voice Encoding Methods

A digitized audio recording is characterized by several parameters as follows:

- the number of samples per second, or sampling rate
- the number of bits used to store a sample, or resolution
- the rate at which data is recorded or played

There are many encoding and storage schemes available for digitized voice. The supported voice encoding methods or data formats are listed in Table 6.

Table 6. Voice Encoding Methods

Digitizing Method	Sampling Rate (kHz)	Resolution (Bits)	Bit Rate (Kbps)	File Format
OKI ADPCM	6	4	24	VOX, WAVE
OKI ADPCM	8	4	32	VOX, WAVE
G.711 PCM A-law and mu-law	6	8	48	VOX, WAVE
G.711 PCM A-law and mu-law	8	8	64	VOX, WAVE
Linear PCM	11	8	88	VOX, WAVE
G.726 bit exact	8	2	16	VOX, WAVE
G.726 bit exact	8	4	32	VOX, WAVE

8.6 G.726 Voice Coder

G.726 is an ITU-T recommendation that specifies an adaptive differential pulse code modulation (ADPCM) technique for recording and playing back audio files. It is useful for applications that require speech compression, encoding for noise immunity, and uniformity in transmitting voice and data signals.

The voice library provides support for a G.726 bit exact voice coder that is compliant with the ITU-T G.726 recommendation.

Audio encoded in the G.726 bit exact format complies with Voice Profile for Internet Messaging (VPIM), a communications protocol that makes it possible to send and receive messages from disparate messaging systems over the Internet. G.726 bit exact is the audio encoding and decoding standard supported by VPIM.

VPIM follows the little endian ordering. The 4-bit code words of the G.726 encoding must be packed into octets/bytes as follows:

- The first code word (A) is placed in the four least significant bits of the first octet, with the least significant bit (LSB) of the code word (A0) in the least significant bit of the octet.

- The second code word (B) is placed in the four most significant bits of the first octet, with the most significant bit (MSB) of the code word (B3) in the most significant bit of the octet.
- Subsequent pairs of the code words are packed in the same way into successive octets, with the first code word of each pair placed in the least significant four bits of the octet. It is preferable to extend the voice sample with silence such that the encoded value consists of an even number of code words. However, if the voice sample consists of an odd number of code words, then the last code word will be discarded.

The G.726 encoding for VPIM is illustrated here:

```
        +--+--+--+--+--+--+--+--+
        |B3|B2|B1|B0|A3|A2|A1|A0|
        +--+--+--+--+--+--+--+--+
MSB -> | 7| 6| 5| 4| 3| 2| 1| 0|  <- LSB
        +--+--+--+--+--+--+--+--+
        32K ADPCM / Octet Mapping
```

For more information on G.726 and VPIM, see RFC 2422 on the Internet Engineering Task Force website at http://www.ietf.org.

To use the G.726 voice coder, specify the coder in the DX_XPB structure. Then use **dx_playiottdata( )** and **dx_reciottdata( )** functions to play and record with this coder. Alternatively, you can also use **dx_playvox( )** and **dx_recvox( )** convenience functions.

To determine the voice resource handles used with the play and record functions, use SRL device mapper functions to return information about the structure of the system.

See the *Voice API Library Reference* for more information on voice functions and data structures. See the *Standard Runtime Library API Library Reference* for more information on SRL functions.

8.7 Streaming to Board

The streaming to board feature is discussed in the following topics:

- Streaming to Board Overview
- Streaming to Board Functions
- Implementing Streaming to Board
- Streaming to Board Hints and Tips

8.7.1 Streaming to Board Overview

The streaming to board feature provides a way to stream data in real time to a network interface. Unlike the standard voice play feature (store and forward method), data can be streamed with little delay as the amount of initial data required to start the stream is configurable. The streaming to board feature is essential for applications such as text-to-speech, distributed prompt servers, and IP gateways.

The streaming to board feature uses a circular stream buffer to hold data, provides configurable high and low water mark parameters, and generates events when those water marks are reached.

8.7.2 Streaming to Board Functions

The following functions are used by the streaming to board feature:

dx_OpenStreamBuffer( )
creates and initializes a circular stream buffer

dx_SetWaterMark( )
sets high and low water marks for the circular stream buffer

dx_PutStreamData( )
places data into the circular stream buffer

dx_GetStreamInfo( )
retrieves information about the circular stream buffer

dx_ResetStreamBuffer( )
resets internal data for a circular stream buffer

dx_CloseStreamBuffer( )
deletes a circular stream buffer

For details on these functions, see the *Voice API Library Reference*.

8.7.3 Implementing Streaming to Board

Perform the following steps to implement streaming to board in your application:

Note: These steps do not represent every task that must be performed to create a working application but are intended as general guidelines for implementing streaming to board.

1. Decide on the size of the circular stream buffer. This value is used as input to the **dx_OpenStreamBuffer( )** function. To determine the circular stream buffer size, see Section 8.7.4, "Streaming to Board Hints and Tips", on page 52.
2. Based on the circular stream buffer and the bulk queue buffer size, decide on values for the high and low water marks for the circular stream buffer. To determine high and low water mark values, see Section 8.7.4, "Streaming to Board Hints and Tips", on page 52.
3. Initialize and create a circular stream buffer using **dx_OpenStreamBuffer( )**.
4. Set the high and low water marks using **dx_SetWaterMark( )**.
5. Start the play using **dx_playiottdata( )** or **dx_play( )** in asynchronous mode with the io_type field in DX_IOTT data structure set to IO_STREAM.
6. Put data in the circular stream buffer using **dx_PutStreamData( )**.
7. Wait for events.

   The TDX_LOWWATER event is generated every time data in the buffer falls below the low water mark. The TDX_HIGHWATER event is generated every time data in the buffer is above the high water mark. The application receives TDX_LOWWATER and TDX_HIGHWATER events regardless of whether or not **dx_SetWaterMark( )** is used in your application. These

events are generated when there is a play operation with this buffer and are reported on the device that is performing the play. If there is no active play, the application will **not** receive any of these events.

TDX_PLAY indicates that play has completed.

8. When all files are played, issue **dx_CloseStreamBuffer( )**.

8.7.4 Streaming to Board Hints and Tips

Consider the following usage guidelines when implementing streaming to board in your application:

- You can create as many circular stream buffers as needed on a channel. You can use more than one circular stream buffer per play via the DX_IOTT structure. In this case, specify that the data ends in one buffer using the STREAM_EOD flag so that the play can process the next DX_IOTT structure in the chain.
- The bulk queue buffer specifies the size of the buffer used to transfer voice data between the application and HMP. This buffer is set to 32 kbytes and cannot be modified; the **dx_setchxfercnt( )** function, which is used to modify the bulk queue buffer size, is not currently supported.
- In general, the larger you define the circular stream buffer size, the better. Factors to take into consideration include the average input file size, the amount of memory on your system, the total number of channels in your system, and so on. Having an optimal circular stream buffer size results in the high and low water marks being reached less often. In a well-tuned system, the high and low water marks should rarely be reached.
- When adjusting circular stream buffer sizes, be aware that you must also adjust the high and low water marks accordingly.
- Recommendation for the high water mark: it should be based on the following:

  (size of the circular stream buffer) minus (two times the size of the bulk queue buffer)

  For example, if the circular stream buffer is 500 kbytes, and the bulk queue buffer size is 32 kbytes, set the high water mark to 436 kbytes (500-64=436).
- Recommendation for the low water mark: it should be two times the size of the bulk queue buffer size.

  Based on the previous example, since the bulk queue buffer size is 32 kbytes, set the low water mark to 64 kbytes.
- When a TDX_LOWWATER event is received, continue putting data in the circular stream buffer. Remember to set STREAM_EOD flag to EOD on the last piece of data.
- When a TDX_HIGHWATER event is received, stop putting data in the circular stream buffer. If using a text-to-speech (TTS) engine, you will have to stop the engine from sending more data. If you cannot control the output of the TTS engine, you will need to control the input to the engine.
- It is recommended that you enable the TDX_UNDERRUN event to notify the application of firmware underrun conditions on the board. Specify DM_UNDERRUN in **dx_setevtmsk( )**.

Volume Control 9

This chapter describes how to control the volume of play on a channel. The following topics are discussed:

9.1 Volume Control Overview

The voice software contains functions and data structures to control the volume of play on a channel. This allows an end user to control the volume of a message by entering a DTMF tone, for example.

Volume can be controlled on all playbacks regardless of the encoding algorithm. For a list of supported encoding methods, see Section 8.5, "Voice Encoding Methods", on page 49.

9.2 Volume Convenience Function

The convenience function sets a digit that will adjust volume, but does not use any data structures. This convenience function will only function properly if you use the default settings of the volume modification table. The function assumes that the modification table has not been modified. The convenience function is:

dx_addvoldig( )
adds a digit that will modify volume by a specified amount

See the *Voice API Library Reference* for detailed information about this function.

9.3 Volume Adjustment Functions

Volume can be adjusted explicitly or can be set to adjust in response to a preset condition, such as a specific digit. For example, volume could be set to increase a certain amount when "1" is pressed on the telephone keypad. The functions used for volume adjustment are:

Note: These functions are used to adjust both speed and volume. On Intel® NetStructure™ Host Media Processing (HMP) software, speed adjustment is not supported.

dx_setsvcond( )
Sets conditions that adjust volume. Use this function to adjust volume in response to a DTMF digit or start of play.

dx_adjsv( )
Adjusts volume explicitly. Use this function if your adjustment condition is not a digit or start of play. For example, the application could call this function after detecting a spoken word (voice recognition) or a certain key on the keyboard.

See the *Voice API Library Reference* for detailed information about these functions.

9.4 Volume Modification Tables

Each channel has a volume modification table for play volume adjustments.

Each volume modification table (SVMT) table has 21 entries, 20 that allow for a maximum of 10 increases and decreases in volume. The entry in the middle of the table is referred to as the "origin" entry that represents normal volume. The normal volume is how playback occurs when the volume control feature is not used. See Table 7, "Default Volume Modification Table", on page 56.

The origin, or normal volume, is the basis for all settings in the table. Typically, the origin is set to 0. Volume increases or decreases by moving up or down the tables. Other entries in the table specify a volume setting in terms of a deviation from normal. For example, if a volume modification table entry is -10, this value represents a 10% decrease from the normal volume.

Although the origin is typically set to normal volume, changing the setting of the origin does not affect the other settings, because all values in the SVMT are based on a deviation from normal volume.

Volume control adjustments are specified by moving the current volume pointer in the table to another SVMT table entry; this translates to increasing or decreasing the current volume to the value specified in the table entry.

A volume adjustment stays in effect until the next adjustment on that channel or until a system reset.

The SVMT is like a 21-speed bicycle. You can select the gear ratio for each of the 21 speeds before you go for a ride (by changing the values in the SVMT). And you can select any gear once you are on the bike, like adjusting the current volume to any setting in the SVMT.

To change the default values of the volume modification table, use the **dx_setsvmt( )** function, which in turn uses the DX_SVMT data structure. To return the current values of a table to the DX_SVMT structure, use **dx_getsvmt( )**. The DX_SVCB data structure uses this table when setting adjustment conditions.

Note: These functions adjust both speed and volume. On HMP, speed adjustment is not supported.

See the *Voice API Library Reference* for detailed information about these functions and data structures.

Adjustments to volume are made by **dx_adjsv( )** and **dx_setsvcond( )** according to the volume modification table settings. These functions adjust volume to one of the following:

- a specified level (that is, to a specified absolute position in the table)
- a change in level (that is, by a specified number of steps up or down in the table)

For example, by default, each entry in the volume modification table is equivalent to 2 dB from the origin. Volume could be decreased by 2 dB by specifying position 1 in the table, or by moving one step down from the origin.

The default volume modification table is shown in Table 7.

Table 7. Default Volume Modification Table

Table Entry	Default Value (dB)	Absolute Position
decrease[0]	-20	-10
decrease[1]	-18	-9
decrease[2]	-16	-8
decrease[3]	-14	-7
decrease[4]	-12	-6
decrease[5]	-10	-5
decrease[6]	-08	-4
decrease[7]	-06	-3
decrease[8]	-04	-2
decrease[9]	-02	-1
origin	0	0
increase[0]	+02	1
increase[1]	+04	2
increase[2]	+06	3
increase[3]	+08	4
increase[4]	+10	5
increase[5]	-128 (80h)	6
increase[6]	-128 (80h)	7
increase[7]	-128 (80h)	8
increase[8]	-128 (80h)	9
increase[9]	-128 (80h)	10

Consider the following usage information on the volume modification table:

- Each entry in the table is a deviation in decibels from the starting point or volume (“origin”). Each entry in the table is equivalent to 2 dB from the origin. Volume can be decreased 2 dB by specifying position 1 in the table, or by moving one step down. For example, the increase[1] position (two steps from the origin) increases volume by 4 dB.
- The total volume modification range is from -20 dB to +10 dB. In this table, the highest position utilized is the increase[4] position. The remaining increase fields are set to -128 (80h). If these “non-adjustment” positions are selected, the default action is to play at the increase[4] volume. These fields can be reset, as long as no values higher than +10 are used; for example, you could spread the 10 dB volume increase over 10 steps rather than 5.
- In the volume modification table, the default entries for index values +6 to +10 are -128 which represent a null-entry. To customize the table entries, you must use the **dx_setsvmt( )** function.
- When adjustment is associated with a DTMF digit, volume can be increased or decreased in increments of 1 (2 dB) only. To achieve an increase in volume of 6 dB for example, the user would press the DTMF digit three times.

9.5 Play Adjustment Digits

The voice software processes play adjustment digits differently from normal digits:

- If a play adjustment digit is entered during playback, it causes a play adjustment only and has no other effect. This means that the digit is not added to the digit queue; it cannot be retrieved with the **dx_getdig( )** function.
- Digits that are used for play adjustment may also be used as a terminating condition. If a digit is defined as both, then both actions are applied upon detection of that digit.
- If the digit queue contains adjustment digits when a play begins and play adjustment is set to be level sensitive, the digits will affect the volume and then be removed from the queue.

9.6 Setting Play Adjustment Conditions

The following steps describe how to set conditions upon which volume should be adjusted:

1. Set up the volume modification table (if you do not want to use the defaults):
   - Set up the DX_SVMT structure to specify the size and number of the steps in the table.
   - Call the **dx_setsvmt( )** function, which points to the DX_SVMT structure, to modify the volume modification table (**dx_setsvmt( )** can also be used to reset the table to its default values).
2. Set up the DX_SVCB structure to specify the condition, the size, and the type of adjustment.
3. Call **dx_setsvcond( )**, which points to an array of DX_SVCB structures. All subsequent plays will adjust volume as required whenever one of the conditions specified in the array occurs.

See the *Voice API Library Reference* for more information about these functions and data structures.

9.7 Explicitly Adjusting Volume

The following steps describe how to adjust volume:

1. Set up the volume modification table (if you do not want to use the defaults):
   - Set up the DX_SVMT structure to specify the size and number of the steps in the table.
   - Call the **dx_setsvmt( )** function, which points to the DX_SVMT structure, to modify the volume modification table (**dx_setsvmt( )** can also be used to reset the table to its default values).
2. When required, call **dx_adjsv( )** to adjust the volume modification table by specifying the size and type of the adjustment.

See the *Voice API Library Reference* for more information about these functions and data structures.

Global Tone Detection and Generation, and Cadenced Tone Generation

10

This chapter discusses global tone detection (GTD), global tone generation (GTG), and cadenced tone generation:

10.1 Global Tone Detection (GTD)

Global tone detection (GTD) is described in the following sections:

- Overview of Global Tone Detection
- Defining Global Tone Detection Tones
- Building Tone Templates
- Working with Tone Templates
- Retrieving Tone Events
- Setting GTD Tones as Termination Conditions
- Estimating Memory
- Guidelines for Creating User-Defined Tones
- Global Tone Detection Applications

10.1.1 Overview of Global Tone Detection

Global tone detection (GTD) allows you to define the characteristics of a tone in order to detect a tone with these same characteristics. The characteristics of a tone are defined using GTD tone templates. The tone templates contain parameters that allow you to assign frequency bounds and cadence components. GTD can detect single- and dual-frequency tones by comparing all incoming sounds to the GTD tone templates.

Global tone detection and GTD tones are also known as **user-defined tone detection** and **user-defined tones**.

GTD operates on a channel-by-channel basis and is active when the channel is off-hook, unless the system uses a DTI/xxx board, in which case GTD is always active. GTD works simultaneously with DTMF and MF tone detection.

The driver responds to a detected tone by producing either a **tone event** (DE_TONEON or DE_TONEOFF) on the event queue or a **digit** on the digit queue. The particular response depends upon the GTD tone configuration.

Use the global tone detection functions to access tone templates and enable detection of single- and dual-frequency tones that fall outside of those automatically provided with the voice driver. This includes tones outside the standard DTMF set of 0-9, a-d, *, and #, and the standard MF tones 0-9, *, and a-c.

10.1.2 Defining Global Tone Detection Tones

GTD tones can have an associated ASCII digit (and digit type) specified using the **digit** and **digtype** parameters in the **dx_addtone( )** function. When the tone is detected, the digit is placed in the DV_DIGIT buffer and can be retrieved using the **dx_getdig( )** function. When the tone is detected, either the tone event or the digit associated with the tone can be used as a termination condition to terminate I/O functions.

Termination conditions are set using the DV_TPT data structure. To terminate on multiple tones (or digits), you must specify the terminating conditions for each tone in a separate DV_TPT data structure.

The functions and data structures associated with global tone detection are described in the *Voice API Library Reference*.

10.1.3 Building Tone Templates

When creating a tone template, you can define the following:

- single-frequency or dual-frequency (300 - 3500 Hz)
- optional ASCII digit associated with the tone template
- cadence components

Adding a tone template to a channel enables detection of a tone on that channel. Although only one tone template can be created at a time, multiple tone templates can be added to a channel. Each channel can have a different set of tone templates. Once created, tone templates can be selectively enabled or disabled.

Note: A particular tone template cannot be changed or deleted. A tone template can be disabled on a channel, but to delete a tone template, all tone templates on that channel must be deleted.

The following functions are used to build and define tone templates:

dx_bldst( )
Defines a single-frequency tone. Subsequent calls to **dx_addtone( )** will use this tone until another tone is defined. Thus, you can build a tone template and add it to several different channels.

dx_blddt( )
Defines a simple dual-frequency tone. Subsequent calls to **dx_addtone( )** will use this tone until another tone is defined. Thus, you can build a tone template and add it to several different channels.

Note that the boards cannnot detect dual tones with frequency components closer than approximately 63 Hz. Use a single tone description to detect dual tones that are closer together than the ranges specified above.

dx_bldstcad( )
Defines a simple single-frequency cadence tone. Subsequent calls to **dx_addtone( )** will use this tone until another tone is defined. Thus, you can build a tone template and add it to several different channels. A single-frequency cadence tone has single-frequency signals with specific on/off characteristics.

dx_blddtcad( )
Defines a simple dual-frequency cadence tone. Subsequent calls to **dx_addtone( )** will use this tone until another tone is defined. Thus, you can build a tone template and add it to several different channels. A dual-frequency cadence tone has dual-frequency signals with specific on/off characteristics.

The minimum on- and off-time for cadence detection is 40 msec on and 40 msec off.

dx_setgtdamp( )
Sets the amplitudes used by GTD. The amplitudes set using **dx_setgtdamp( )** are the default amplitudes that apply to all tones built using the **dx_bld...( )** functions. The amplitudes remain valid for all tones built until **dx_setgtdamp( )** is called again and amplitudes are changed.

Notes: ***1.*** GTD build functions define new tone templates, and **dx_addtone( )** adds the tone templates to a channel.

2. Use **dx_addtone( )** to enable detection of the tone template on a channel.

3. After building a tone template using a **dx_bld...( )** function, **dx_addtone( )** must be called to add this tone template to a channel. If the template is not added, the next call to a **dx_bld...( )** function will overwrite the tone definition contained in the previous template.

Table 8 lists some standard Bell System Network call progress tones. The frequencies are useful when creating the tone templates.

Table 8. Standard Bell System Network Call Progress Tones

Tone	Frequency (Hz)	On Time (msec)	Off Time (msec)
Dial	350 + 440	Continuous	
Busy	480 + 620	500	500
Congestion (Toll)	480 + 620	200	300
Reorder (Local)	480 + 620	300	200
Ringback	440 + 480	2000	4000

10.1.4 Working with Tone Templates

Use the following functions to add/delete tone templates or to enable/disable tone detection:

dx_addtone( )
Adds a tone template that was defined by the most recent GTD build-tone function call to the specified channel. Adding a tone template to a channel downloads it to the board and enables detection of tone-on and tone-off events for that tone template.

dx_deltones( )
Removes all tone templates previously added to a channel with **dx_addtone( )**. If no tone templates were previously enabled for this channel, the function has no effect.

dx_deltones( ) does not affect tones defined by build-tone template functions and tone templates not yet defined. If you have added tones for call progress analysis, these tones are also deleted.

dx_distone( )
Disables the detection of DE_TONEON and/or DE_TONEOFF events on a channel. Detection capability for user-defined tones is enabled on a channel by default when **dx_addtone( )** is called.

dx_enbtone( )
Enables the detection of DE_TONEON and/or DE_TONEOFF events on a channel. Detection capability for tones is enabled on a channel by default when **dx_addtone( )** is called. The function can re-enable tones disabled by **dx_distone( )**. DE_TONEON and DE_TONEOFF events are call status transition (CST) events.

Caution: Each tone template must have a unique identification.

Caution: Errors will occur if you use **dx_addtone( )** to change a tone template that has previously been added.

10.1.5 Retrieving Tone Events

Tone-on and tone-off events are call status transition (CST) events. Retrieval of these events is handled differently for asynchronous and synchronous applications. Table 9 outlines the different processes for retrieving tone events.

Table 9. Asynchronous/Synchronous Tone Event Handling

Synchronous	Asynchronous
Call **dx_addtone( )** or **dx_enbtone( )** to enable tone-on/off detection.	Call **dx_addtone( )** or **dx_enbtone( )** to enable tone-on/off detection.
Call **dx_getevt( )** to wait for CST event(s). Events are returned in the DX_EBLK data structure.	Use Standard Runtime Library (SRL) to asynchronously wait for TDX_CST event(s).
N/A	Use **sr_getevtdatap( )** to retrieve DX_CST data structure.
Note: These procedures are the same as the retrieval of any other CST event, except that **dx_addtone( )** or **dx_enbtone( )** are used to enable event detection instead of **dx_setevtmsk( )**.	

You can optionally specify an associated ASCII digit (and digit type) with the tone template. In this case, the tone template is treated like DTMF tones. When the digit is detected, it is placed in the digit buffer and can be used for termination. When an associated ASCII digit is specified, tone events will not be generated for that tone.

Cadence tone on events are reported differently on HMP versus Springware boards. On HMP, if a cadence tone occurs continuously, a DE_TONEON event is reported for each on/off cycle. On Springware boards, a DE_TONEON event is reported for the first on/off cycle only. On HMP and on Springware boards, a DE_TONEOFF event is reported when the tone is no longer present.

10.1.6 Setting GTD Tones as Termination Conditions

To detect a GTD (user-defined) tone, you can specify it as a termination condition for I/O functions. Set the tp_termno field in the DV_TPT structure to DX_TONE, and specify DX_TONEON or DX_TONEOFF in the tp_data field.

10.1.7 Estimating Memory

Refer to the following guidelines to estimate the memory used for each tone on each channel.

Calculate the total frequency range covered by the tone. For single tones, this is twice the deviation (unless the range is truncated by the GTD detection range); for dual tones, this is twice the deviation of **each** of the two tones minus any overlap (and minus any truncation by the GTD detection range).

For example:

- Single Tone: 400 Hz (125 Hz deviation) = bandwidth of 275 to 525 Hz, total of 250 Hz.
- Dual Tone: 450 Hz (50 Hz deviation) and 1000 Hz (75 Hz deviation) = bandwidth of 400 to 500 Hz and 925 to 1075 Hz, total of 250 Hz.
- Dual Tone: 450 Hz (100 Hz deviation) and 600 Hz (100 Hz deviation) = bandwidth of 350 to 550 Hz and 500 to 700 Hz; eliminating overlap, total range = 350 to 700 Hz, total of 350 Hz.

Each tone costs, on average, 1 + round up [1/16 * round up (total frequency range /62.5)].

This allows for:

- 1 memory block for the actual template
- 1/16 portion of a memory block for an index entry
- range/62.5 multiple indexing for speed

In practice, the 1/16 should be added across tones that have frequency overlap, rather than rounded up for each tone.

Note: The firmware will often use memory more efficiently than described by the guidelines given above. These guidelines estimate the maximum memory usage for user-defined tones; the actual usage may be lower.

10.1.8 Guidelines for Creating User-Defined Tones

When creating user-defined tones, keep the following guidelines in mind:

Note: The terms "user-defined tones" and "tone templates" are used interchangeably. Each tone template specifies the characteristics of one user-defined tone.

- The maximum number of tone templates is based on tone templates that define a *dual* tone with a frequency range (bandwidth) of 63 Hz. (The detection range is the difference between the minimum and maximum defined frequencies for the tone.)
- The tone detection range should be limited to a maximum of 200 Hz per tone to reduce the chance of exceeding the available memory.
- The number of tone templates which can be added to a voice device and enabled for detection is limited. The default maximum number of events for each instance is 20. This number can be increased using the SD_ParmMaxSigsPerInst parameter. By increasing this number, the memory usage will increase. For more information on this parameter, see the *Configuration Guide* for your product family.

  Also, for single tone definition and dual tone definition, the frequency deviation that is defined in the tone template for each frequency must be no less than +/- 30 Hz.
- If you use call progress analysis to detect the different call progress signals (dial tone, busy tone, ringback tone, fax or modem tone), call progress analysis creates GTD tones. This reduces the number of user-defined tones you can create. Call progress analysis creates 8 GTD tones; this uses 17 memory blocks on each channel on which it is activated.
- If you use call progress analysis to identify tri-tone special information tone (SIT) sequences, call progress analysis will create GTD tones internally, and this reduces the number of user-defined tones you can create. Call progress analysis creates one GTD tone (or tone template) for each single-frequency tone that you define in the DX_CAP structure. For example, if detecting the SIT tri-tone sequences per channel (3 frequencies), the GTD memory will be reduced by five blocks.
- If you initiate call progress analysis and there is not enough memory to create the GTD tones, you will get an EDX_MAXTMPLT error. This indicates that you are trying to exceed the maximum number of GTD tones.

- If you use **dx_blddt( )** (or one of the **dx_bld...( )** build-tone functions) to define a user-defined tone that alone or with existing user-defined tones exceeds the available memory, you will get an EDX_MAXTMPLT error.
- The **dx_deltones( )** function deletes all user-defined tones from a specified channel and releases the memory that was used for those user-defined tones. When an associated ASCII digit is specified, tone events will not be generated for that tone.
- On Linux, call progress analysis SIT detection releases GTD memory when call progress analysis has completed. The other features do not release GTD memory until a **dx_deltones( )** is performed.
- On Linux, if you initiate call progress analysis and there is not enough memory to create the SIT tones internally, you will get a T_CAERROR event and the **evtdata** field will contain MEMERR.
- On Windows, if you initiate call progress analysis and there is not enough memory to create the SIT tones internally, you will get a CR_MEMERR.

10.1.9 Global Tone Detection Applications

Sample applications for global tone detection (GTD) are described in the following sections:

- Detecting Disconnect Tone
- Detecting Leading Edge Debounce Time

10.1.9.1 Detecting Disconnect Tone

Global tone detection can be used to detect a disconnect tone or fast-busy signal.

Perform the following to detect the signal:

1. Determine the frequencies of the signal.
2. Characterize the on/off durations and tolerances of the signal cadence.
3. Use a build-tone function to define the characteristics of a single or dual tone with cadence in a tone template.
4. Use the **dx_addtone( )** function to add the GTD tone template for global tone detection on each channel.

10.1.9.2 Detecting Leading Edge Debounce Time

Rather than detecting a signal immediately, an application may want to wait for a period of time (debounce time) before the DE_TONEON event is generated indicating the detection of the signal. The **dx_bldstcad( )** and **dx_blddtcad( )** functions can detect leading edge debounce on-time. A tone must be present at a given frequency and for a period of time (debounce time) before a DE_TONEON event is generated. The debounce time is specified using the ontime and ondev parameters in the **dx_bldstcad( )** or **dx_blddtcad( )** functions.

To use this application, specify the following values for the **dx_bldstcad( )** or **dx_blddtcad( )** function parameters listed below:

- For **ontime**, specify 1/2 of the desired debounce time
- For **ondev**, specify -1/2 of the desired debounce time
- For **offtime**, specify 0
- For **offdev**, specify 0
- For **repcnt**, specify 0

Note: This application cannot work with the functions **dx_blddt( )** and **dx_bldst( )** since these functions do not have timing field parameters.

10.2 Global Tone Generation (GTG)

The following topics provide information on using global tone generation:

- Using GTG
- GTG Functions
- Building and Implementing a Tone Generation Template

10.2.1 Using GTG

Global tone generation enables the creation of user-defined tones. The tone generation template, TN_GEN, is used to define the tones with the following information:

- Single or dual tone
- Frequency fields
- Amplitude for each frequency
- Duration of tone

The functions and data structures associated with global tone generation are described in the *Voice API Library Reference*.

10.2.2 GTG Functions

The following functions are used to generate tones:

dx_bldtngen( )
Builds a tone generation template. This convenience function sets up the tone generation template data structure (TN_GEN) by allowing the assignment of specified values to the appropriate fields. The tone generation template is placed in the user's return buffer and can then be used by the **dx_playtone( )** function to generate the tone.

dx_playtone( )
Plays a tone specified by the tone generation template (pointed to by **tngenp**). Termination conditions are set using the DV_TPT structure. The reason for termination is returned by the

ATDX_TERMMSK( ) function. **dx_playtone( )** returns a 0 to indicate that it has completed successfully.

10.2.3 Building and Implementing a Tone Generation Template

The tone generation template defines the frequency, amplitude, and duration of a single- or dual-frequency tone to be played. You can use the convenience function **dx_bldtngen( )** to set up the structure. Use **dx_playtone( )** to play the tone.

The TN_GEN data structure is defined as:

```
typedef struct {
   unsigned short tg_dflag;        /* dual tone = 1, single tone = 0 */
   unsigned short tg_freq1;        /* frequency of tone 1 (in Hz) */
   unsigned short tg_freq2;        /* frequency of tone 2 (in Hz) */
   short int      tg_ampl1;        /* amplitude of tone 1 (in dB) */
   short int      tg_ampl2;        /* amplitude of tone 2 (in dB) */
   short int      tg_dur;          /* duration (in 10 msec units) */
} TN_GEN;
```

After you build the TN_GEN data structure, there are two ways to define each tone template. You may either:

- Include the values in the structure
- Pass the values to TN_GEN using the **dx_bldtngen( )** function

If you include the values in the structure, you must create a structure for each tone template. If you pass the values using the **dx_playtone( )** function, then you can reuse the structure. If you are only changing one value in a template with many variables, it may be more convenient to use several structures in the code instead of reusing just one.

After defining the template by either of these methods, pass TN_GEN to **dx_playtone( )** to play the tone.

10.3 Cadenced Tone Generation

The following topics provide information on enabling and using cadenced tone generation:

- Using Cadenced Tone Generation
- How To Generate a Custom Cadenced Tone
- How To Generate a Non-Cadenced Tone
- TN_GENCAD Data Structure - Cadenced Tone Generation
- How To Generate a Standard PBX Call Progress Signal
- Predefined Set of Standard PBX Call Progress Signals
- Important Considerations for Using Predefined Call Progress Signals

10.3.1 Using Cadenced Tone Generation

Cadenced tone generation is an enhancement to global tone generation that enables you to generate a signal with up to four single- or dual-tone elements, each with its own on/off duration creating the signal pattern or cadence.

Cadenced tone generation is accomplished with the **dx_playtoneEx( )** function and the TN_GENCAD data structure.

You can define your own custom cadenced tone or take advantage of the built-in set of standard PBX call progress signals.

The functions and data structures associated with cadenced tone generation are described in the *Voice API Library Reference*.

10.3.2 How To Generate a Custom Cadenced Tone

A custom cadenced tone is defined by specifying in a TN_GENCAD data structure the repeating elements of the signal (the cycle) and the number of desired repetitions.

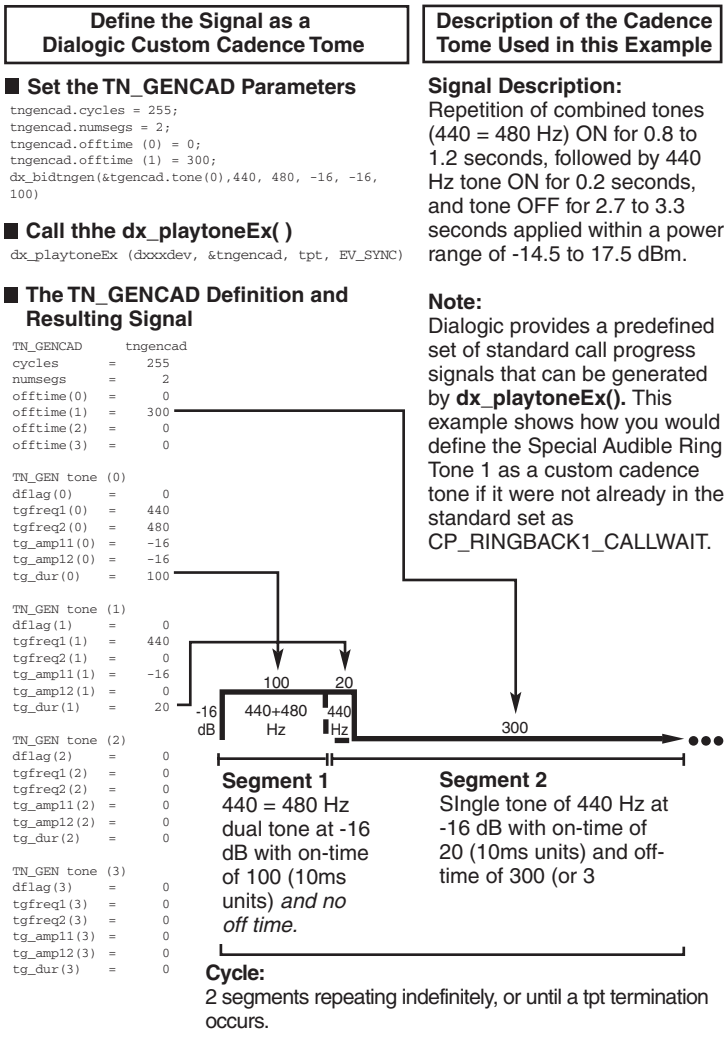
The cycle can consist of up to 4 segments, each with its own tone definition and cadence. A segment consists of a TN_GEN single- or dual-tone definition (frequency, amplitude, & duration) followed by a corresponding off-time (silence duration) that is optional. The **dx_bldtngen( )** function can be used to set up the TN_GEN components of the TN_GENCAD structure. The tone duration, or on-time, from TN_GEN (tg_dur) and the offtime from TN_GENCAD are combined to produce the cadence for the segment. The segments are seamlessly concatenated in ascending order to generate the signal cycle.

Use the following procedure to generate a custom cadenced tone:

1. Identify the repeating elements of the signal (the cycle).
2. Use a TN_GENCAD structure to define the segments in the cycle:
   a. Start with the first tone element in the cycle and identify the single- or dual-tone frequencies, amplitudes, and duration (on-time).
   b. Use the **dx_bldtngen( )** function to specify this tone definition in tone[0] (the first TN_GEN tone array element) of the TN_GENCAD structure.
   c. Identify the off-time for the first tone element and specify it in offtime[0]. If the first tone element is followed immediately by a second tone element, set offtime[0] = 0.
   d. Define the next segment of the cycle in tone[1] and offtime[1] the same way as above, and so on, up to the maximum of 4 segments or until you reach the end of the cycle.
3. Use the TN_GENCAD to define the parameters of the cycle:
   a. Specify the number of segments in the cycle (numsegs).
   b. Specify the number of cycle repetitions (cycles).
4. Set the termination conditions in a DV_TPT structure.
5. Call the **dx_playtoneEx( )** function and use the **tngencadp** parameter to specify the custom cadenced tone that you defined in the TN_GENCAD.

For an illustration of this procedure, see Figure 3.

Figure 3. Example of Custom Cadenced Tone Generation



10.3.3 How To Generate a Non-Cadenced Tone

Both **dx_playtoneEx( )** and **dx_playtone( )** can generate a non-cadenced tone.

Non-cadenced call progress signals can be generated by the **dx_playtone( )** function if you define them in a TN_GEN: Dial Tone, Executive Override Tone, and Busy Verification Tone Part A.

The **dx_playtoneEx( )** function can also generate a non-cadenced tone by using a TN_GENCAD data structure that defines a single segment.

If you want to generate a continuous, non-cadenced signal, use a single segment and zero off-time, and specify 1) an infinite number of cycles, 2) an infinite on-time, or 3) both. (You must also specify the appropriate termination conditions in a DV_TPT structure or else the tone will never end). For example:

```
cycles = 255;
numsegs = 1;
offtime[0] = 0;
tone[0].tg_dur = -1
```

10.3.4 TN_GENCAD Data Structure - Cadenced Tone Generation

TN_GENCAD is a voice library data structure (*dxxxlib.h*) that defines a cadenced tone that can be generated by using the **dx_playtoneEx( )** function.

The TN_GENCAD data structure contains a tone array with four elements that are TN_GEN data structures (Tone Generation Templates). For details on TN_GEN and TN_GENCAD, see the *Voice API Library Reference*.

For examples of TN_GENCAD, see the definitions of standard call progress signals in Table 11, “TN_GENCAD Definitions for Standard PBX Call Progress Signals”, on page 75.

10.3.5 How To Generate a Standard PBX Call Progress Signal

Use the following procedure to generate a standard PBX call progress signal from the predefined set of standard PBX call progress signals:

1. Select a call progress signal from Table 10, “Standard PBX Call Progress Signals”, on page 72 and note the signal ID (see also Figure 4, “Standard PBX Call Progress Signals (Part 1)”, on page 73).
2. Set the termination conditions in a DV_TPT structure.
3. Call the **dx_playtoneEx( )** function and specify the signal ID for the **tngencadp** parameter. For example:
   ```
   dx_playtoneEx(dxxxdev, CP_BUSY, tpt, EV_SYNC)
   ```

10.3.6 Predefined Set of Standard PBX Call Progress Signals

The following information describes the predefined set of standard PBX call progress signals that are provided by Intel:

- Table 10, "Standard PBX Call Progress Signals", on page 72 lists the predefined, standard, call progress signals and their signal IDs. The signal IDs can be used with the **dx_playtoneEx( )** function to generate the signal. (The #defines for the signal IDs are located in the *dxxxlib.h* file.)
- Figure 4, "Standard PBX Call Progress Signals (Part 1)", on page 73 illustrates the signals along with their tone specifications and cadences. The signals were divided into two parts so they could be illustrated to scale while providing sufficient detail. Part 1 uses a smaller scale than Part 2. (For this reason, the order of the signals is different than in the tables.)
- Table 11, "TN_GENCAD Definitions for Standard PBX Call Progress Signals", on page 75 lists the TN_GENCAD definitions of the signal cycle and segment definitions for each predefined call progress signal. These definitions are located in the *dxgtd.c* file.
- Section 10.3.7, "Important Considerations for Using Predefined Call Progress Signals", on page 76 describes what standard was used, how the standard was implemented, information regarding the signal power levels, usage and other considerations.

Table 10. Standard PBX Call Progress Signals

Name	Meaning	Signal ID (tngencadp)
Dial Tone	Ready for dialing	CP_DIAL
Reorder Tone (Paths-Busy, All-Trunks-Busy, Fast Busy)	Call blocked: resources unavailable	CP_REORDER
Busy Tone (Slow Busy)	Called line is busy	CP_BUSY
Audible Ring Tone 1 (Ringback Tone)	Called party is being alerted	CP_RINGBACK1
Audible Ring Tone 2 [1] (Slow Ringback Tone)	Called party is being alerted	CP_RINGBACK2
Special Audible Ring Tone 1 [1]	Called party has Call Waiting feature and is being alerted	CP_RINGBACK1_CALLWAIT
Special Audible Ring Tone 2 [1]	Called party has Call Waiting feature and is being alerted	CP_RINGBACK2_CALLWAIT
Recall Dial Tone	Ready for additional dialing on established connection	CP_RECALL_DIAL
Intercept Tone	Call blocked: invalid	CP_INTERCEPT
Call Waiting Tone 1 [2]	Call is waiting: single burst	C_CALLWAIT1
Call Waiting Tone 2 [2]	Call is waiting: double burst	CP_CALLWAIT2
Busy Verification Tone (Part A)	Alerts parties that attendant is about to enter connection	CP_BUSY_VERIFY_A
Busy Verification Tone (Part B)	Attendant remains connected	CP_BUSY_VERIFY_B
Executive Override Tone	Overriding party about to be bridged onto connection	CP_EXEC_OVERRIDE
Confirmation Tone	Feature has been activated or deactivated	CP_FEATURE_CONFIRM
Stutter Dial Tone (Message Waiting Dial Tone)	Message waiting; ready for dialing	CP_STUTTER_DIAL or CP_MSG_WAIT_DIAL

[1] Either of the two Audible Ring Tones can be used but are not intended to be intermixed in a system. When using the *Special* Audible Ring Tone (1 or 2), it should correspond to the Audible Ring Tone (1 or 2) that is used.
[2] The two Call Waiting Tones (1 & 2) can be used to differentiate between internally and externally originated calls. Call Waiting Tone 2 is defined as a double burst in this implementation, although “multiple” bursts are permissible.

Figure 4. Standard PBX Call Progress Signals (Part 1)

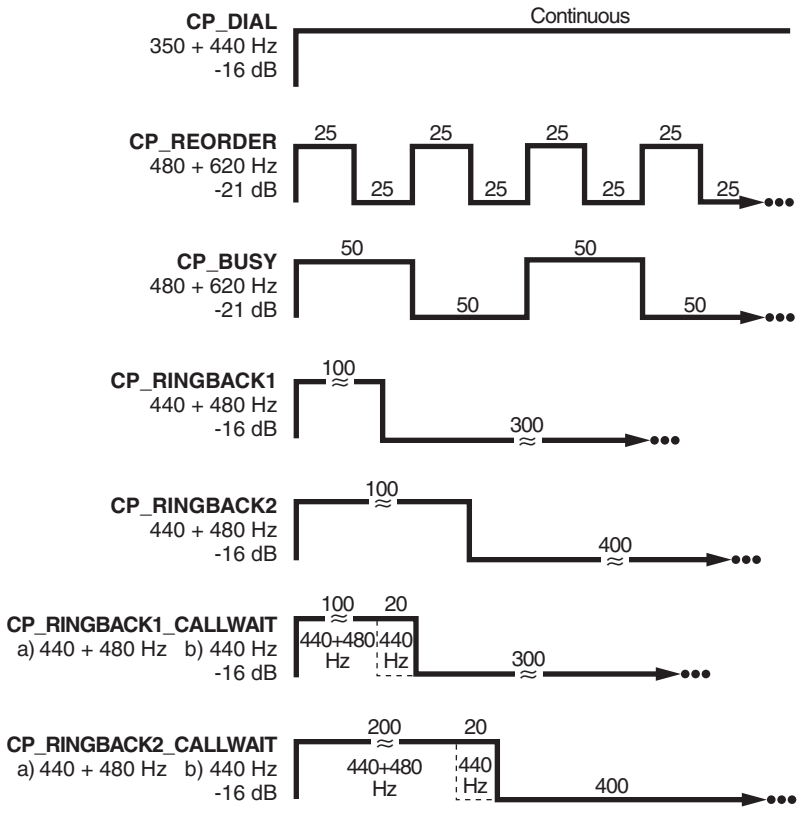


Figure 5. Standard PBX Call Progress Signals (Part 2)

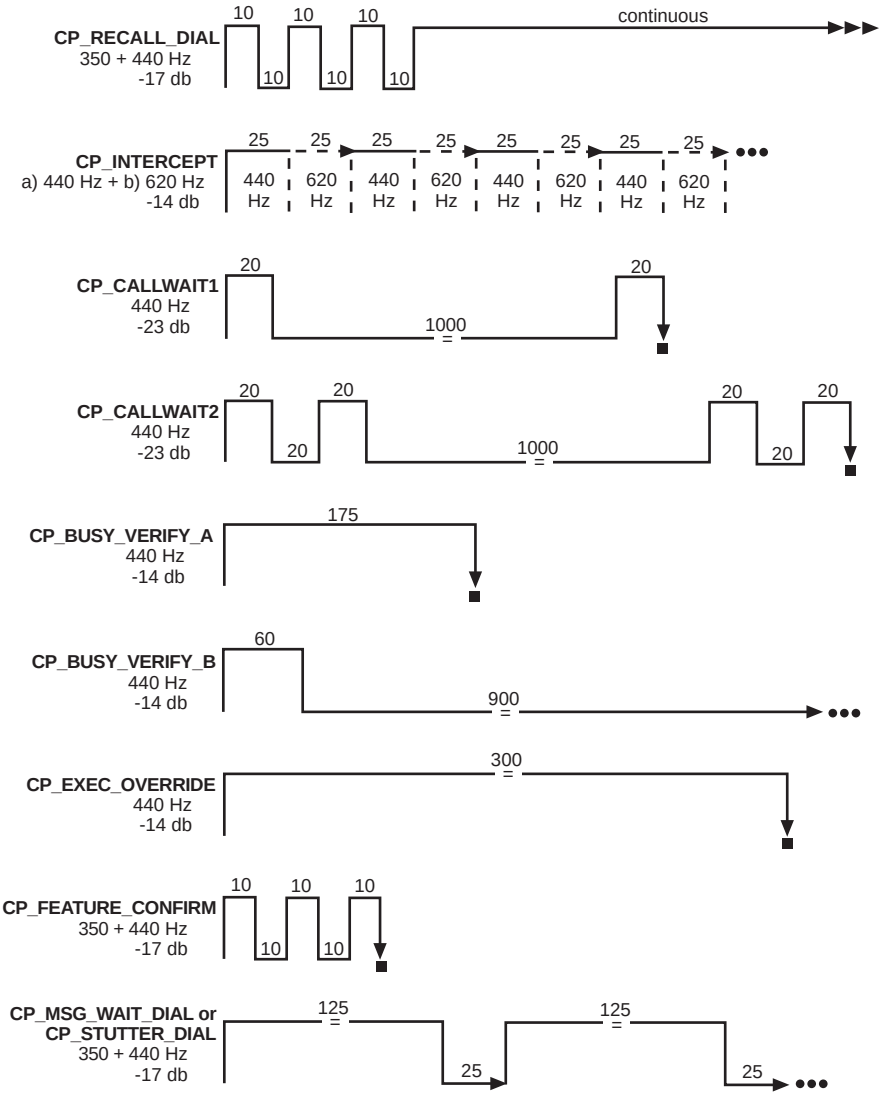


Table 11. TN_GENCAD Definitions for Standard PBX Call Progress Signals

SIGNAL_ID							
Cycle Definition		Segment Definitions					
Number of Cycles[1]	Number of Segments in Cycle	Frequency #1 (Hz)	Frequency #2 (Hz)	Amplitude #1 (dB)	Amplitude #2 (dB)	On-Time[2] (10 msec)	Off-Time (10 msec)
cycles	numsegs	tg_freq1	tg_freq2	tg_ampl1	tg_ampl2	tg_dur	offtime
CP_DIAL							
1	1	350	440	-17	-17	-1	0
CP_REORDER							
255	1	480	620	-21	-21	25	25
CP_BUSY							
255	1	480	620	-21	-21	50	50
CP_RINGBACK1							
255	1	440	480	-16	-16	100	300
CP_RINGBACK2							
255	1	440	480	-16	-16	200	400
CP_RINGBACK1_CALLWAIT							
255	2	440 440	480 0	-16 -16	-16 0	100 20	0 300
CP_RINGBACK2_CALLWAIT							
255	2	440 440	480 0	-16 -16	-16 0	200 20	0 400
CP_RECALL_DIAL							
1	4	350 350 350 350	440 440 440 440	-17 -17 -17 -17	-17 -17 -17 -17	10 10 10 -1	10 10 10 0
CP_INTERCEPT							
255	2	440 620	0 0	-14 -14	0 0	25 25	0 0
CP_CALLWAIT1							
1	2	440 440	0 0	-23 -23	0 0	20 20	1000 0
CP_CALLWAIT2							
1	4	440 440 440 440	0 0 0 0	-23 -23 -23 -23	0 0 0 0	20 20 20 20	20 1000 20 0

[1] 255 specifies an infinite number of cycles (**cycles**)
[2] -1 specifies an infinite tone duration (**tg_dur**)

Table 11. TN_GENCAD Definitions for Standard PBX Call Progress Signals (Continued)

SIGNAL_ID							
Cycle Definition		Segment Definitions					
Number of Cycles[1]	Number of Segments in Cycle	Frequency #1 (Hz)	Frequency #2 (Hz)	Amplitude #1 (dB)	Amplitude #2 (dB)	On-Time[2] (10 msec)	Off-Time (10 msec)
cycles	numsegs	tg_freq1	tg_freq2	tg_ampl1	tg_ampl2	tg_dur	offtime
CP_BUSY_VERIFY_A							
1	1	440	0	-14	0	175	0
CP_BUSY_VERIFY_B							
255	1	440	0	-14	0	60	900
CP_EXEC_OVERRIDE							
1	1	440	0	-14	0	300	0
CP_FEATURE_CONFIRM							
1	3	350 350 350	440 440 440	-17 -17 -17	-17 -17 -17	10 10 10	10 10 0
CP_STUTTER_DIAL or CP_MSG_WAIT_DIAL							
255	1	350	440	-17	-17	125	25

[1] 255 specifies an infinite number of cycles (**cycles**)
[2] -1 specifies an infinite tone duration (**tg_dur**)

10.3.7 Important Considerations for Using Predefined Call Progress Signals

Take into account the following considerations when using the predefined call progress signals:

- Signal definitions are based on the TIA/EIA Standard: Requirements for Private Branch Exchange (PBX) Switching Equipment, TIA/EIA-464-B, April 1996 (Telecommunications Industry Association in association with the Electronic Industries Association, Standards and Technology Department, 2500 Wilson Blvd., Arlington, VA 22201). To order copies, contact Global Engineering Documents in the USA at 1-800-854-7179 or 1-303-397-7956.
- A separate Line Lockout Warning Tone, which indicates that the station line has been locked out because dialing took too long or the station failed to disconnect at the end of a call, is not necessary and is not recommended. You can use the Reorder tone over trunks; or the Intercept, Reorder, or Busy tone over stations.
- For signals that specify an infinite repetition of the signal cycle (cycles = 255 on Springware or 40 on DM3) or an infinite duration of a tone (tg_dur = -1), you must specify the appropriate termination conditions in the DV_TPT structure used by **dx_playtoneEx( )**.
- There may be more than one way to use TN_GENCAD to generate a given signal. For example, the three bursts of the Confirmation Tone can be created through one cycle containing three segments (as in the Intel implementation) or through a single segment that is repeated in three cycles.

- To generate a continuous, non-cadenced signal, you can use **dx_playtoneEx( )** and TN_GENCAD to specify a single segment with zero off-time and with an infinite number of cycles and/or an infinite on-time.

  Alternatively, you could use **dx_playtone( )** and TN_GEN to generate a non-cadenced signal. The following non-cadenced call progress signals could be generated by the **dx_playtone( )** function if you defined them in a TN_GEN: 1) Dial Tone, 2) Executive Override Tone, and 3) Busy Verification Tone Part A.
- Note that the Intercept Tone consists of alternating single tones.
- Although the TIA/EIA Standard describes the Busy Verification Tone as one signal, the two segments are separate tones/events: Part A is a single burst almost three times longer than Part B and it alerts the parties before the attendant intrudes; Part B is a short burst every 9 seconds continuing as long as the interruption lasts. The TIA/EIA Standard does not define an off-time between Part A and B. Therefore, the application developer is responsible for implementing the timing between the two parts of this signal.
- The TIA/EIA Standard specifies the range of permissible power levels per frequency for 1) the Central Office trunk interface and 2) all other interfaces (including off-premise stations and tie trunks). The Intel implementation adopted the approximate middle value in the acceptable range of power levels for applying the signals to the CO trunk interface. These power levels were more restrictive than those for the other interfaces. According to the following statement in the TIA/EIA Standard, additional requirements and considerations may apply:

  "Studies have shown that the lower level tones that are transmitted over trunks should be 6 dB hotter at the trunk interface (than at the line interface) to compensate for increased loss on the end-to-end connection. In the case of tones used at higher levels, the 6 dB difference is not used since power at trunk interfaces must be limited to -13 dBm0 total when averaged over any 3-second interval to prevent carrier overload. Maximum permissible powers listed are consistent with this requirement taking into account the allowable interruption rates for the various tones. Uninterrupted tones, such as Dial Tone and Intercept Tone, shall be continuously limited to -13 dBm."

  For related power level information, see also Note 1 for Tables 29 and 30, Section 5.9, and Section 6.3.5.

Building Applications 11

This chapter provides information on building applications using the voice library. The following topics are discussed:

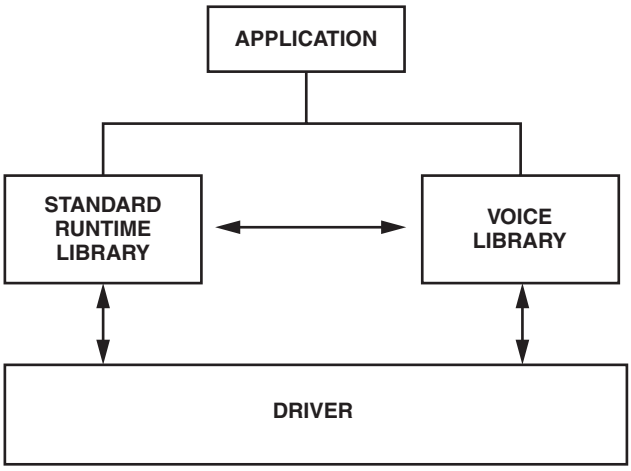
11.1 Voice and SRL Libraries

The C-language application programming interface (API) included with the voice software provides a library of functions used to create voice processing applications.

The voice library and Standard Runtime Library (SRL) files are part of the voice software. These libraries provide the interface to the voice driver. For detailed information on the SRL, see the *Standard Runtime Library API Programming Guide* and *Standard Runtime Library API Programming Guide*.

Figure 6 illustrates how the voice and SRL libraries interface with the driver.

Figure 6. Voice and SRL Libraries



11.2 Compiling and Linking

The following topics discuss compiling and linking requirements:

- Include Files
- Required Libraries
- Variables for Compiling and Linking

11.2.1 Include Files

Function prototypes and equates are defined in include files, also known as header files. Applications that use voice library functions must contain statements for include files in this form, where filename represents the include file name:

```
#include <filename.h>
```

The following header files must be included in application code **in the order shown** prior to calling voice library functions:

srllib.h
Contains function prototypes and equates for the Standard Runtime Library (SRL). Used for all application development.

dxxxlib.h
Contains function prototypes and equates for the voice library. Used for voice processing applications.

Note: *srllib.h* must be included in code before all other Intel header files.

11.2.2 Required Libraries

Simple C language interfaces in source-code format are provided to each individual technology DLL (such as standard runtime, voice, fax, and network interfaces). These C language interfaces allow an application to perform run-time linking instead of compile-time linking.

Note: Compile-time linking requires that all functions called in an application be contained in the DLL that resides on the system.

By default, the library files are located in the directory given by the INTEL_DIALOGIC_LIB environment variable.

Linux

You must link the following shared object library files **in the order shown** when compiling your voice processing application:

libdxxx.so
Main voice library file. Specify **-ldxxx** in makefile.

libsrl.so
Standard Runtime Library file. Specify **-lsrl** in makefile.

If you use **curses**, you must ensure that it is the last library to be linked.

Windows

You must link the following library files **in the order shown** when compiling your voice processing application:

libdxxmt.lib
Main voice library file.

libsrlmt.lib
Standard Runtime Library file.

11.2.3 Variables for Compiling and Linking

The following variables provide a standardized way of referencing the directories that contain header files and shared objects:

INTEL_DIALOGIC_INC
Variable that points to the directory where header files are stored.

INTEL_DIALOGIC_LIB
Variable that points to the directory where shared library files are stored.

These variables are automatically set at login and should be used in compiling and linking commands. The following is an example of a compiling and linking command that uses these variables:

```
cc -I${INTEL_DIALOGIC_INC} -o myapp myapp.c -L${INTEL_DIALOGIC_LIB} -lsrl
```

Note: It is strongly recommended that you use these variables when compiling and linking applications. The name of the variables will remain constant, but the values may change in future releases.

Glossary

A-law: Pulse Code Modulation (PCM) algorithm used in digitizing telephone audio signals in E1 areas. Contrast with mu-law.

ADPCM (Adaptive Differential Pulse Code Modulation): A sophisticated compression algorithm for digitizing audio that stores the differences between successive samples rather than the absolute value of each sample. This method of digitization reduces storage requirements from 64 kilobits/second to as low as 24 kilobits/second.

AGC (Automatic Gain Control): An electronic circuit used to maintain the audio signal volume at a constant level. AGC maintains nearly constant gain during voice signals, thereby avoiding distortion, and optimizes the perceptual quality of voice signals by using a new method to process silence intervals (background noise).

analog: 1. A method of telephony transmission in which the signals from the source (for example, speech in a human conversation) are converted into an electrical signal that varies continuously over a range of amplitude values analogous to the original signals. 2. Not digital signaling. 3. Used to refer to applications that use loop start signaling.

ANI (Automatic Number Identification): Identifies the phone number that is calling. Digits may arrive in analog or digital form.

API (Application Programming Interface): A set of standard software interrupts, calls, and data formats that application programs use to initiate contact with network services, mainframe communications programs, or other program-to-program communications.

ASCIIZ string: A null-terminated string of ASCII characters.

asynchronous function: A function that allows program execution to continue without waiting for a task to complete. To implement an asynchronous function, an application-defined event handler must be enabled to trap and process the completed event. Contrast with synchronous function.

bit mask: A pattern which selects or ignores specific bits in a bit-mapped control or status field.

bitmap: An entity of data (byte or word) in which individual bits contain independent control or status information.

board device: On Host Media Processing (HMP) software, a board-level object that can be manipulated by a physical library. HMP performs like a virtual DM3 board.

buffer: A block of memory or temporary storage device that holds data until it can be processed. It is used to compensate for the difference in the rate of the flow of information (or time occurrence of events) when transmitting data from one device to another.

bus: An electronic path that allows communication between multiple points or devices in a system.

busy device: A device that has one of the following characteristics: is stopped, being configured, has a multitasking or non-multitasking function active on it, or I/O function active on it.

cadence: A pattern of tones and silence intervals generated by a given audio signal. The pattern can be classified as a single ring, a double ring, or a busy signal.

cadence detection: A voice driver feature that analyzes the audio signal on the line to detect a repeating pattern of sound and silence.

call progress analysis: A process used to automatically determine what happens after an outgoing call is dialed. A further distinction is made. Call progress refers to activity that occurs before a call is connected (pre-connect), such as busy or ringback. Call analysis refers to activity that occurs after a call is connected (post-connect), such as voice detection and answering machine detection. The term call progress analysis is used to encompass both call progress and call analysis.

call status transition event functions: A class of functions that set and monitor events on devices.

caller ID: calling party identification information.

CCITT (Comite Consultatif Internationale de Telegraphique et Telephonique): One of the four permanent parts of the International Telecommunications Union, a United Nations agency based in Geneva. The CCITT is divided into three sections: 1. Study Groups set up standards for telecommunications equipment, systems, networks, and services. 2. Plan Committees develop general plans for the evolution of networks and services. 3. Specialized Autonomous Groups produce handbooks, strategies, and case studies to support developing countries.

channel: 1. When used in reference to an Intel analog expansion board, an audio path, or the activity happening on that audio path (for example, when you say the channel goes off-hook). 2. When used in reference to an Intel® digital expansion board, a data path, or the activity happening on that data path. 3. When used in reference to a bus, an electrical circuit carrying control information and data.

channel device: A channel-level object that can be manipulated by a physical library, such as an individual telephone line connection. A channel is also a subdevice of a board. See also subdevice.

CO (Central Office): A local phone network exchange, the telephone company fadcility where subscriber lines are linked, through switches, to other subscriber lines (including local and long distance lines). The term “Central Office” is used in North America. The rest of the world calls it “PTT”, for Post, Telephone, and Telegraph.

computer telephony (CT): The extension of computer-based intelligence and processing over the telephone network to a telephone. Sometimes called computer-telephony integration (CTI), it lets you interact with computer databases or applications from a telephone, and enables computer-based applications to access the telephone network. Computer telephony technology supports applications such as: automatic call processing; automatic speech recognition; text-to-speech conversion for information-on-demand; call switching and conferencing; unified messaging, which lets you access or transmit voice, fax, and e-mail messages from a single point; voice mail and voice messaging; fax systems, including fax broadcasting, fax mailboxes, fax-on-demand, and fax gateways; transaction processing, such as Audiotex and Pay-Per-Call information systems; and call centers handling a large number of agents or telephone operators for processing requests for products, services, or information.

configuration file: An unformatted ASCII file that stores device initialization information for an application.

convenience function: A class of functions that simplify application writing, sometimes by calling other, lower-level API functions.

CPE: customer premise equipment.

CT Bus: Computer Telephony bus. A time division multiplexing communications bus that provides 4096 time slots for transmission of digital information between CT Bus products. See TDM bus.

data structure: Programming term for a data element consisting of fields, where each field may have a different type definition and length. A group of data structure elements usually share a common purpose or functionality.

DCM: configuration manager. On Windows only, a utility with a graphical user interface (GUI) that enables you to add new boards to your system, start and stop system service, and work with board configuration data.

debouncing: Eliminating false signal detection by filtering out rapid signal changes. Any detected signal change must last for the minimum duration as specified by the debounce parameters before the signal is considered valid. Also known as deglitching.

deglitching: See debouncing.

device: A computer peripheral or component controlled through a software device driver. An Intel voice and/or network interface expansion board is considered a physical board containing one or more logical board devices, and each channel or time slot on the board is a device.

device channel: An Intel voice data path that processes one incoming or outgoing call at a time (equivalent to the terminal equipment terminating a phone line).

device driver: Software that acts as an interface between an application and hardware devices.

device handle: Numerical reference to a device, obtained when a device is opened using **xx_open( )**, where *xx* is the prefix defining the device to be opened. The device handle is used for all operations on that device.

device name: Literal reference to a device, used to gain access to the device via an **xx_open( )** function, where *xx* is the prefix defining the device to be opened.

digitize: The process of converting an analog waveform into a digital data set.

DM3: Refers to Intel mediastream processing architecture, which is open, layered, and flexible, encompassing hardware as well as software components. A whole set of products from Intel are built on the Intel® DM3™ architecture. Contrast with Springware, which is earlier-generation architecture.

download: The process where board level program instructions and routines are loaded during board initialization to a reserved section of shared RAM.

downloadable Springware firmware: Software features loaded to Intel voice hardware. Features include voice recording and playback, enhanced voice coding, tone detection, tone generation, dialing, call progress analysis, voice detection, answering machine detection, speed control, volume control, ADSI support, automatic gain control, and silence detection.

driver: A software module which provides a defined interface between an application program and the firmware interface.

DSP (Digital Signal Processor): A specialized microprocessor designed to perform speedy and complex operations on digital signals.

DTMF (Dual-Tone Multi-Frequency): Push-button or touch-tone dialing based on transmitting a high- and a low-frequency tone to identify each digit on a telephone keypad.

echo: The component of an analog device's receive signal reflected into the analog device's transmit signal.

echo cancellation: Removal of echo from an echo-carrying signal.

event: An unsolicited or asynchronous message from a hardware device to an operating system, application, or driver. Events are generally attention-getting messages, allowing a process to know when a task is complete or when an external event occurs.

event handler: A portion of an application program designed to trap and control processing of device-specific events.

extended attribute functions: A class of functions that take one input parameter (a valid Intel device handle) and return device-specific information. For instance, a voice device's extended attribute function returns information specific to the voice devices. Extended attribute function names are case-sensitive and must be in capital letters. See also standard runtime library (SRL).

firmware: A set of program instructions that reside on an expansion board.

firmware load file: The firmware file that is downloaded to a voice board.

flash: A signal generated by a momentary on-hook condition. This signal is used by the voice hardware to alert a telephone switch that special instructions will follow. It usually initiates a call transfer. See also I/O.

G.726: An international standard for encoding 8 kHz sampled audio signals for transmission over 16, 24, 32 and 40 kbps channels. The G.726 standard specifies an adaptive differential pulse code modulation (ADPCM) system for coding and decoding samples.

GSM: A speech compression algorithm developed for the Global System for Mobile telecommunication (GSM), Europe's popular protocol suite for digital cellular communication.

I/O: Input-Output

idle device: A device that has no functions active on it.

in-band: The use of robbed-bit signaling (T1 systems only) on the network. The signaling for a particular channel or time slot is carried within the voice samples for that time slot, thus within the 64 kbps (kilobits per second) voice bandwidth.

kernel: A set of programs in an operating system that implement the system's functions.

mu-law: (1) Pulse Code Modulation (PCM) algorithm used in digitizing telephone audio signals in T1 areas. (2) The PCM coding and companding standard used in Japan and North America. See also A-law.

PBX: Private Branch Exchange. A small version of the phone company's larger central switching office. A local premises or campus switch.

PCM (Pulse Code Modulation): A technique used in DSP voice boards for reducing voice data storage requirements. Intel supports either mu-law PCM, which is used in North America and Japan, or A-law PCM, which is used in the rest of the world.

polling: The process of repeatedly checking the status of a resource to determine when state changes occur.

PSTN (or STN): Public (or Private) Switched Telephony Network

resource: Functionality (for example, voice-store-and-forward) that can be assigned to a call. Resources are *shared* when functionality is selectively assigned to a call and may be shared among multiple calls. Resources are *dedicated* when functionality is fixed to the one call.

resource board: An Intel expansion board that needs a network or switching interface to provide a technology for processing telecommunications data in different forms, such as voice store-and-forward, speech recognition, fax, and text-to-speech.

RFU: reserved for future use

ring detect: The act of sensing that an incoming call is present by determining that the telephone switch is providing a ringing signal to the voice board.

route: Assign a resource to a time slot.

sampling rate: Frequency at which a digitizer quantizes the analog voice signal.

SCbus (Signal Computing Bus): A hardwired connection between Switch Handlers on SCbus-based products. SCbus is a third generation TDM (Time Division Multiplexed) resource sharing bus that allows information to be transmitted and received among resources over 1024 time slots.

signaling insertion: The signaling information (on hook/off hook) associated with each channel is digitized, inserted into the bit stream of each time slot by the device driver, and transmitted across the bus to another resource device. The network interface device generates the outgoing signaling information.

silence threshold: The level that sets whether incoming data to the voice board is recognized as silence or non-silence.

SIT: (1) Standard Information Tones: tones sent out by a central office to indicate that the dialed call has been answered by the distant phone. (2) Special Information Tones: detection of a SIT sequence indicates an operator intercept or other problem in completing the call.

solicited event: An expected event. It is specified using one of the device library's asynchronous functions.

Springware: Software algorithms built into the downloadable firmware that provide the voice processing features available on older-generation Intel® Dialogic® voice boards. The term Springware is also used to refer to a whole set of boards from Intel built using this architecture. Contrast with DM3, which is a newer-generation architecture.

SRL: See **Standard Runtime Library**.

standard attribute functions: Class of functions that take one input parameter (a valid device handle) and return generic information about the device. For instance, standard attribute functions return IRQ and error information for all device types. Standard attribute function names are case-sensitive and must be in capital letters. Standard attribute functions for Intel telecom devices are contained in the SRL. See standard runtime library (SRL).

standard runtime library (SRL): An Intel software resource containing event management and standard attribute functions and data structures used by Intel telecom devices.

station device: Any analog telephone or telephony device (such as a telephone or headset) that uses a loop-start interface and connects to a station interface board.

string: An array of ASCII characters.

subdevice: Any device that is a direct child of another device. Since "subdevice" describes a relationship between devices, a subdevice can be a device that is a direct child of another subdevice, as a channel is a child of a board.

synchronous function: Blocks program execution until a value is returned by the device. Also called a blocking function. Contrast with asynchronous function.

system release: The software and user documentation provided by Intel that is required to develop applications.

TDM (Time Division Multiplexing): A technique for transmitting multiple voice, data, or video signals simultaneously over the same transmission medium. TDM is a digital technique that interleaves groups of bits from each signal, one after another. Each group is assigned its own "time slot" and can be identified and extracted at the receiving end. See also time slot.

TDM bus: Time division multiplexing bus. A resource sharing bus such as the SCbus or CT Bus that allows information to be transmitted and received among resources over multiple data lines.

termination condition: An event or condition which, when present, causes a process to stop.

termination event: An event that is generated when an asynchronous function terminates. See also **asynchronous function**.

time division multiplexing (TDM): See TDM (Time Division Multiplexing).

time slot: The smallest, switchable data unit on a TDM bus. A time slot consists of 8 consecutive bits of data. One time slot is equivalent to a data path with a bandwidth of 64 kbps. In a digital telephony environment, a normally continuous and individual communication (for example, someone speaking on a telephone) is (1) digitized, (2) broken up into pieces consisting of a fixed number of bits, (3) combined with pieces of other individual communications in a regularly repeating, timed sequence (multiplexed), and (4) transmitted serially over a single telephone line. The process happens at such a fast rate that, once the pieces are sorted out and put back

together again at the receiving end, the speech is normal and continuous. Each individual, pieced-together communication is called a time slot.

time slot assignment: The ability to route the digital information contained in a time slot to a specific analog or digital channel on an expansion board. See also device channel.

virtual board: In the traditional voice processing board environment, the device driver views a single physical voice board with more than four channels as multiple emulated D/4x boards. These emulated boards are called virtual boards. This concept extends to the Host Media Processing (HMP) software environment. A system with 44 channels consists of 11 virtual boards.

voice processing: The science of converting human voice into data that can be reconstructed and played back at a later time.

voice system: A combination of expansion boards and software that lets you develop and run voice processing applications.

Index

E

F

G

U

V