Dialogic® Voice API

Programming Guide

October 2010

05-2332-007

Publication Date: October 2010

Document Number: 05-2332-007

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
05-2332-007	October 2010	Made global changes to add Dialogic® Springware Architecture board support for Dialogic® HMP Software. Product Description chapter: Added two forms of Call Progress Analysis, Dial Pulse Detection, Send and Receive FSK Data, Caller ID. Event Handling chapter: Added dx_sethook( ) to Table 1 and Table 2. Application Development Guidelines chapter: In Setting Termination Conditions for I/O Functions, added loop current drop, maximum length of non-silence, pattern of silence and non-silence, maximum FSK data received. In Additional Considerations, added Device Discovery. Call Progress Analysis chapter: Added support for Springware boards. Recording and Playback chapter: Added coder support for Springware boards section in Voice Encoding Methods. In Voice Encoding Methods (HMP Software) table, (1) added G.729A; (2) added 24 and 40 Kbps for G.726; (3) added linear PCM 64 Kbps; (4) removed VOX from the GSM 6.10 full rate (Microsoft format) row; (5) removed GSM 6.10 full rate (TIPHON format) row. Updated and added Springware board support in Transaction Record, Silence Compressed Record, and Recording with the Voice Activity Detector. Added a new first step (calling dx_open()) in Implementing Streaming to Board. Speed and Volume Control chapter: Added support for Springware boards. Send and Receive FSK Data chapter: New. Applies to Springware boards only. Caller ID chapter: New. Applies to Springware boards only. Global Tone Detection and Generation, and Cadenced Tone Generation chapter: Added support for Springware boards. Global Dial Pulse Detection chapter: New. Applies to Springware boards only. Building Applications chapter: Removed information about run-time linking in Required Libraries for Linux and Required Libraries for Windows® sections, as this is no longer supported.
05-2332-006	April 2009	Recording and Playback chapter: Added GSM 6.10 full-rate coder (Microsoft format and TIPHON format) to Voice Encoding Methods table. Added Recording with the Voice Activity Detector. Updated Streaming to Board Guidelines to indicate that on Windows® the bulk queue buffer size can be modified; dx_setchxfercnt( ) is supported on Windows®.
05-2332-005	January 2008	Made global changes to reflect Dialogic brand. Product Description chapter: Added information about silence compressed record in Play and Record Features section. Recording and Playback chapter: Added support for silence compressed record in Silence Compressed Record section.
05-2332-004	August 2006	Product Description chapter: Added support for speed control in Speed and Volume Control section. Speed and Volume Control chapter: Added support for speed control.

Document No.	Publication Date	Description of Revisions
05-2332-003	December 2005	Product Description chapter: Updated TDM Bus Routing section to include information about Dialogic® digital network interface boards. Application Development Guidelines chapter: Added note about continuous speech processing (CSP) multiprocess support in Multithreading and Multiprocessing section. Added bullet about digits not always being cleared by dx_clrdigbuf( ) in Tone Detection Considerations section. Call Progress Analysis chapter: Updated dial tone detection row to 'yes' in Call Progress Analysis Support with dx_dial( ) table. Updated to add support for ATDX_CRTNID( ) and enhanced SIT sequences. Added eight new SIT sequences that can be returned by ATDX_CRTNID( ) for DM3 boards in Types of Tones section. Revised values of TID_SIT_NC (Freq of first segment changed from 950/1001 to 950/1020) and TID_SIT_VC (Freq of first segment changed from 950/1001 to 950/1020) in table of Special Information Tone Sequences (DM3); also added four new SIT sequences to this table. Added note about SIT sequences that cannot be modified in API Functions for Manipulating Tone Definitions section. Added note about SRL device mapper functions in Steps to Modify a Tone Definition on DM3 Boards section. Recording and Playback chapter: Added 128 Kbps (8 kHz, 16-bit) linear PCM to Voice Encoding Methods table. Added cross-reference to a related technical note in Transaction Record section. Global Tone Detection and Generation, and Cadenced Tone Generation chapter: Added note about the effect of exhausting the number of tone templates in Guidelines for Creating User-Defined Tones section. Added bullet about the effect of adding a tone with a frequency of zero in Guidelines for Creating User-Defined Tones section. [PTR 34546]
05-2332-002	April 2005	Product Description chapter: Added section on Transaction Record. Application Development Guidelines chapter: Added caveat about dx_clrdigbuf( ) in Tone Detection Considerations section. Recording and Playback chapter: Added section on Transaction Record. Global Tone Detection and Generation, and Cadenced Tone Generation chapter: Updated the following sections: Overview of Global Tone Detection, Building Tone Templates, Working with Tone Templates, and Guidelines for Creating User-Defined Tones. Added new section on Global Tone Detection on HMP versus Springware Boards.
05-2332-001	September 2004	Initial version of document.

About This Publication

The following topics provide information about this publication:

- Purpose
- Applicability
- Intended Audience
- How to Use This Publication
- Related Information

Purpose

This guide provides instructions for developing applications on Linux and Windows® operating systems using the Dialogic® Voice API that is supplied with the Dialogic® Host Media Processing (HMP) Software product.

This document is a companion guide to the *Dialogic® Voice API Library Reference*, which describes the voice functions, data structures, events, and error codes.

Applicability

This document version is published for Dialogic® Host Media Processing Software Release 3.0WIN Service Update and Dialogic® Host Media Processing Software Release 4.1LIN Service Update.

This document also applies to Dialogic® Springware Architecture PCIe boards that are supported by Dialogic® HMP Software; for example, the D/80PCIE-LS board.

This document may also be applicable to other software releases (including service updates) on Linux or Windows® operating systems. 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 choose to access the voice software. They may include distributors, system integrators, toolkit developers, independent software vendors (ISVs), value added resellers (VARs), and 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 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 one can 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, "Speed and Volume Control" explains how to control the speed and volume of playback recordings through API functions and data structures.
- Chapter 10, "Send and Receive FSK Data" describes the two-way frequency shift keying (FSK) feature and the Analog Display Services Interface (ADSI).
- Chapter 11, "Caller ID" describes the caller ID feature.
- Chapter 12, "Global Tone Detection and Generation, and Cadenced Tone Generation" describes the tone detection and tone generation features.
- Chapter 13, "Global Dial Pulse Detection" discusses the Global DPD feature, the API functions provided for this feature, and programming guidelines.
- Chapter 14, "Building Applications" discusses compiling and linking requirements such as include files and library files.

Related Information

See the following for additional information:

- *http://www.dialogic.com/manuals/* (for Dialogic® product documentation)
- *http://www.dialogic.com/support/* (for Dialogic technical support)
- *http://www.dialogic.com/* (for Dialogic® product information)

Product Description 1

This chapter provides information on Dialogic® Voice API library features and capabilities. The following topics are covered:

1.1 Overview

The Dialogic® Voice API provides a comprehensive set of features for building a wide range of computer telephony applications such as voice messaging, interactive voice response, telemarketing/call center, operator services, and more. Features include DTMF detection, tone signaling, global tone detection and generation, call progress analysis, and a variety of voice encoding algorithms selectable on a channel-by-channel basis.

The Dialogic® Voice API consists of a C language library of functions, device drivers, and firmware.

The Dialogic® Voice API library is well integrated with other technology libraries provided by Dialogic such as fax, conferencing, speech, and multimedia services. This architecture enables new capabilities to be added to your application as needed.

1.2 Dialogic® 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 Dialogic® R4 API is a rich set of proprietary APIs for building computer telephony applications on Dialogic® products, including Dialogic® Host Media Processing (HMP) software. These APIs encompass technologies such as fax, conferencing, speech, and multimedia. This document describes the Dialogic® Voice API.

The R4 API supports the original Dialogic® Springware architecture products (also referred to herein as "Dialogic® Springware boards" or "Springware boards") and the later Dialogic® DM3 mediastream architecture products (also referred to herein as "Dialogic® DM3 boards" or "DM3 boards").

Feature differences between these two categories of products, as well as restrictions and limitations, are noted in this document as applicable.

1.3 Dialogic® Host Media Processing (HMP) Software

Dialogic® Host Media Processing (HMP) Software performs media processing tasks on general-purpose servers without the need for specialized hardware. When installed on a system, Dialogic® HMP Software performs like a virtual Dialogic® DM3 board to the customer application, but all media processing takes place on the host processor.

Note: In this document, the term "**DM3 board**" or " **board**" represents the virtual Dialogic® DM3 board. The terms are used interchangeably with Dialogic® HMP Software.

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).

There are two forms of call progress analysis: basic and Perfect Call. Perfect Call call progress analysis uses an improved method of signal identification and can detect fax machines and answering machines. Basic call progress analysis provides backward compatibility for older applications written before Perfect Call call progress analysis became available.

Basic call progress analysis is supported on Dialogic® Springware boards only.

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 12, “Global Tone Detection and Generation, and Cadenced Tone Generation” for more 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 12, “Global Tone Detection and Generation, and Cadenced Tone Generation” for more 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 12, “Global Tone Detection and Generation, and Cadenced Tone Generation” for more information about cadenced tone generation.

1.6 Dial Pulse Detection

Dial pulse detection (DPD) allows applications to detect dial pulses from rotary or pulse phones by detecting the audible clicks produced when a number is dialed, and to use these clicks as if they were DTMF digits. Global dial pulse detection, called global DPD, is a software-based dial pulse detection method that can use country-customized parameters for extremely accurate performance.

Dial pulse detection is supported on Dialogic® Springware boards only.

See Chapter 13, “Global Dial Pulse Detection” for more information about this feature.

1.7 Play and Record Features

The play and record features that are supported by the Dialogic® Voice API library include the following:

- Play and Record Functions
- Speed and Volume Control
- Transaction Record
- Silence Compressed Record
- Streaming to Board

1.7.1 Play and Record Functions

The Dialogic® Voice API 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". For information about voice encoding methods supported, see Section 8.5, "Voice Encoding Methods", on page 76. For information about play and record functions, see the *Dialogic® Voice API Library Reference*.

1.7.2 Speed and Volume Control

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

Se Chapter 9, "Speed and Volume Control" for more information about this feature.

1.7.3 Transaction Record

The transaction record feature allows voice activity on two channels to be summed and stored in a single file, or in a combination of files, devices, and memory. This feature is useful in call center applications where it is necessary to archive a verbal transaction or record a live conversation.

See Chapter 8, "Recording and Playback" for more information on the transaction record feature.

1.7.4 Silence Compressed Record

The silence compressed record (SCR) feature enables recording with silent pauses eliminated. This results in smaller recorded files with no loss of intelligibility.

When the audio level is at or falls below the silence threshold for a minimum duration of time, silence compressed record begins. If a short burst of noise (glitch) is detected, the compression does not end unless the glitch is longer than a specified period of time.

See Chapter 8, "Recording and Playback" for more information.

1.7.5 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.

Streaming to board is not supported on Dialogic® Springware boards.

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

1.8 Send and Receive FSK Data

The send and receive frequency shift keying (FSK) data interface is used for Analog Display Services Interface (ADSI) and fixed-line short message service, also called small message service, or SMS. Frequency shift keying is a frequency modulation technique to send digital data over voiced band telephone lines. ADSI allows information to be transmitted for display on a display-based telephone connected to an analog loop start line, and to store and forward SMS messages in the Public Switched Telephone Network (PSTN). The telephone must be a true ADSI-compliant or fixed line SMS-compliant device.

Send and receive FSK data is supported on Dialogic® Springware boards only.

See Chapter 10, "Send and Receive FSK Data" for more information on ADSI, FSK, and SMS.

1.9 Caller ID

An application can enable the caller ID feature on specific channels to process caller ID information as it is received with an incoming call. Caller ID information can include the calling party's directory number (DN), the date and time of the call, and the calling party's subscriber name.

Caller ID as described here is applicable to Dialogic® Springware boards only.

See Chapter 11, "Caller ID" for more information about this feature.

1.10 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.

The CT Bus is an implementation of the computer telephony bus standard developed by the Enterprise Computer Telephony Forum (ECTF) and accepted industry-wide. The H.100 hardware specification covers CT Bus implementation using the PCI form factor. The CT Bus has 4096 bi-directional time slots.

On Dialogic® HMP Software, no physical TDM bus exists but its functionality is implemented in the software. Starting with Dialogic® HMP Software Release 2.0 for Windows®, Dialogic® HMP Interface Boards are supported. These boards have a bridge device that is capable of streaming data between HMP Software and boards connected to the CT Bus. Additionally, the bridge device is capable of providing clocking to HMP Software. The clocking provided to HMP Software from an HMP Interface Board is derived from CT Bus clocking. For more information on clocking, see the Configuration Guide associated with your software release.

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

Programming Models 2

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

2.1 Dialogic® Standard Runtime Library

The Dialogic® Standard Runtime Library (SRL) provides a set of common system functions that are device independent and are applicable to all Dialogic® 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 *Dialogic® Standard Runtime Library API Library Reference* and *Dialogic® 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 *Dialogic® 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 *Dialogic® 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 important in 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 Dialogic® 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 Dialogic® 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 Dialogic® Standard Runtime Library device mapper functions to retrieve information on all devices in a system.

For complete information on device handling, see the *Dialogic® 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. Dialogic® Voice API 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 *Dialogic® 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 Dialogic® 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 Dialogic® devices. For more information on event management and event handling, see the *Dialogic® 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 *Dialogic® 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 *Dialogic® 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_SETHOOK	**dx_sethook( )**
		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_SETHOOK	**dx_sethook( )**
		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 Dialogic® Voice API 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 Dialogic® Voice API 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 *Dialogic® 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 Dialogic® Voice API library function, see the *Dialogic® 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 Dialogic® Voice API library. The following topics are discussed:

6.1 General Considerations

The following considerations apply to all applications written using the Dialogic® Voice API:

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

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 *Dialogic® 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 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.

The termination conditions are described in the following paragraphs.

byte transfer count

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.

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.

end of file reached

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. When this termination condition is met, a TM_EOD termination reason is returned from **ATDX_TERMMSK( )**.

loop current drop (DX_LCOFF)

Not supported on Dialogic® HMP Software using the voice library; however, support is available via call control API. For more information, see the *Dialogic® Global Call Analog Technology User’s Guide*.

In some central offices, switches, and PBXs, a drop in loop current indicates disconnect supervision. An I/O function can terminate if the loop current drops for a specified amount of time. The amount of time is specified in the tp_length field of a DV_TPT structure. The amount of time can be specified in 100 msec units (default) or 10 msec units. 10 msec can be specified in the tp_flags field of the DV_TPT. When this termination condition is met, a TM_LCOFF termination reason is returned from **ATDX_TERMMSK( )**.

maximum delay between digits (DX_IDDTIME)

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( )**.

On Dialogic® HMP Software, this termination condition is only supported by the **dx_getdig( )** function.

maximum digits received (DX_MAXDTMF)

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 non-silence ((DX_MAXNOSIL)

Supported on Springware boards only.

Non-silence is the absence of silence: noise or meaningful sound, such as a person speaking. This condition is enabled by setting the tp_length field of a DV_TPT to a specific period of time. When non-silence is detected for this length of time, the I/O function will terminate. This termination condition is frequently used to detect dial tone, or the howler tone that is used by central offices to indicate that a phone has been off-hook for an extended 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_MAXNOSIL termination reason is returned from **ATDX_TERMMSK( )**.

maximum length of silence (DX_MAXSIL)

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( )**.

pattern of silence and non-silence (DX_PMON and DX_PMOFF)

Supported on Springware boards only.

A known pattern of silence and non-silence can terminate a function. A pattern can be specified by using DX_PMON and DX_PMOFF in the tp_termno field in two separate DV_TPT structures, where one represents a period of silence and one represents a period of non-silence. When this termination condition is met, a TM_PATTERN termination reason is returned from **ATDX_TERMMSK( )**.

DX_PMOFF and DX_PMON termination conditions must be used together. The DX_PMON terminating condition must directly follow the DX_PMOFF terminating condition. A combination of both DV_TPT structures using these conditions is used to form a single termination condition.

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( )**.

On HMP Software, using more than one DV_TPT structure for detecting different digits is not supported. Instead, use one DV_TPT structure, set DX_DIGMASK in the tp_termno field, and bitwise-OR "DM_1 | DM_2" in the tp_length field. For uniformity, it is also strongly recommended to use the same method to detect different digits on Springware boards.

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)

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.

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.

6.1.5 Working with User-Defined I/O Functions

Two library functions can be used to install user-defined I/O functions, also called user I/O functions or UIO: **dx_setuio**( ) and **dx_setdevuio**( ).

The following cautions apply when working with user I/O functions:

- Do not include sleeps, critical sections, or any other delays in the user I/O function.
- Do not call any other Dialogic function inside the user I/O function. One exception is the **ec_getblkinfo( )** function which is called from within a user I/O function. For more information on this function, see the *Dialogic® Continuous Speech Processing API Library Reference*.

The reason for these cautions is as follows. On Springware boards, while the user I/O function is executing, the Dialogic® Standard Runtime Library (SRL) is blocked and cannot process further messages from the driver. Data will be lost if the driver cannot hand off messages to the SRL. On HMP Software, you may see chopped audio or underruns. In all cases, be aware that the risk of underruns increases as density rises.

6.2 Additional Considerations

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

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

6.2.1 Multithreading and Multiprocessing

The Dialogic® Voice API supports multithreading and multiprocessing on the board level but not on the channel level on Dialogic® HMP Software.

The following restrictions apply:

- A channel can only be opened in one process at a time; the same channel cannot be used by more than one process concurrently. However, multiple processes can access different sets of channels. Ensure that each process is provided with a unique set of devices to manipulate.
- If a channel is opened in process A and then closed, process B is allowed to open the same channel. However, you should avoid this type of sequence. 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, unpredictable results may occur.
- Multiple processes that define tones (GTD or GTG) do not share tone definitions in the firmware. For example, if you define tone A in process 1 for channel dxxxB1C1 and the same tone A in process 2 for channel dxxxB1C1 , two firmware tones are consumed. 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. For more information, see Chapter 12, “Global Tone Detection and Generation, and Cadenced Tone Generation”.

It is recommended that you develop your application using a limited number of total threads rather than a single thread per channel. For more information on programming models and performance considerations, see the *Dialogic® Standard Runtime Library API Programming Guide*.

Note: The continuous speech processing architecture allows a voice channel to be shared between processes or applications on Dialogic® JCT (Springware) boards or on Dialogic® HMP Software (starting with Dialogic® Host Media Processing Software Release 1.3 for Windows®), providing one process does the play activity and the other process does the record/stream activity. Other CSP scenarios are **not** supported, such as playing or recording/streaming from both processes.

6.2.2 Device Discovery

Applications that use both HMP Software devices and Springware devices must have a way of differentiating what type of device is to be opened. The TDM bus routing functions such as **dx_getctinfo( )** provide a programming solution. HMP Software devices are identified by the CT_DFHMPDM3 value in the ct_devfamily field of the CT_DEVINFO structure. Springware devices are identified by CT_DFD41E in the ct_devfamily field. See the *Dialogic® Voice API Library Reference* for details on the CT_DEVINFO structure.

Note: Use SRL device mapper functions to return information about the structure of the system. For information on these functions, see the *Dialogic® Standard Runtime Library API Library Reference*.

The following procedure shows how to initialize an application and perform device discovery when the application supports both HMP Software devices and Springware boards.

1. Open the first voice channel device on the first voice board in the system with **dx_open( )**.
2. Call **dx_getctinfo( )** and check the CT_DEVINFO.ct_devfamily value.
3. If ct_devfamily is CT_DFHMPDM3, then flag all the voice channel devices associated with the board as HMP Software type. If ct_devfamily is CT_DFD41E, then flag all the voice channel devices associated with the board as Springware type.
4. Close the voice channel with **dx_close( )**.
5. Repeat steps 1 to 4 for each voice board.

6.2.3 Device Initialization Hint

The xx_open( ) functions for the voice (dx), Global Call (gc), network (dt), and fax (fx) APIs are asynchronous for HMP Software devices (on Springware boards, these functions are synchronous). This should usually have no impact on an application, except in cases where a subsequent function calls on an HMP Software 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( )** on an HMP Software device followed by **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 any internal delays.

  This change is not necessary for all applications, but if you experience poor initialization performance, you can gain back speed by using this hint.
- Perform device initialization in a single thread. 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.

6.2.4 Tone Detection Considerations

The following consideration applies to tone detection on HMP Software:

- Digits will not always be cleared by the time the **dx_clrdigbuf( )** function returns, because processing may continue on the board even after the function returns. For this reason, careful consideration should be given when using this function before or during a section where digit detection or digit termination is required; the digit may be cleared after the function has returned or possibly during the next function call.

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.

There are two forms of call progress analysis:

Perfect Call call progress analysis
: Also called enhanced call progress analysis. Uses an improved method of signal identification and can detect fax machines and answering machines. You should design all new applications using Perfect Call call progress analysis.

  Note: In this document, the term call progress analysis refers to Perfect Call call progress analysis unless stated otherwise.

Basic call progress analysis
: Supported on Springware boards for backward compatibility only.

  Provides backward compatibility for older applications written before Perfect Call call progress analysis became available. It is strongly recommended that you do not design new applications using basic call progress analysis.

If your application also uses the Dialogic® Global Call API, see the Global Call documentation set for call progress analysis special considerations. 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 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 basic call progress analysis (supported on Springware boards only). In basic call progress analysis, cadence detection is the sole means of detecting a no ringback, busy, or no answer.

Figure 1. Basic Call Progress Analysis Components (Springware only)

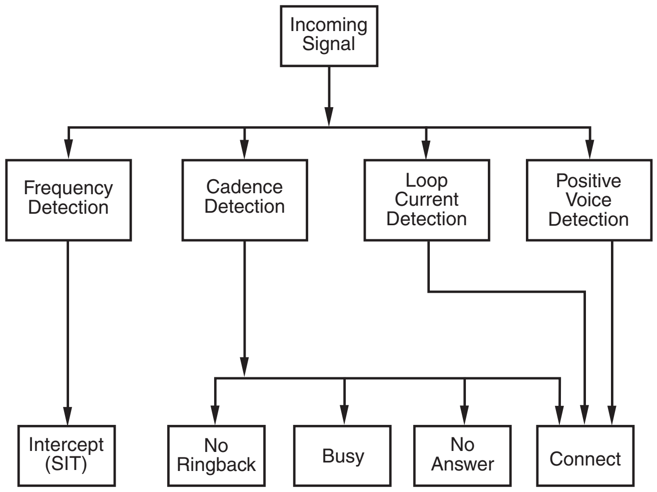
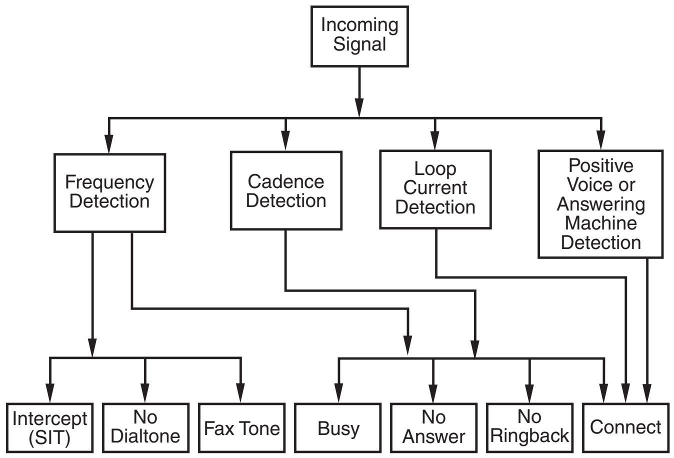


Figure 2 illustrates the components of Perfect Call call progress analysis. These components can operate simultaneously. Perfect Call call progress analysis uses cadence detection plus frequency detection to identify these signals plus fax machine tones. A connect can be detected through the complementary methods of cadence detection, frequency detection, positive voice detection, and positive answering machine detection.

Note: Loop current detection shown in Figure 2 is not applicable to Dialogic® HMP voice devices.

Figure 2. Perfect Call Call Progress Analysis Components



7.4 Call Progress Analysis Errors

If **ATDX_CPTERM( )** returns CR_ERROR, you can use **ATDX_CPERROR( )** to determine the call progress analysis error that occurred.

7.5 Using Call Progress Analysis on HMP Voice Devices

The following topics provide information on how to use call progress analysis on Dialogic® HMP voice devices:

- Call Progress Analysis Rules
- Initiating Call Progress Analysis
- Setting Up Call Progress Analysis Parameters
- Executing a Dial Function
- Determining the Outcome of a Call
- Obtaining Additional Call Outcome Information

7.5.1 Call Progress Analysis Rules

The following rules apply to the use of call progress analysis on Dialogic® HMP voice devices:

- For IP ports using IP protocols:
  - Pre-connect is typically provided by the protocol via the Dialogic® Global Call API or via a native IP protocol API.
  - In general the **dx_dial( )** function does not need to be used for pre-connect call progress. *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 on Dialogic® HMP voice devices.

Table 3. Call Progress Analysis Support with dx_dial( ) (HMP Voice Devices)

CPA Feature	dx_dial( ) support
Busy	Yes
No ringback	Yes
SIT frequency detection	Yes
No answer	Yes
Cadence break	Yes
Loop current detection	No
Dial tone detection	Yes
Fax tone detection	Yes

Table 3. Call Progress Analysis Support with dx_dial( ) (HMP Voice Devices) (Continued)

CPA Feature	dx_dial( ) support
Positive Voice Detection (PVD)	Yes
Positive Answering Machine Detection (PAMD)	Yes

7.5.2 Initiating Call Progress Analysis

Review the information in Section 7.5.1, "Call Progress Analysis Rules", on page 37.

If you choose to use the voice API for call progress analysis on HMP voice devices, follow these steps 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 *Dialogic® Voice API Library Reference*.

7.5.3 Setting Up Call Progress Analysis Parameters

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 on HMP voice devices:

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.

7.5.4 Executing a Dial Function

To use call progress analysis on HMP voice devices, 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 Dialogic® 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.* On HMP voice devices, **dx_dial( )** cannot be used to start an outbound call; instead use the Dialogic® Global Call API.

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

7.5.5 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.

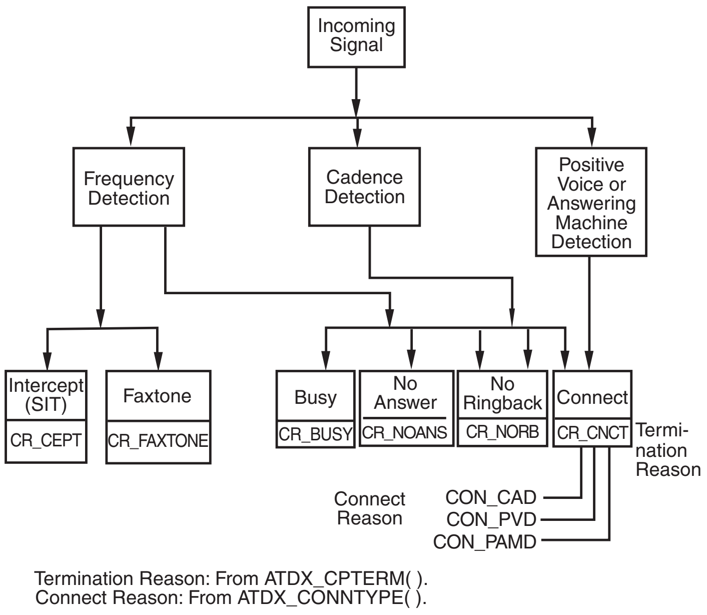
CR_NOANS
: Called line did not answer.

CR_NORB
: No ringback on called line.

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

Figure 3 illustrates the possible outcomes of call progress analysis on HMP voice devices.

Figure 3. Call Outcomes for Call Progress Analysis (HMP Voice Devices)



7.5.6 Obtaining Additional Call Outcome Information

To obtain additional call progress analysis information on HMP voice devices, 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.

7.6 Call Progress Analysis Tone Detection on HMP Voice Devices

The following topics discuss tone detection used in call progress analysis on HMP voice devices:

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

7.6.1 Tone Detection Overview

On HMP voice devices, 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".

7.6.2 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 for HMP voice devices. Tone identifiers are returned by the **ATDX_CRTNID( )** function.

TID_BUSY1	First signal busy
TID_BUSY2	Second signal busy
TID_DIAL_INTL	International dial tone
TID_DIAL_LCL	Local dial tone
TID_DISCONNECT	Disconnect tone (post-connect)
TID_FAX1	First fax or modem tone
TID_FAX2	Second fax or modem tone
TID_RNGBK1	Ringback (detected as single tone)

TID_RNGBK2
: Ringback (detected as dual tone)

TID_SIT_ANY
: Catch all (returned for a Special Information Tone sequence or SIT sequence that falls outside the range of known default SIT sequences)

TID_SIT_INEFFECTIVE_OTHER or
TID_SIT_IO
: Ineffective other SIT sequence

TID_SIT_NO_CIRCUIT or
TID_SIT_NC
: No circuit found SIT sequence

TID_SIT_NO_CIRCUIT_INTERLATA or
TID_SIT_NC_INTERLATA
: InterLATA no circuit found SIT sequence

TID_SIT_OPERATOR_INTERCEPT or
TID_SIT_IC
: Operator intercept SIT sequence

TID_SIT_REORDER_TONE or
TID_SIT_RO
: Reorder (system busy) SIT sequence

TID_SIT_REORDER_TONE_INTERLATA or
TID_SIT_RO_INTERLATA
: InterLATA reorder (system busy) SIT sequence

TID_SIT_VACANT_CIRCUIT or
TID_SIT_VC
: Vacant circuit SIT sequence

Some of these tone identifiers may be used as input to function calls to change the tone definitions. For more information, see Section 7.9, “Modifying Default Call Progress Analysis Tone Definitions on HMP Voice Devices”, on page 47.

7.6.3 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 7.5.3, “Setting Up Call Progress Analysis Parameters”, on page 38.

The following DX_CAP fields govern ringback behavior for HMP voice devices:

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).

7.6.4 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. **ATDX_CRTNID( )** identifies which busy tone was detected.

To enable busy tone detection, turn on SIT frequency detection in the DX_CAP ca_intflg field. For details, see Section 7.5.3, “Setting Up Call Progress Analysis Parameters”, on page 38.

7.6.5 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. **ATDX_CRTNID( )** identifies which fax or modem tone was detected.

To enable fax or modem tone detection, use the ca_intflg field of the DX_CAP structure. For details, see Section 7.5.3, “Setting Up Call Progress Analysis Parameters”, on page 38.

7.6.6 SIT Frequency Detection

Special Information Tone (SIT) frequency detection is a component of call progress analysis. On HMP voice devices, 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 7.5.3, “Setting Up Call Progress Analysis Parameters”, on page 38.

Table 4 shows default tone definitions for SIT sequences used on HMP voice devices. 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. N/A means “not applicable.”

Table 4. Special Information Tone Sequences (HMP Voice Devices)

SIT		1st Segment		2nd Segment		3rd Segment	
Tone ID	**Description**	**Freq.**	**Time**	**Freq.**	**Time**	**Freq.**	**Time**
TID_SIT_NC	No Circuit Found	950/1020	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	950/1020	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
TID_SIT_NC_ INTERLATA	InterLATA No Circuit Found	874/955	32/45	1310/1430	32/45	1740/1850	N/A
TID_SIT_RO_ INTERLATA	InterLATA Reorder (system busy)	950/1020	15/30	1310/1430	32/45	1740/1850	N/A
TID_SIT_IO	Ineffective Other	874/955	32/45	1400/1450	15/30	1740/1850	N/A
TID_SIT_ANY	Catch all tone definition	Open	Open	Open	Open	1725/1825	N/A

The following considerations apply to SIT sequences on HMP voice devices:

- 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 7.9.4, “Rules for Using a Single Tone Proxy for a Dual Tone”, on page 49.
- These tone IDs have aliases:
  - TID_SIT_NO_CIRCUIT (TID_SIT_NC)
  - TID_SIT_OPERATOR_INTERCEPT (TID_SIT_IC)
  - TID_SIT_VACANT_CIRCUIT (TID_SIT_VC)
  - TID_SIT_REORDER_TONE (TID_SIT_RO)
  - TID_SIT_NO_CIRCUIT_INTERLATA (TID_SIT_NC_INTERLATA)
  - TID_SIT_REORDER_TONE_INTERLATA (TID_SIT_RO_INTERLATA)
  - TID_SIT_INEFFECTIVE_OTHER (TID_SIT_IO)
- Default SIT definitions can be modified, **except for** the following SIT sequences: TID_SIT_ANY, TID_SIT_IO, TID_SIT_NC_INTERLATA, and TID_SIT_RO_INTERLATA. For more information, see Section 7.9, “Modifying Default Call Progress Analysis Tone Definitions on HMP Voice Devices”, on page 47.
- For TID_SIT_ANY, the frequency and time of the first and second segments are open; that is, they are ignored. Only the frequency of the third segment is relevant. This catch-all SIT sequence definition is intended to cover SIT sequences that fall outside the range of the defined SIT sequences.

7.7 Media Tone Detection on HMP Voice Devices

Media tone detection in call progress analysis on HMP voice devices is discussed in the following topics:

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

7.7.1 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 7.5.3, "Setting Up Call Progress Analysis Parameters", on page 38.

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.

7.7.2 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 7.5.3, "Setting Up Call Progress Analysis Parameters", on page 38.

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 on HMP voice devices:

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).

7.8 Default Call Progress Analysis Tone Definitions on HMP Voice Devices

Table 5 provides the range of values for default tone definitions on HMP voice devices. 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.*** On HMP voice devices, voice API functions are provided to manipulate the tone definitions in this table (see Section 7.9, “Modifying Default Call Progress Analysis Tone Definitions on HMP Voice Devices”, on page 47). However, not all the functionality provided by these tones is available through the Dialogic® Voice API. You may need to use the Dialogic® 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 7.9.4, “Rules for Using a Single Tone Proxy for a Dual Tone”, on page 49.

Table 5. Default Call Progress Analysis Tone Definitions (HMP Software)

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_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

Table 5. Default Call Progress Analysis Tone Definitions (HMP Software) (Continued)

Tone ID	Freq1 (in Hz)	Freq2 (in Hz)	On Time (in 10 msec)	Off Time (in 10 msec)	Reps	Twin Tone Freq (Hz)
TID_DIAL_INTL	300 - 380	400 - 480	100 - 0	-	1	300 - 480
TID_DIAL_LCL	300 - 380	400 - 480	10 - 0	-	1	0
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

7.9 Modifying Default Call Progress Analysis Tone Definitions on HMP Voice Devices

On HMP Software, call progress analysis tones are maintained in the software. 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

7.9.1 API Functions for Manipulating Tone Definitions

The following Dialogic® Voice API functions are used to manipulate the default tone definitions shown in Table 5, "Default Call Progress Analysis Tone Definitions (HMP Software)", on page 46 and some, but not all, of the default tone definitions shown in Table 4, "Special Information Tone Sequences (HMP Voice Devices)", on page 44.

Note: Default SIT definitions can be modified, **except for** the following SIT sequences: TID_SIT_ ANY, TID_SIT_IO, TID_SIT_NC_INTERLATA, and TID_SIT_RO_INTERLATA.

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

7.9.2 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.

7.9.3 Rules for Modifying a Tone Definition

Consider the following rules and guidelines for modifying default tone definitions on HMP voice devices, using the Dialogic® 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 (HMP Voice Devices)", on page 44 and Table 5, "Default Call Progress Analysis Tone Definitions (HMP Software)", on page 46.
- These three Dialogic® 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 Dialogic® Voice API. You may need to use the Dialogic® 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.

7.9.4 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 on HMP voice devices:

- 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.

7.9.5 Steps to Modify a Tone Definition

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

Note: This procedure assumes that you have already opened the board device handle in your application. To get the board name in the form *brdBn*, use the **SRLGetPhysicalBoardName( )** function. This function and other device mapper functions return information about the structure of the system. For more information, see the *Dialogic® Standard Runtime Library API Library Reference*.

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.

7.10 Using Call Progress Analysis on Springware Boards

The following topics provide information on how to use call progress analysis when making an outbound call on Springware boards:

- Initiating Call Progress Analysis
- Setting Up Call Progress Analysis Parameters
- Enabling Call Progress Analysis
- Executing a Dial Function
- Determining the Outcome of a Call
- Obtaining Additional Call Outcome Information

7.10.1 Initiating Call Progress Analysis

Perform the following procedure to initiate an outbound call with call progress analysis on Springware boards:

1. Set up the call analysis parameter structure (DX_CAP), which contains parameters to control the operation of call progress analysis, such as frequency detection, cadence detection, loop current, positive voice detection, and positive answering machine detection.
2. On Springware boards, enable call progress analysis on a specified channel using **dx_initcallp( )**. Modify tone definitions as appropriate.
3. Call **dx_dial( )** to start an outbound call.
4. Use the **ATDX_CPTERM( )** extended attribute function to determine the outcome of the call.
5. Obtain additional termination, frequency, or cadence information (such as the length of the salutation) 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 *Dialogic® Voice API Library Reference*.

7.10.2 Setting Up Call Progress Analysis Parameters

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 frequency detection, positive voice detection (PVD), and positive answering machine detection (PAMD).

To customize the parameters for your environment, you must set up the call progress analysis parameter structure before calling a dial function.

To set up the DX_CAP structure for call progress analysis on Springware boards:

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.
   - DX_OPTNOCON. Enables SIT frequency detection and returns an “intercept” immediately after detecting a valid frequency.
   - DX_PVDENABLE. Enables PVD and fax tone detection.
   - DX_PVDOPTNOCON. Enables PVD, DX_OPTNOCON, and fax tone detection.
   - DX_PAMDENABLE. Enables PAMD, PVD, and fax tone detection.
   - DX_PAMDOPTEN. Enables PAMD, PVD, DX_OPTNOCON, and fax tone detection.

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

7.10.3 Enabling Call Progress Analysis

On Springware boards, call progress analysis is activated on a per-channel basis and is initiated using **dx_initcallp( )**.

Perform the following steps to enable call progress analysis. This procedure needs to be followed only once per channel; thereafter, any outgoing calls made using a dial function will benefit from call progress analysis.

1. Make any desired modifications to the default dial tone, busy tone, fax tone, and ringback signal definitions using the **dx_chgfreq( )**, **dx_chgdur( )**, and **dx_chgrepcnt( )** functions.

2. Call **dx_deltones( )** to clear all tone templates remaining on the channel. Note that this function deletes all global tone definition (GTD) tones for the given channel, and not just those involved with call progress analysis.
3. Execute the **dx_initcallp( )** function to activate call progress analysis. Call progress analysis stays active until **dx_deltones( )** is called.

The **dx_initcallp( )** function initializes call progress analysis on the specified channel using the current tone definitions. Once the channel is initialized with these tone definitions, this initialization cannot be altered. The only way to change the tone definitions in effect for a given channel is to issue a **dx_deltones( )** call for that channel, then invoke another **dx_initcallp( )** with different tone definitions.

7.10.4 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 the Dialogic® 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.

7.10.5 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). Extended attribute functions provide information on detected frequencies and duration.

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.

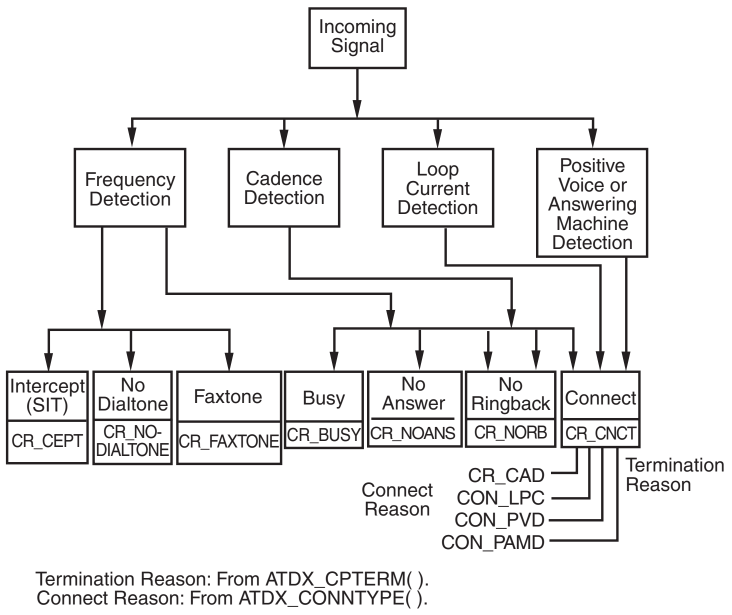
CR_NODIALTONE
: Timeout occurred while waiting for dial tone.

CR_NORB
: No ringback on called line.

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

Figure 4 illustrates the possible outcomes of call progress analysis on Dialogic® Springware boards.

Figure 4. Call Outcomes for Call Progress Analysis (Springware)



7.10.6 Obtaining Additional Call Outcome Information

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

ATDX_ANSRSIZ( )
: Returns duration of answer.

ATDX_CPERROR( )
: Returns call analysis error.

ATDX_CPTERM( )
: Returns last call analysis termination.

ATDX_CONNTYPE( )
: Returns connection type

ATDX_CRTNID( )
Returns the identifier of the tone that caused the most recent call progress analysis termination.

ATDX_DTNFAIL( )
Returns the dial tone character that indicates which dial tone call progress analysis failed to detect.

ATDX_FRQDUR( )
Returns duration of first frequency detected.

ATDX_FRQDUR2( )
Returns duration of second frequency detected.

ATDX_FRQDUR3( )
Returns duration of third frequency detected.

ATDX_FRQHZ( )
Returns frequency detected in Hz of first detected tone.

ATDX_FRQHZ2( )
Returns frequency of second detected tone.

ATDX_FRQHZ3( )
Returns frequency of third detected tone.

ATDX_LONGLOW( )
Returns duration of longer silence.

ATDX_FRQOUT( )
Returns percent of frequency out of bounds.

ATDX_SHORTLO( )
Returns duration of shorter silence.

ATDX_SIZEHI( )
Returns duration of non-silence.

7.11 Call Progress Analysis Tone Detection on Springware Boards

Tone detection in Perfect Call call progress analysis differs from the one in basic call progress analysis. The following topics discuss tone detection in Perfect Call call progress analysis on Dialogic® Springware boards:

- Tone Detection Overview
- Types of Tones
- Dial Tone Detection
- Ringback Detection
- Busy Tone Detection
- Fax or Modem Tone Detection
- Loop Current Detection

7.11.1 Tone Detection Overview

Perfect Call 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".

Unlike basic call progress analysis, which uses fields in the DX_CAP structure to store signal cadence information, Perfect Call call progress analysis uses tone definitions which are contained in the voice driver itself. Functions are available to modify these default tone definitions.

7.11.2 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 on Springware boards. Tone identifiers are returned by the **ATDX_CRTNID( )** function.

TID_BUSY1	Busy signal
TID_BUSY2	Alternate busy signal
TID_DIAL_INTL	International dial tone
TID_DIAL_LCL	Local dial tone
TID_DIAL_XTRA	Special (extra) 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 7.14, "Modifying Default Call Progress Analysis Tone Definitions on Springware Boards", on page 60.

7.11.3 Dial Tone Detection

Wherever call progress analysis is in effect, a dial string for an outgoing call may specify special ASCII characters that instruct the system to wait for a certain kind of dial tone. The following additional special characters may appear in a dial string:

L
: wait for a local dial tone

I
: wait for an international dial tone

X
: wait for a special ("extra") dial tone

The tone definitions for each of these dial tones is set for each channel at the time of the **dx_initcallp( )** function. In addition, the following DX_CAP fields identify how long to wait for a dial tone, and how long the dial tone must remain stable.

ca_dtn_pres
: Dial Tone Present: the length of time that the dial tone must be continuously present (in 10 msec units). If a dial tone is present for this amount of time, dialing of the dial string proceeds. Default value: 100 (one second).

ca_dtn_npres
: Dial Tone Not Present: the length of time to wait before declaring the dial tone not present (in 10 msec units). If a dial tone of sufficient length (ca_dtn_pres) is not found within this period of time, call progress analysis terminates with the reason CR_NODIALTONE. The dial tone character (**L**, **I**, or **X**) for the missing dial tone can be obtained using **ATDX_DTNFAIL( )**. Default value: 300 (three seconds).

ca_dtn_deboff
: Dial Tone Debounce: the maximum duration of a break in an otherwise continuous dial tone before it is considered invalid (in 10 msec units). This parameter is used for ignoring short drops in dial tone. If a drop longer than ca_dtn_deboff occurs, then dial tone is no longer considered present, and another dial tone must begin and be continuous for ca_dtn_pres. Default value: 10 (100 msec).

7.11.4 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 7.10.2, "Setting Up Call Progress Analysis Parameters", on page 51.

The following DX_CAP fields govern ringback behavior:

ca_stdely
: Start Delay: the delay after dialing has been completed before starting cadence detection, frequency detection, and positive voice detection (in 10 msec units). Default: 25 (0.25 seconds).

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).

ca_maxintering
: Maximum Inter-ring: the maximum length of time to wait between consecutive ringback signals (in 10 msec units). If this duration is exceeded, call progress analysis is terminated with the reason CR_CNCT (connected). Default value: 800 (8 seconds).

7.11.5 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. **ATDX_CRTNID( )** identifies which busy tone was detected.

To enable busy tone detection, turn on SIT frequency detection in the DX_CAP ca_intflg field. For details, see Section 7.10.2, "Setting Up Call Progress Analysis Parameters", on page 51.

7.11.6 Fax or Modem Tone Detection

Two tones are defined: 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. **ATDX_CRTNID( )** identifies which fax or modem tone was detected.

To enable fax or modem tone detection, turn on SIT frequency detection in the DX_CAP ca_intflg field. For details, see Section 7.10.2, "Setting Up Call Progress Analysis Parameters", on page 51.

7.11.7 Loop Current Detection

Loop current detection, available through the **dx_dial( )** function, is supported on Springware boards only.

Some telephone systems return a momentary drop in loop current when a connection has been established (**answer supervision**). Loop current detection returns a **connect** when a transient loop current drop is detected.

In some environments, including most PBXs, answer supervision is not provided. In these environments, Loop current detection will not function. Check with your Central Office or PBX supplier to see if answer supervision based on loop current changes is available.

In some cases, the application may receive one or more transient loop current drops before an actual connection occurs. This is particularly true when dialing long-distance numbers, when the call may be routed through several different switches. Any one of these switches may be capable of generating a momentary drop in loop current.

To disable loop current detection, set DX_CAP ca_lcdly to -1.

Note: For applications that use loop current reversal to signal a disconnect, it is recommended that DXBD_MINLCOFF be set to 2 to prevent Loop Current On and Loop Current Off from being reported instead of Loop Current Reversal.

7.11.7.1 Loop Current Detection Parameters Affecting a Connect

To prevent detecting a connect prematurely or falsely due to a spurious loop current drop, you can delay the start of loop current detection by using the parameter ca_lcdly.

Loop current detection returns a **connect** after detecting a loop current drop. To allow the person who answered the phone to say "hello" before the application proceeds, you can delay the return of the **connect** by using the parameter ca_lcdly1.

ca_lcdly
: Loop Current Delay: the delay after dialing has been completed and before beginning Loop Current Detection. To disable loop current detection, set to -1. Default: 400 (10 msec units).

ca_lcdly1
: Loop Current Delay 1: the delay after loop current detection detects a transient drop in loop current and before call progress analysis returns a connect to the application. Default: 10 (10 msec units).

If the **ATDX_CONNTYPE( )** function returns CON_LPC, the connect was due to loop current detection.

Note: When a connect is detected through positive voice detection or loop current detection, the DX_CAP parameters ca_hedge, ca_ansrdgl, and ca_maxansr are ignored.

7.12 Media Tone Detection on Springware Boards

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

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

7.12.1 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 7.10.2, “Setting Up Call Progress Analysis Parameters”, on page 51.

PVD is especially useful in those situations where answer supervision is not available for loop current detection to identify a connect, 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.

7.12.2 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 7.10.2, “Setting Up Call Progress Analysis Parameters”, on page 51.

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 on Springware boards:

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_FULL

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

ca_pamd_qtemp
: PAMD Qualification Template: the algorithm to use in PAMD. At present there is only one template: PAMD_QUAL1TMP. This parameter must be set to this value.

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).

ca_pamd_minring
: minimum allowable ring duration for positive answering machine detection. Default Value: 190 (in 10 msec units).

7.13 Default Call Progress Analysis Tone Definitions on Springware Boards

Table 6 provides call progress analysis default tone definitions for Springware boards. Frequencies are specified in Hz, durations in 10 msec units, and repetitions in integers. For information on manipulating these tone definitions, see Section 7.14, "Modifying Default Call Progress Analysis Tone Definitions on Springware Boards", on page 60.

Table 6. Default Call Progress Analysis Tone Definitions (Springware)

Tone ID	Freq1 (in Hz)	Freq2 (in Hz)	On Time (in 10 msec)	Off Time (in 10 msec)	Reps
TID_BUSY1	500 ± 200		55 ± 40	55 ± 40	4
TID_BUSY2	500 ± 200	500 ± 200	55 ± 40	55 ± 40	4
TID_DIAL_LCL	400 ± 125				
TID_DIAL_INTL	402 ± 125				
TID_DIAL_XTRA	401 ± 125				
TID_DISCONNECT	500 ± 200	500 ± 200	55 ± 40	55 ± 40	4
TID_FAX1	1650 ± 100		20 ± 20		
TID_FAX2	1100 ± 50		25 ± 25		
TID_RNGBK1	450 ± 150		130 ± 105	580 ± 415	
TID_RNGBK2	450 ± 150	450 ± 150	130 ± 105	580 ± 415	

7.14 Modifying Default Call Progress Analysis Tone Definitions on Springware Boards

On Dialogic® Springware boards, call progress analysis makes use of global tone detection (GTD) tone definitions for three different types of dial tones, two busy tones, one ringback tone, and two fax tones. The tone definitions specify the frequencies, durations, and repetition counts necessary to identify each of these signals. Each signal may consist of a single tone or a dual tone.

The voice driver contains default definitions for each of these tones. The default definitions will allow applications to identify the tones correctly in most countries and for most switching equipment. However, if a situation arises in which the default tone definitions are not adequate, three functions are provided to modify the standard tone definitions:

dx_chgfreq( )
: specifies frequencies and tolerances for one or both frequencies of a single- or dual-frequency tone

dx_chgdur( )
: specifies the cadence (on time, off time, and acceptable deviations) for a tone

dx_chgrepcnt( )
: specifies the repetition count required to identify a tone

These functions can be used to modify the tone definitions shown in Table 6, "Default Call Progress Analysis Tone Definitions (Springware)", on page 60. These functions only change the tone definitions; they do not alter the behavior of call progress analysis itself. When the **dx_initcallp( )** function is invoked to activate call progress analysis on a particular channel, it uses the current tone definitions to initialize that channel. Multiple calls to **dx_initcallp( )** may therefore use varying tone definitions, and several channels can operate simultaneously with different tone definitions.

For more information on tones and tone detection, see Section 7.11, "Call Progress Analysis Tone Detection on Springware Boards", on page 54.

Note: The Learn Mode API and Tone Set File (TSF) API provide a more comprehensive way to manage call progress tones, in particular the unique call progress tones produced by PBXs, key systems, and PSTNs. Applications can learn tone characteristics using the Learn Mode API. Information on several different tones forms one tone set. Tone sets can be written to a tone set file using the Tone Set File API. For more information, see the *Learn Mode and Tone Set File API Software Reference for Linux and Windows Operating Systems*.

7.15 SIT Frequency Detection on Springware Boards

Special Information Tone (SIT) frequency detection is a component of call progress analysis on Springware boards. The following topics provide more information on this component:

- Tri-Tone SIT Sequences
- Setting Tri-Tone SIT Frequency Detection Parameters
- Obtaining Tri-Tone SIT Frequency Information
- Global Tone Detection Tone Memory Usage
- Frequency Detection Errors
- Setting Single Tone Frequency Detection Parameters
- Obtaining Single Tone Frequency Information

7.15.1 Tri-Tone SIT Sequences

SIT frequency detection operates simultaneously with all other call progress analysis detection methods. The purpose of frequency detection is to detect the tri-tone special information tone (SIT) sequences and other single-frequency tones. Detection of a SIT sequence indicates an operator intercept or other problem in completing the call.

SIT frequency detection can detect virtually any single-frequency tone below 2100 Hz and above 300 Hz.

Table 7 provides tone information for the four SIT sequences on Springware boards. The frequencies are represented in Hz and the length of the signal is in 10 msec units. The length of the first segment is not dependable; often it is shortened or cut.

Table 7. Special Information Tone Sequences (Springware)

SIT		1st Segment		2nd Segment		3rd Segment	
Name	**Description**	**Freq.**	**Len.**	**Freq.**	**Len.**	**Freq.**	**Len.**
NC	No Circuit Found	985	38	1429	38	1777	38
IC	Operator Intercept	914	27	1371	27	1777	38
VC	Vacant Circuit	985	38	1370	27	1777	38
RO	Reorder (system busy)	914	27	1429	38	1777	38

7.15.2 Setting Tri-Tone SIT Frequency Detection Parameters

Frequency detection on Springware voice boards is designed to detect all three tones in a tri-tone SIT sequence. To detect all three tones in a SIT sequence, you must specify the frequency detection parameters in the DX_CAP for all three tones in the sequence.

To detect all four tri-tone SIT sequences:

1. Set an appropriate frequency detection range in the DX_CAP to detect each tone across all four SIT sequences. Set the first frequency detection range to detect the first tone for all four SIT sequences (approximately 900 to 1000 Hz). Set the second frequency detection range to detect the second tone for all four SIT sequences (approximately 1350 to 1450 Hz). Set the third frequency detection range to detect the third tone for all four SIT sequences (approximately 1725 to 1825 Hz).
2. Set an appropriate detection time using the ca_timefrq and ca_mxtimefrq parameters to detect each tone across all four SIT sequences. For each tone, set ca_timefrq to 5 and ca_mxtimefrq to 50 to detect all SIT tones. The tones range in length from 27 to 38 (in 10 msec units), with some tones occasionally cut short by the Central Office.

   Note: Occasionally, the first tone can also be truncated by a delay in the onset of call progress analysis due to the setting of ca_stdely.
3. After a SIT sequence is detected, **ATDX_CPTERM( )** will return CR_CEPT to indicate an operator intercept, and you can determine which SIT sequence was detected by obtaining the

actual detected frequency and duration for the tri-tone sequence using extended attribute functions. These functions are described in detail in the *Voice API Library Reference*.

The following fields in the DX_CAP are used for frequency detection on voice boards. Frequencies are specified in Hertz, and time is specified in 10 msec units. To enable detection of the second and third tones, you must set the frequency detection range and time for each tone.

General

The following field in the DX_CAP is used for frequency detection on voice boards.

ca_stdely
: Start Delay. The delay after dialing has been completed and before starting frequency detection. This parameter also determines the start of cadence detection and positive voice detection. Note that this can affect detection of the first element of an operator intercept tone.

  Default: 25 (10 msec units).

First Tone

The following fields in the DX_CAP are used for frequency detection for the first tone. Frequencies are specified in Hertz, and time is specified in 10 msec units.

ca_lowerfrq
: Lower bound for first tone in Hz.

  Default: 900.

ca_upperfrq
: Upper bound for first tone in Hz. Adjust higher for additional operator intercept tones.

  Default: 1000.

ca_timefrq
: Minimum time for first tone to remain in bounds. The minimum amount of time required for the audio signal to remain within the frequency detection range for it to be detected. The audio signal must not be greater than ca_upperfrq or lower than ca_lowerfrq for at least the time interval specified in ca_timefrq.

  Default: 5 (10 msec units).

ca_mxtimefrq
: Maximum allowable time for first tone to be present.

  Default: 0 (10 msec units).

Second Tone

The following fields in the DX_CAP are used for frequency detection for the second tone. Frequencies are specified in Hertz, and time is specified in 10 msec units. To enable detection of the second and third tones, you must set the frequency detection range and time for each tone.

Note: This tone is disabled initially and must be activated by the application using these variables.

ca_lower2frq
: Lower bound for second tone in Hz. Default: 0.

ca_upper2frq
: Upper bound for second tone in Hz. Default: 0.

ca_time2frq
: Minimum time for second tone to remain in bounds. Default: 0 (10 msec units).

ca_mxtime2frq
: Maximum allowable time for second tone to be present. Default: 0 (10 msec units).

Third Tone

The following fields in the DX_CAP are used for frequency detection for the third tone. Frequencies are specified in Hertz, and time is specified in 10 msec units. To enable detection of the second and third tones, you must set the frequency detection range and time for each tone.

Note: This tone is disabled initially and must be activated by the application using these variables.

ca_lower3frq
: Lower bound for third tone in Hz. Default: 0.

ca_upper3frq
: Upper bound for third tone in Hz. Default: 0.

ca_time3frq
: Minimum time for third tone to remain in bounds. Default: 0 (10 msec units).

ca_mxtime3frq
: Maximum allowable time for third tone to be present. Default: 0 (10 msec units).

7.15.3 Obtaining Tri-Tone SIT Frequency Information

Upon detection of the specified sequence of frequencies, you can use extended attribute functions to provide the exact frequency and duration of each tone in the sequence. The frequency and duration information will allow exact determination of all four SIT sequences.

On Springware boards, the following extended attribute functions are used to provide information on the frequencies detected by call progress analysis.

ATDX_FRQHZ( )
: Frequency in Hz of the tone detected in the tone detection range specified by the DX_CAP ca_lowerfrq and ca_upperfrq parameters; usually the first tone of an SIT sequence. This function can be called on non-DSP boards.

ATDX_FRQDUR( )
: Duration of the tone detected in the tone detection range specified by the DX_CAP ca_lowerfrq and ca_upperfrq parameters; usually the first tone of an SIT sequence (10 msec units).

ATDX_FRQHZ2( )
: Frequency in Hz of the tone detected in the tone detection range specified by the DX_CAP ca_lower2frq and ca_upper2frq parameters; usually the second tone of an SIT sequence.

ATDX_FRQDUR2( )
Duration of the tone detected in the tone detection range specified by the DX_CAP ca_lower2frq and ca_upper2frq parameters; usually the second tone of an SIT sequence (10 msec units).

ATDX_FRQHZ3( )
Frequency in Hz of the tone detected in the tone detection range specified by the DX_CAP ca_lower3frq and ca_upper3frq parameters; usually the third tone of an SIT sequence.

ATDX_FRQDUR3( )
Duration of the tone detected in the tone detection range specified by the DX_CAP ca_lower3frq and ca_upper3frq parameters; usually the third tone of an SIT sequence (10 msec units).

7.15.4 Global Tone Detection Tone Memory Usage

On Dialogic® Springware boards, if you use call progress analysis to identify the tri-tone SIT sequences, call progress analysis will create tone detection templates internally, and this will reduce the number of tone templates that can be created using Global Tone Detection functions.

Call progress analysis will create one tone detection template for each single-frequency tone with a 100 Hz detection range. For example, if detecting the set of tri-tone SIT sequences (three frequencies) on each of four channels, the number of allowable user-defined tones will be reduced by three per channel.

If you initiate call progress analysis and there is not enough memory to create the SIT tone detection templates internally, you will get a CR_MEMERR error. This indicates that you are trying to exceed the maximum number of tone detection templates. The tone detection range should be limited to a maximum of 100 Hz per tone to reduce the chance of exceeding the available memory.

7.15.5 Frequency Detection Errors

On Dialogic® Springware boards, the frequency detection range specified by the lower and upper bounds for each tone cannot overlap; otherwise, an error will be produced when the driver attempts to create the internal tone detection templates. For example, if ca_upperfrq is 1000 and ca_lower2frq is also 1000, an overlap occurs and will result in an error. Also, the lower bound of each frequency detection range must be less than the upper bound (for example, ca_lower2frq must be less than ca_upper2frq).

7.15.6 Setting Single Tone Frequency Detection Parameters

The following paragraphs describe how to set single tone frequency detection on Dialogic® Springware boards.

Setting single tone frequency detection parameters allows you to identify that a SIT sequence was encountered because one of the tri-tones in the SIT sequence was detected. But frequency detection cannot determine exactly which SIT sequence was encountered, because it is necessary to identify two tones in the SIT sequence to distinguish among the four possible SIT sequences.

The default frequency detection range is 900-1000 Hz, which is set to detect the first tone in any SIT sequence. Because the first tone is often truncated, you may want to increase ca_upperfrq to 1800 Hz so that it includes the third tone. If this results in too many false detections, you can set frequency detection to detect only the third tone by setting ca_lowerfrq to 1750 and ca_upperfrq to 1800.

The following fields in the DX_CAP are used for frequency detection. Frequencies are specified in Hertz, and time is specified in 10 msec units.

ca_stdely
: Start Delay: the delay after dialing has been completed and before starting frequency detection. This parameter also determines the start of cadence detection. Default: 25 (10 msec units).

ca_lowerfrq
: lower bound for tone in Hz. Default: 900.

ca_upperfrq
: upper bound for tone in Hz. Default: 1000.

ca_timefrq
: time frequency. Minimum time for 1st tone in an SIT to remain in bounds. The minimum amount of time required for the audio signal to remain within the frequency detection range specified by ca_upperfrq and ca_lowerfrq for it to be considered valid. Default: 5 (10 msec units)

7.15.7 Obtaining Single Tone Frequency Information

On Dialogic® Springware boards, upon detection of a frequency in the specified range, you can use the **ATDX_FRQHZ**( ) extended attribute function to return the frequency in Hz of the tone detected in the range specified by the DX_CAP ca_lowerfrq and ca_upperfrq parameters. The frequency returned is usually the first tone of an SIT sequence.

7.16 Cadence Detection in Basic Call Progress Analysis on Springware Boards

Cadence detection is a component of basic call progress analysis. The following topics discuss cadence detection and some of the most commonly adjusted cadence detection parameters in basic call progress analysis on Springware boards:

- Overview
- Typical Cadence Patterns
- Elements of a Cadence
- Outcomes of Cadence Detection
- Setting Selected Cadence Detection Parameters
- Obtaining Cadence Information

7.16.1 Overview

Caution: Cadence detection in basic call progress analysis is supported on Springware boards and is provided for backward compatibility purposes only. You should not develop new applications based on basic call progress analysis. Instead you should use Perfect Call call progress analysis. For information on cadence detection in Perfect Call call progress analysis, see Section 7.11, "Call Progress Analysis Tone Detection on Springware Boards", on page 54.

The cadence detection algorithm has been optimized for use in the United States standard network environment.

If your system is operating in another type of environment (such as behind a PBX), you can customize the cadence detection algorithm to suit your system through the adjustment of the cadence detection parameters.

Cadence detection analyzes the audio signal on the line to detect a repeating pattern of sound and silence, such as the pattern produced by a ringback or a busy signal. These patterns are called **audio cadences**. Once a cadence has been established, it can be classified as a single ring, a double ring, or a busy signal by comparing the periods of sound and silence to established parameters.

Note: Sound is referred to as **nonsilence**.

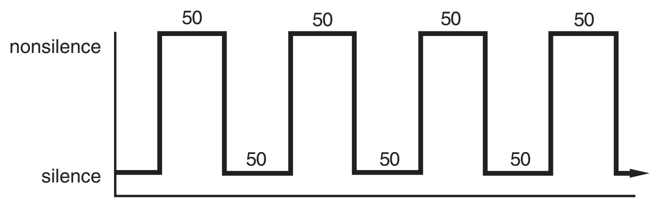
The algorithm used for cadence detection is disclosed and protected under U.S. patent 4,477,698 of Melissa Electronic Labs, and other patents pending.

7.16.2 Typical Cadence Patterns

Note: Cadence detection in basic call progress analysis is supported on Springware boards and is provided for backward compatibility purposes only.

Figure 5, Figure 6, and Figure 7 show some typical cadence patterns for a standard busy signal, a standard single ring, and a double ring.

Figure 5. A Standard Busy Signal



The timings are given in units of 10ms.

Figure 6. A Standard Single Ring

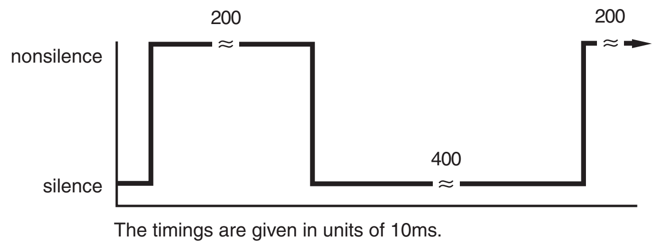
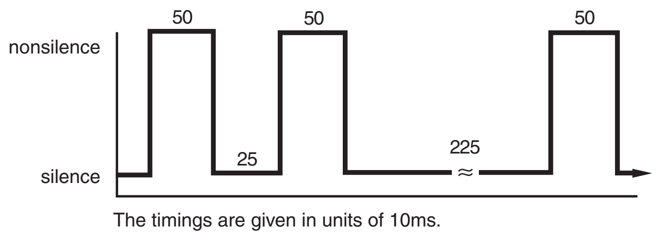


Figure 7. A Type of Double Ring

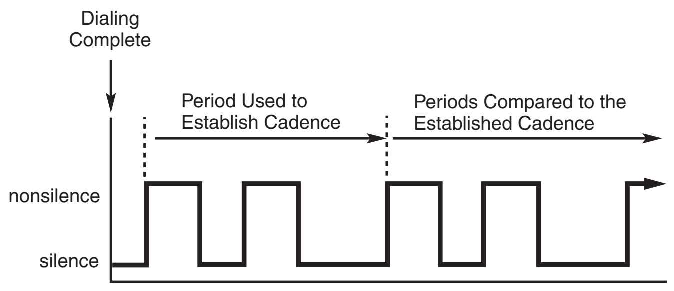


7.16.3 Elements of a Cadence

Note: Cadence detection in basic call progress analysis is supported on Springware boards and is provided for backward compatibility purposes only.

From the preceding cadence examples, you can see that a given cadence may contain two silence periods with different durations, such as for a double ring; but in general, the nonsilence periods have the same duration. To identify and distinguish between the different types of cadences, the voice driver must detect two silence and two nonsilence periods in the audio signal. Figure 8 illustrates cadence detection.

Figure 8. Cadence Detection


Dialing Complete
Period Used to Establish Cadence
Periods Compared to the Established Cadence
nonsilence
silence

Once the cadence is established, the cadence values can be retrieved using the following extended attribute functions:

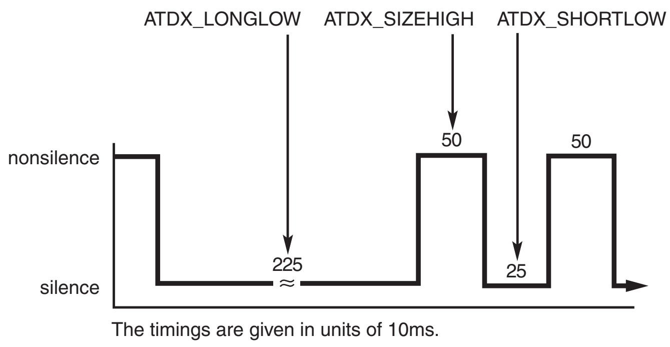
ATDX_SIZEHI( )
: length of the nonsilence period (in 10 msec units) for the detected cadence

ATDX_SHORTLOW( )
: length of the shortest silence period for the detected cadence (in 10 msec units)

ATDX_LONGLOW( )
: length of the longest silence period for the detected cadence (in 10 msec units).

Only one nonsilence period is used to define the cadence because the nonsilence periods have the same duration. Figure 9 shows the elements of an established cadence.

Figure 9. Elements of Established Cadence



The durations of subsequent states are compared with these fields to see if the cadence has been broken.

7.16.4 Outcomes of Cadence Detection

Note: Cadence detection in basic call progress analysis is supported on Springware boards and is provided for backward compatibility purposes only.

Cadence detection can identify the following conditions during the period used to establish the cadence or after the cadence has been established:

- No Ringback
- Connect
- Busy
- No Answer

Although loop current detection and positive voice detection provide complementary means of detecting a connect, cadence detection provides the only way in basic call progress analysis to detect a no ringback, busy, or no answer.

Cadence detection can identify the following conditions during the period used to establish the cadence:

No Ringback
: While the cadence is being established, cadence detection determines whether the signal is continuous silence or nonsilence. In this case, cadence detection returns a **no ringback**, indicating there is a problem in completing the call.

Connect
: While the cadence is being established, cadence detection determines whether the audio signal departs from acceptable network standards for busy or ring signals. In this case, cadence detection returns a **connect**, indicating that there was a “break” from general cadence standards.

Cadence detection can identify the following conditions after the cadence has been established:

Connect
: After the cadence has been established, cadence detection determines whether the audio signal departs from the established cadence. In this case, cadence detection returns a **connect**, indicating that there was a break in the established cadence.

No Answer
: After the cadence has been established, cadence detection determines whether the cadence belongs to a single or double ring. In this case, cadence detection can return a **no answer**, indicating there was no break in the ring cadence for a specified number of times.

Busy
: After the cadence has been established, cadence detection determines whether the cadence belongs to a slow busy signal. In this case, cadence detection can return a **busy**, indicating that the busy cadence was repeated for a specified number of times.

To determine whether the ring cadence is a double or single ring, compare the value returned by the **ATDX_SHORTLOW( )** function to the DX_CAP field ca_lo2rmin. If the **ATDX_SHORTLOW( )** value is less than ca_lo2rmin, the cadence is a double ring; otherwise, it is a single ring.

7.16.5 Setting Selected Cadence Detection Parameters

Note: Cadence detection in basic call progress analysis is supported on Springware boards and is provided for backward compatibility purposes only.

Only the most commonly adjusted cadence detection parameters are discussed here. For a complete listing and description of the DX_CAP data structure, see the *Dialogic® Voice API Library Reference*.

You should only need to adjust cadence detection parameters for network environments that do not conform to the U.S. standard network environment (such as behind a PBX).

7.16.5.1 General Cadence Detection Parameters

The following are general cadence detection parameters in DX_CAP:

ca_stdely
: Start Delay: the delay after dialing has been completed and before starting cadence detection. This parameter also determines the start of frequency detection and positive voice detection. Default: 25 (10 msec units) = 0.25 seconds.

  Be careful with this variable. Setting this variable too small may allow switching transients or, if too long, miss critical signaling.

ca_higltch
: High Glitch: the maximum nonsilence period to ignore. Used to help eliminate spurious nonsilence intervals. Default: 19 (in 10 msec units).

  To eliminate audio signal glitches over the telephone line, the parameters ca_logltch and ca_higltch are used to determine the minimum acceptable length of a valid silence or nonsilence duration. Any silence interval shorter than ca_logltch is ignored, and any nonsilence interval shorter than ca_higltch is ignored.

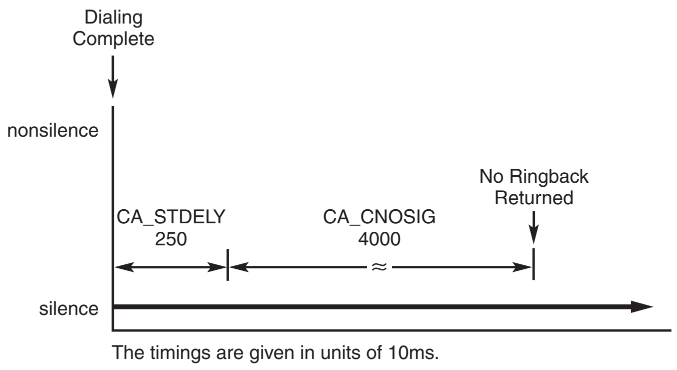
ca_logltch
: Low Glitch: the maximum silence period to ignore. Used to help eliminate spurious silence intervals. Default: 15 (in 10 msec units).

7.16.5.2 Cadence Detection Parameters Affecting a No Ringback

After cadence detection begins, it waits for an audio signal of nonsilence. The maximum waiting time is determined by the parameter ca_cnosig (continuous no signal). If the length of this period of silence exceeds the value of ca_cnosig, a **no ringback** is returned. Figure 10 illustrates this. This usually indicates a dead or disconnected telephone line or some other system malfunction.

ca_cnosig
: Continuous No Signal: the maximum time of silence (no signal) allowed immediately after cadence detection begins. If exceeded, a no ringback is returned. Default: 4000 (in 10 msec units), or 40 seconds.

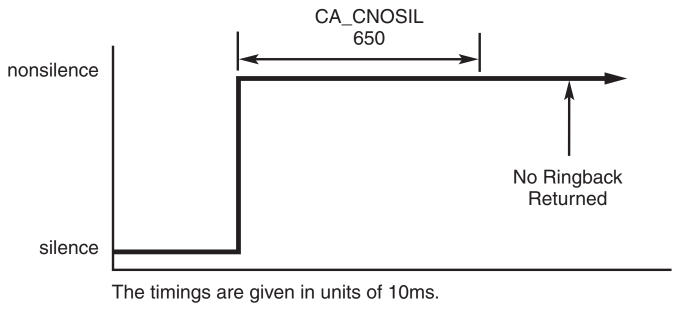
Figure 10. No Ringback Due to Continuous No Signal



If the length of any period of nonsilence exceeds the value of ca_cnosil (continuous nonsilence), a **no ringback** is returned, shown in Figure 11.

ca_cnosil
: Continuous Nonsilence: the maximum length of nonsilence allowed. If exceeded, a no ringback is returned. Default: 650 (in 10 msec units), or 6.5 seconds.

Figure 11. No Ringback Due to Continuous Nonsilence



7.16.5.3 Cadence Detection Parameters Affecting a No Answer or Busy

By using the ca_nbrdna parameter, you can set the maximum number of ring cadence repetitions that will be detected before returning a **no answer**.

By using the ca_nbrdna and ca_nsbusy parameters, you can set the maximum number of busy cadence repetitions.

ca_nbrdna
: Number of Rings Before Detecting No Answer: the number of single or double rings to wait before returning a no answer. Default: 4.

ca_nsbusy
: Nonsilence Busy: the number of nonsilence periods in addition to ca_nbrdna to wait before returning a busy. Default: 0. ca_nsbusy is added to ca_nbrdna to give the actual number of busy cadences at which to return busy. Note that even though ca_nsbusy is declared as an unsigned variable, it can be a small negative number.
Do not allow ca_nbrdna + ca_nsbusy to equal 2. This is a foible of the 2's complement bit mapping of a small negative number to an unsigned variable.

7.16.5.4 Cadence Detection Parameters Affecting a Connect

You can cause cadence detection to measure the length of the salutation when the phone is answered. The salutation is the greeting when a person answers the phone, or an announcement when an answering machine or computer answers the phone.

By examining the length of the greeting or salutation you receive when the phone is answered, you may be able to distinguish between an answer at home, at a business, or by an answering machine.

The length of the salutation is returned by the **ATDX_ANSRSIZ( )** function. By examining the value returned, you can estimate the kind of answer that was received.

Normally, a person at home will answer the phone with a brief salutation that lasts about 1 second, such as “Hello” or “Smith Residence.” A business will usually answer the phone with a longer greeting that lasts from 1.5 to 3 seconds, such as “Good afternoon, Dialogic Corporation.” An answering machine or computer will usually play an extended message that lasts more than 3 or 4 seconds.

This method is not 100% accurate, for the following reasons:

- The length of the salutation can vary greatly.
- A pause in the middle of the salutation can cause a premature connect event.
- If the phone is picked up in the middle of a ringback, the ringback tone may be considered part of the salutation, making the **ATDX_ANSRSIZ( )** return value inaccurate.

In the last case, if someone answers the phone in the middle of a ring and quickly says “Hello”, the nonsilence of the ring will be indistinguishable from the nonsilence of voice that immediately follows, and the resulting **ATDX_ANSRSIZ( )** return value may include both the partial ring and the voice. In this case, the return value may deviate from the actual salutation by 0 to +1.8 seconds. The salutation would appear to be the same as when someone answers the phone after a full ring and says two words.

Note: A return value of 180 to 480 may deviate from the actual length of the salutation by 0 to +1.8 seconds.

Cadence detection will measure the length of the salutation when the ca_hedge (hello edge) parameter is set to 2 (the default).

ca_hedge

Hello Edge: the point at which a connect will be returned to the application, either the rising edge (immediately when a connect is detected) or the falling edge (after the end of the salutation).

1 = rising edge. 2 = falling edge. Default: 2 (connect returned on falling edge of salutation). Try changing this if the called party has to say “Hello” twice to trigger the answer event.

Because a greeting might consist of several words, call progress analysis waits for a specified period of silence before assuming the salutation is finished. The ca_ansrdgl (answer deglitcher) parameter determines when the end of the salutation occurs. This parameter specifies the maximum amount of silence allowed in a salutation before it is determined to be the end of the salutation. To use ca_ansrdgl, set it to approximately 50 (in 10 msec units).

ca_ansrdgl

Answer Deglitcher: the maximum silence period (in 10 msec units) allowed between words in a salutation. This parameter should be enabled only when you are interested in measuring the length of the salutation. Default: -1 (disabled).

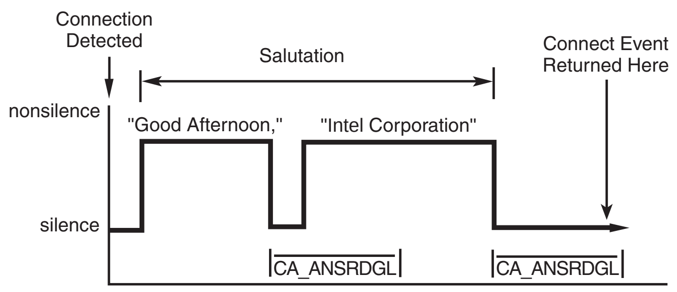
The ca_maxansr (maximum answer) parameter determines the maximum allowable answer size before returning a **connect**.

ca_maxansr

Maximum Answer: the maximum allowable length of ansrsize. When ansrsize exceeds ca_maxansr, a connect is returned to the application. Default: 1000 (in 10 msec units), or 10 seconds.

Figure 12 shows how the ca_ansrdgl parameter works.

Figure 12. Cadence Detection Salutation Processing



When ca_hedge = 2, cadence detection waits for the end of the salutation before returning a **connect**. The end of the salutation occurs when the salutation contains a period of silence that exceeds ca_ansrdgl or the total length of the salutation exceeds ca_maxansr. When the **connect** event is returned, the length of the salutation can be retrieved using the **ATDX_ANSRSIZ( )** function.

After call progress analysis is complete, call **ATDX_ANSRSIZ( )**. If the return value is less than 180 (1.8 seconds), you have probably contacted a residence. A return value of 180 to 300 is probably a business. If the return value is larger than 480, you have probably contacted an answering machine. A return value of 0 means that a **connect** was returned because excessive silence was detected. This can vary greatly in practice.

Note: When a connect is detected through positive voice detection or loop current detection, the DX_CAP parameters ca_hedge, ca_ansrdgl, and ca_maxansr are ignored.

7.16.6 Obtaining Cadence Information

Note: Cadence detection in basic call progress analysis is supported on Springware boards and is provided for backward compatibility purposes only.

To return cadence information, you can use the following extended attribute functions:

ATDX_SIZEHI( )
: duration of the cadence non-silence period (in 10 msec units)

ATDX_SHORTLOW( )
: duration of the cadence shorter silence period (in 10 msec units)

ATDX_LONGLOW( )
: duration of the cadence longer silence period (in 10 msec units)

ATDX_ANSRSIZ( )
: duration of answer if a connect occurred (in 10 msec units)

ATDX_CONNTYPE( )
: connection type. If **ATDX_CONNTYPE( )** returns CON_CAD, the connect was due to cadence detection.

Recording and Playback 8

This chapter discusses playback and recording features supported by the Dialogic® Voice API library. The following topics are discussed:

8.1 Overview of Recording and Playback

The primary voice processing operations provided by a Dialogic® 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 27.
- Controlling the speed and volume when messages are played back is documented in Chapter 9, “Speed and 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 Dialogic® 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.

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.

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.

Note: Not all voice coders are supported in all scenarios, such as for silence compressed record or for speed control. Whenever a restriction exists, it is noted.

The voice encoding methods or data formats supported on Dialogic® HMP Software are listed in Table 8.

Table 8. Voice Encoding Methods (HMP Software)

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	8	8	64	VOX, WAVE
Linear PCM	8	16	128	VOX, WAVE
Linear PCM	11	8	88	VOX, WAVE
GSM 6.10 full rate (Microsoft format)	8	(value ignored)	13	WAVE
G.726 bit exact	8	2	16	VOX, WAVE
G.726 bit exact	8	3	24	VOX, WAVE
G.726 bit exact	8	4	32	VOX, WAVE
G.726 bit exact	8	5	40	VOX, WAVE
G.729A	8	8	64	WAVE

The voice encoding methods supported on Dialogic® Springware boards are listed in Table 9.

Table 9. Voice Encoding Methods (Springware)

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	8	8	64	VOX, WAVE

Table 9. Voice Encoding Methods (Springware)

Digitizing Method	Sampling Rate (kHz)	Resolution (Bits)	Bit Rate (Kbps)	File Format
Linear PCM	11	8	88	VOX, WAVE
GSM 6.10 full rate (Microsoft format)	8	(value ignored)	13	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 Mail (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 3802 on the Internet Engineering Task Force (IETF) 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.

8.7 Transaction Record

Transaction record enables the recording of a two-party conversation by allowing two time-division multiplexing (TDM) bus time slots from a single channel to be recorded. This feature is useful for call center applications where it is necessary to archive a verbal transaction or record a live conversation. A live conversation requires two time slots on the TDM bus, but Dialogic® voice boards today can only record one time slot at a time. No loss of channel density is realized with this feature. Voice activity on two channels can be summed and stored in a single file, or in a combination of files, devices, and/or memory.

Use the following function for transaction record on Windows®:

dx_mreciottdata( )
records voice data from two channels to a data file, memory, or custom device

On Dialogic® Springware boards on Linux, use the following functions for transaction record:

dx_recm( )
records voice data from two channels to a data file, memory, or custom device

dx_recmf( )
records voice data from two channels to a single file

8.8 Silence Compressed Record

The silence compressed record (SCR) feature is discussed in more detail in the following topics:

- Overview
- Enabling
- Encoding Methods Supported

8.8.1 Overview

The silence compressed record feature (SCR) enables recording with silent pauses eliminated. This results in smaller recorded files with no loss of intelligibility.

On Dialogic® Springware boards, when the audio level is at or falls below the silence threshold for a minimum duration of time, SCR begins. When a short burst of noise (glitch) is detected, the compression does not end unless the glitch is longer than a specified period of time.

On Dialogic® HMP Software, the SCR algorithm is based on energy detection and zero crossing.

The SCR algorithm operates on one msec blocks of speech and uses a two-fold approach to determine whether a sample is speech or silence. Two probability of speech values are calculated using a zero crossing algorithm and an energy detection algorithm. These values are put together to calculate a combined probability of speech.

The energy detection algorithm allows you to modify the background noise threshold range. Signals above the high threshold are declared speech, and signals below the low threshold are declared silence.

Speech or silence is declared based on the previous sample, the current combined probability of speech in relation to the speech probability threshold and silence probability threshold parameters, and the trailing silence parameter.

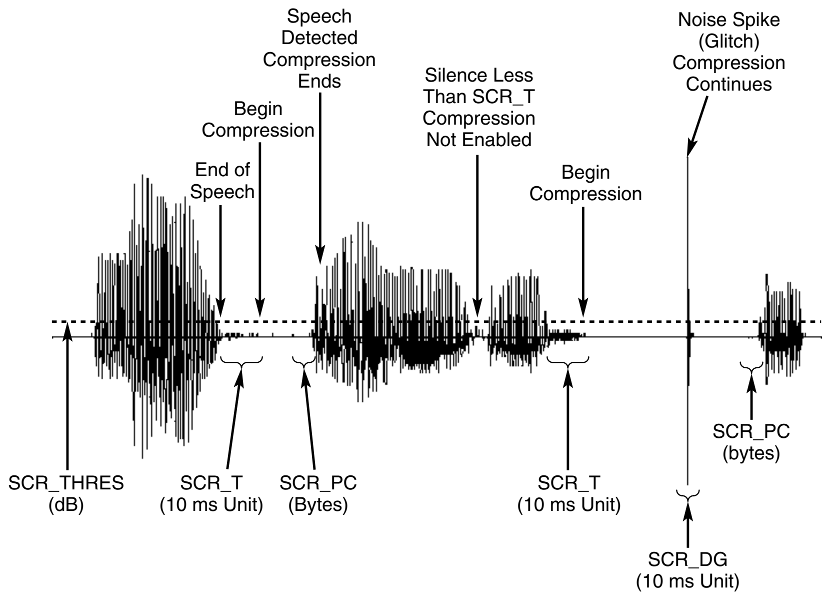
8.8.2 Enabling

On Dialogic® HMP Software, use **dx_setparm( )** and the DXCH_SCRFEATURE define to turn silence compressed record (SCR) on and off. Once enabled, voice record functions automatically record with SCR. For information on modifying SCR parameters, see the Configuration Guide.

On Dialogic® Springware boards, you enable SCR in the *voice.prm* file which is downloaded to the board during initialization. You must edit this file and set appropriate values for the SCR parameters for use in your working environment before initializing the board. You cannot enable this feature through the voice API. After SCR is enabled in the *voice.prm* file, it is automatically activated by the use of voice record functions such as **dx_rec( )**.

On Dialogic® Springware boards, the SCR parameters specify the silence threshold, the duration of silence at the end of speech before silence compression begins, the duration of a glitch in the line which does not stop silence compression, and more. Figure 13 illustrates how these parameters work. See the Springware Architecture Products Configuration Guide for details of the parameters and information on how to enable and configure this feature.

Figure 13. Silence Compressed Record Parameters Illustrated



8.8.3 Encoding Methods Supported

On Dialogic® HMP Software, the following encoding algorithms and sampling rates are supported in silence compressed record (SCR):

- OKI ADPCM, 6 kHz and 8 kHz
- linear PCM, 8 kHz and 11 kHz
- G.711 PCM, 6 kHz and 8 kHz
- G.726, 8 kHz

On Dialogic® Springware boards, the following encoding algorithms and sampling rates are supported in SCR:

- OKI ADPCM, 6 kHz and 8 kHz
- linear PCM, 8 kHz and 11 kHz
- G.711 PCM, 6 kHz and 8 kHz

8.9 Recording with the Voice Activity Detector

Recording with the voice activity detector is discussed in the following topics:

- Overview
- Enabling
- Encoding Methods Supported

8.9.1 Overview

The voice activity detector (VAD) is not supported on Dialogic® Springware boards.

The **dx_reciottdata( )** function, used to record voice data, has two modes that work with the voice activity detector. One mode enables voice activity detection with event notification upon detection. The second mode adds initial silence compression on the line before voice energy is detected; if initial silence is greater than the default allowable amount of silence, the amount in excess is eliminated. This mode uses the same algorithm as the silence compressed record (SCR) feature described in Section 8.8, "Silence Compressed Record", on page 79.

The voice activity detector is a component in the voice software that examines the incoming signal and determines if the signal contains significant energy and is likely to be voice.

8.9.2 Enabling

The modes related to the voice activity detector are specified in the **mode** parameter of the **dx_reciottdata( )** function. They are:

RM_VADNOTIFY
: Generates an event, TDX_VAD, on detection of voice energy during the recording operation.

  Note: TDX_VAD does not indicate function termination; it is an unsolicited event. Do not confuse this event with the TEC_VAD event which is used in the continuous speech processing (CSP) library.

RM_ISCR
: Adds initial silence compression to the VAD capability. Initial silence here refers to the amount of silence on the line *before* voice activity is detected. When using RM_ISCR, the default value for the amount of initial silence allowable is 3 seconds. Any initial silence longer than that will be eliminated to the default allowable amount. This default value can be changed by modifying a parameter in the .config file for the board and then generating a new .fcd file. The 0x416 parameter must be added in the [encoder] section of the .config file. For details on using this parameter, see the Configuration Guide.

  Note: The RM_ISCR mode can only be used in conjunction with RM_VADNOTIFY.

When these two modes are used together, no data is recorded as output until voice activity is detected on the line. The TDX_VAD event indicates the initiation of voice. The output file will be empty before voice activity is detected, although some initial silence may be included as specified in the .fcd file.

To enable these modes, OR them to the **mode** parameter. For example:

```
t_Return=dx_reciottdata(DevHandle, Iott, Tpt, &t_Xpb, EV_ASYNC|RM_VADNOTIFY);
```

```
t_Return=dx_reciottdata(DevHandle, Iott, Tpt, &t_Xpb, EV_ASYNC|RM_VADNOTIFY|RM_ISCR);
```

Note: The **dx_reciottdata( )** function does not perform echo-cancelled streaming. For automatic speech recognition applications, use record or streaming functions in the Dialogic® Continuous Speech Processing (CSP) API library. For more information, see the *Dialogic® Continuous Speech Processing API Programming Guide* and *Dialogic® Continuous Speech Processing API Programming Guide*.

8.9.3 Encoding Methods Supported

The following encoding algorithms and sampling rates are supported for recording with the voice activity detector:

- OKI ADPCM, 6 kHz and 8 kHz
- linear PCM, 8 kHz and 8 kHz
- G.711 PCM, 6 kHz and 8 kHz
- G.726, 8 kHz

8.10 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 Guidelines

8.10.1 Streaming to Board Overview

The streaming to board feature is not supported on Dialogic® Springware boards.

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.

8.10.2 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

8.10.3 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. Before calling the **dx_OpenStreamBuffer( )** function, you must call **dx_Open( )** on a board, channel, or physical board. Otherwise, the DM3 library will not load, and **dx_OpenStreamBuffer( )** will fail.
2. 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 8.10.4, “Streaming to Board Guidelines”, on page 84.
3. 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 8.10.4, “Streaming to Board Guidelines”, on page 84.
4. Initialize and create a circular stream buffer using **dx_OpenStreamBuffer( )**.
5. Set the high and low water marks using **dx_SetWaterMark( )**.
6. 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.
7. Put data in the circular stream buffer using **dx_PutStreamData( )**.
8. 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.
9. When all files are played, issue **dx_CloseStreamBuffer( )**.

8.10.4 Streaming to Board Guidelines

On Windows®

Consider the following usage guidelines when implementing streaming to board in your application on a Windows® system:

- 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.

- 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 100 kbytes, and the bulk queue buffer size is 8 kbytes, set the high water mark to 84 kbytes. The bulk queue buffer size is set through the **dx_setchxfercnt( )** function.
- Recommendation for the low water mark:
  - If the bulk queue buffer size is less than 8 kbytes, the low water mark should be four times the size of the bulk queue buffer size.
  - If the bulk queue buffer size is greater than 8 kbytes and less than 16 kbytes, the low water mark should be three times the size of the bulk queue buffer size.
  - If the bulk queue buffer size is greater than 16 kbytes, the low water mark should be two times the size of the bulk queue buffer size.
- 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( )**.

On Linux

Consider the following usage guidelines when implementing streaming to board in your application on a Linux system:

- 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 the driver. This buffer is set to 16 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 16 kbytes, set the high water mark to 468 kbytes (500-32=468).
- 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 16 kbytes, set the low water mark to 32 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( )**.

Speed and Volume Control 9

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

9.1 Speed and Volume Control Overview

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

Note: On Dialogic® HMP Software, before using the speed control feature, you must enable this feature in the [decoder] section of the CONFIG file. The speed control feature is disabled by default to preserve MIPS usage. For more information on enabling speed control, see the Configuration Guide.

On Dialogic® HMP Software, speed can be controlled on playbacks using the following encoding methods:

- OKI ADPCM 24 kbps and 32 kbps
- G.711 PCM A-law or mu-law encoding 48 kbps and 64 kbps
- linear PCM 128 kbps

On Dialogic® Springware boards, speed can be controlled on playbacks using 24 kbps or 32 kbps OKI ADPCM only.

If an attempt is made to adjust speed on an unsupported board, the firmware will ignore the request and play will continue at normal speed.

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 76.

9.2 Speed and Volume Convenience Functions

The convenience functions set a digit that will adjust speed or volume, but do not use any data structures. These convenience functions will only function properly if you use the default settings of the speed or volume modification tables. These functions assume that the modification tables have not been modified. The convenience functions are:

dx_addspddig( )
adds a digit that will modify speed by a specified amount

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

9.3 Speed and Volume Adjustment Functions

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

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

dx_adjsv( )
Adjusts speed or 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.

9.4 Speed and Volume Modification Tables

Each channel has a speed or volume modification table for play speed or play volume adjustments. Except for the value of the settings, the table is the same for speed and volume.

Each speed or volume modification table (SVMT) table has 21 entries, 20 that allow for a maximum of 10 increases and decreases in speed or volume. The entry in the middle of the table is referred to as the "origin" entry that represents normal speed or volume. The normal speed or volume is how playback occurs when the speed and volume control feature is not used. See Table 10, "Default Speed Modification Table", on page 90 and Table 11, "Default Volume Modification Table", on page 91.

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

Although the origin is typically set to normal speed/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 speed/volume.

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

A speed/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 speed/volume to any setting in the SVMT.

To change the default values of the speed or 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.

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

- a specified level (that is, to a specified absolute position in the speed table or volume table)
- a change in level (that is, by a specified number of steps up or down in the speed table or volume 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 speed modification table is shown in Table 10.

Table 10. Default Speed Modification Table

Table Entry	Default Value (%)	Absolute Position
decrease[0]	-128 (80h)	-10
decrease[1]	-128 (80h)	-9
decrease[2]	-128 (80h)	-8
decrease[3]	-128 (80h)	-7
decrease[4]	-128 (80h)	-6
decrease[5]	-50	-5
decrease[6]	-40	-4
decrease[7]	-30	-3
decrease[8]	-20	-2
decrease[9]	-10	-1
origin	0	0
increase[0]	+10	1
increase[1]	+20	2
increase[2]	+30	3
increase[3]	+40	4
increase[4]	+50	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 speed modification table:

- Each entry in the table is a percentage deviation from the default play speed (“origin”). For example, the decrease[6] position reduces speed by 40%. This is four steps from the origin.
- The total speed modification range is from -50% to +50%. In this table, the lowest position used is the decrease[5] position. The remaining decrease fields are set to -128 (80h). If these “nonadjustment” positions are selected, the default action is to play at the decrease[5] speed.
- These fields can be reset, as long as no values lower than -50 are used. For example, you could spread the 50% speed decrease over 10 steps rather than 5. Similarly, you could spread the 50% speed increase over 10 steps rather than 5.
- The default entries for index values -10 to -6 and +6 to +10 are -128 which represent a null-entry. To customize the table entries, you must use the **dx_setsvmt( )** function.
- On Dialogic® HMP Software, when adjustment is associated with a DTMF digit, speed can be increased or decreased in increments of 1 (10%) only. To achieve an increase in speed of 30% for example, the user would press the DTMF digit three times.

The default volume modification table is shown in Table 11.

Table 11. 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.
- On Dialogic® HMP Software, 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.
- On Dialogic® HMP Software, 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.
- On Dialogic® Springware boards, digits that are used for play adjustment will not be used as a terminating condition. If a digit is defined as both, then the play adjustment will take priority.
- 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 speed or volume and then be removed from the queue.

9.6 Setting Play Adjustment Conditions

Adjustment conditions are set in the same way for speed or volume. 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.

9.7 Explicitly Adjusting Speed and Volume

Speed and volume adjustments are made in the same way. The following steps describe how to adjust speed, but you can use exactly the same procedure for volume:

1. Set up the speed 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 speed 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 speed modification table by specifying the size and type of the adjustment.

Send and Receive FSK Data 10

This chapter describes the Analog Display Services Interface (ADSI) protocol, two-way frequency shift keying (FSK), and guidelines for implementing ADSI and two-way FSK support using voice library functions.

10.1 Overview of ADSI and Two-Way FSK Support

The features and functionality described in this chapter apply to Dialogic® Springware boards only.

The Analog Display Services Interface (ADSI) is a Telcordia Technologies (formerly Bellcore) standard that defines a protocol used to transmit data to a display-based, ADSI-compliant telephone. ADSI enables data to be sent across an analog telephone line, providing asynchronous data communications with 8 data bits, 1 start and 1 stop bit, and no parity.

For many years, one-way ADSI support was provided through the **dx_play( )** and **dx_playf( )** functions. This ADSI support enabled developers to use Dialogic® telecom boards to make ADSI servers that work with ADSI phones and to support ADSI features such as visual voice mail. This is referred to as the "older" implementation of one-way ADSI.

Dialogic has expanded the capabilities of basic ADSI with the introduction of two-way frequency shift keying (FSK) capabilities. Two-way FSK is a convenient and robust mechanism to exchange small amounts of data between the telephone and the server using FSK as the transport layer. The two-way FSK functionality allows products to transmit and receive half-duplex FSK Bell 202 1200 bps data over the Public Switched Telephone Network (PSTN).

One of the applications of two-way FSK is fixed-line short message service, also called small message service, or SMS. (This service is also known as text messaging.) This service allows the server and display-based telephone to exchange short text messages via the PSTN.

As with basic ADSI, the transmission and reception of two-way FSK data is initiated after a call between the server and the display-based telephone (or CPE) has been established, by one of the devices sending a special alerting signal (typically a CAS tone). The other device will then acknowledge and the data transmission (and/or reception) will then be initiated. Once the data transmission/reception is complete, the devices switch back to voice mode.

This newer implementation of ADSI is supported through the **dx_RxIottData( )**, **dx_TxIottData( )**, and **dx_TxRxIottData( )** functions. This implementation is referred to simply as "ADSI Support" or "Two-Way ADSI." This newer ADSI support provides for both one-way and two-way ADSI transmission and is the recommended method for implementing either one-way or two-way ADSI in an application program. The older one-way ADSI support can be used but is not recommended. Future enhancements to ADSI feature and functionality will not be made to the **dx_play( )** and **dx_playf( )** functions. See Section 10.9, "Modifying Older One-Way ADSI Applications", on page 103 for information on converting from the older to the newer method for using ADSI.

10.2 ADSI Protocol

ADSI is a superset of the caller ID and call waiting functions. ADSI is built on the same protocol as caller ID and shares the same ADSI Data Message Format (ADMF). The ADSI protocol requires a Bell 202/V.23 1200 bps FSK-based modem for data transmission.

The ADSI protocol supports a variable display size on a display-based telephone. An ADSI telephone can work in either voice mode or data mode. Voice mode is for normal telephone audio communication, and data mode is for transmitting ADSI commands and controlling the telephone display (voice is muted in data mode). An ADSI alert tone is used to verify that the hardware is connected to an ADSI telephone and to alert the telephone that ADSI data will be transferred.

The ADSI protocol consists of three defined layers, as follows:

message assembly layer
: assembles the body of the ADMF message

data link layer
: generates the checksum, which is used for error detection, and sends it to the driver

physical layer
: transports the composite message via the modem to the CPE on a transparent (bit-for-bit) basis

Dialogic provides only the physical layer and a portion of the data link layer of the ADSI protocol. The user is responsible for creating the ADSI messages and the corresponding checksums.

The ADSI data must conform to interface requirements described in Telcordia Technologies Generic Requirements GR-30-CORE, *Voiceband Data Transmission Interface Generic Requirements*. For information about message requirements (how the data should be displayed on the CPE), see Generic Requirements GR-1273, *Generic Requirements for and SPCS to Customer Premises Equipment Data Interface for Analog Display Services*. To obtain a copy of these technical references, visit http://www.telcordia.com.

10.3 ADSI Operation

ADSI data is encoded using a standard 1200 baud modem specification and transmitted to the telephone on the voice channel. The voice is muted for the data transfer to occur. Responses from the ADSI telephone are mapped into DTMF sequences.

ADSI data is sent to the ADSI telephone in a message burst corresponding to a single transmission. Each message burst or transmission can contain up to 5 messages, with each message consisting of one or more ADSI commands.

The ADSI alert tone causes the ADSI telephone to switch to data mode for 1 message burst or transmission. When the transmission is complete, the ADSI phone will revert to voice mode unless the transmission contained a message with the “Switch to Data” command.

After the data is transmitted, the ADSI telephone sends an acknowledgment consisting of a DTMF “d” plus a digit from 1 to 5 indicating the number of messages in the transmission that the ADSI telephone received and understood. By obtaining this message count and comparing it with the number of messages transmitted, you can check for errors and retransmit any messages not received. (If you send 4 messages and the telephone receives 2, you must resend messages 3 and 4.)

You can send more than one transmission during a call. After the initial transmission of a call, you do not have to re-establish the handshaking (sending the alert tone or receiving the acknowledgment digit) as long as you have left the ADSI telephone in data mode using the ADSI “Switch to Data” command. This is useful for performing additional data transmissions during the same call without needing to send the alert tone or receive the acknowledgment digit for each transmission.

10.4 One-Way ADSI

One-way ADSI support enables Dialogic® telecom boards to be used as ADSI servers and to support ADSI features such as visual voice mail. One-way ADSI allows for the one-way transmission of data from a server to a customer premises equipment (CPE) device, such as a display-based telephone. The phone (CPE) sends dual tone multi-frequency (DTMF) messages to the server, indicating whether the data was received successfully.

For a more detailed description of the one-way ADSI data transfer process, see Section 10.8, “Developing ADSI Applications”, on page 98.

ADSI data can be transferred only to display-based telephones that are ADSI compliant. Check with your telephone manufacturer to find out if your telephone is a true ADSI-compliant device. An ADSI alert tone, referred to as a CAS (CPE Alerting Signal), is sent by the server to query a CPE device, such as an ADSI display phone. The device responds appropriately and, if the device is ADSI-compliant, the ADSI data transfer is initiated.

Note: ADSI-compliant phones are also referred to as "Type 3 CPE Devices" by Telcordia Technologies and by the Electronic Industry Association/Telecommunications Industry Association (EIA/TIA).

10.5 Two-Way ADSI

Two-way ADSI includes several enhancements to one-way ADSI, including two-way frequency shift keying (FSK). The following topics discuss two-way ADSI:

- Transmit to On-Hook CPE
- Two-Way FSK

10.5.1 Transmit to On-Hook CPE

The transmit to on-hook customer premises equipment (CPE) feature allows messages to be sent to an ADSI phone when the phone is either on-hook or off-hook.

This feature supports the transmission of FSK data burst messages to CPE devices that are kept in the on-hook state by either the Central Office (CO) or the PBX/KTS. This allows an ADSI/Caller ID phone to receive and potentially display messages while it is in the on-hook state. For example, ADSI phones can be configured, accessed, and downloaded with features, outside of regular business hours while the phone is on-hook, without ringing and without subscriber intervention.

Note: The transmit to on-hook CPE feature works only if the CO supports this feature.

10.5.2 Two-Way FSK

The two-way frequency shift keying (FSK) feature allows users to send and receive character or binary data at 1200 bits/second between the server and compatible devices, such as certain ADSI phones with keyboards. The two-way FSK feature supports applications such as off-line e-mail editing and sending FSK Caller ID data to a customer premises equipment (CPE) device.

FSK (frequency shift keying) is a modulation technique used to transfer data over voice lines. The basic ADSI capability supports only FSK Transmit (one-way FSK), in which an FSK message is sent from the server to an ADSI display phone, with the phone in the off-hook state. The phone (CPE) sends dual tone multi-frequency (DTMF) messages to the server. As DTMF messages are sent to the server, the effective data rate is very slow, approximately 6 characters per second maximum. This speed is satisfactory for ACK/NAK signaling but it is not usable for any bulk data transport in the inbound direction from the CPE.

FSK data reception uses a DSP-based Bell 202/V.23 low speed (1200 baud) modem receiver. A 1200 baud modem does not need to train for data transmission, and therefore is faster than a high-speed modem for short data bursts.

Two-way FSK for ADSI supports the transmission and the reception of FSK data between the server and the CPE. The server initiates the reception of data from the CPE by sending a CAS to tell the CPE to switch to data mode, followed by a message that tells the CPE to switch to peripheral mode. Once it is in peripheral mode, the CPE can send FSK messages to the server using the ADSI Data Message Format (ADMF), instead of the slower DTMF-based scheme.

See Section 10.8, “Developing ADSI Applications”, on page 98 for a more detailed description of how to use library functions to develop two-way ADSI data transfer applications. For more information about two-way FSK transmission, see Telcordia Technologies Special Report SR-3462, *A Two-Way Frequency Shift Keying Communication for the ADSI*.

In addition to features provided by basic ADSI, two-way FSK for ADSI can be used in the following applications:

- sending and receiving e-mail between display-based ADSI phones and the server
- sending FSK caller ID data to a CPE device

10.6 Fixed-Line Short Message Service (SMS)

Fixed-line short message service or SMS is a service that allows text messages to be sent and received in the PSTN network. SMS is also known as small message service or text messaging.

The voice library supports the creation of fixed-line SMS applications through the **dx_RxIottData( )**, **dx_TxIottData( )**, and **dx_TxRxIottData( )** functions.

Fixed-line SMS solutions can be created using the standard Telcordia Technologies (formerly Bellcore) ADSI specification *or* using the ETSI-FSK specification ETSI ES 201 912.

The ETSI-FSK specification differs from the ADSI FSK specification in these ways:

- It uses a different physical layer. Settings for channel seizure and mark length differ. For more information on FSK transmission requirements, see ITU-T EN 300 659-2 specification.
- It uses different handshaking and timing specifications.

To set the voice channel to ETSI compatibility, specify the two-way FSK transmit framing parameters in the *voice.prm* file. For more information on these parameters, see the Configuration Guide for Springware boards.

10.7 ADSI and Two-Way FSK Voice Library Support

The following voice library functions and data structures support this functionality:

dx_RxIottData( ) function
: Receives ADSI data on a specified channel.

dx_TxIottData( ) function
: Transmits ADSI data on a specified channel.

dx_TxRxIottData( ) function
: Starts a transmit-initiated reception of data (two-way ADSI) on a specified channel.

ADSI_XFERSTRUC data structure
: Stores information for the transmission and reception of ADSI data. It is used by the **dx_RxIottData( )**, **dx_TxIottData( )**, and **dx_TxRxIottData( )** functions.

DV_TPT data structure
: Specifies a termination condition for an I/O function; in this case, **dx_RxIottData( )**, **dx_TxIottData( )**, or **dx_TxRxIottData( )**.

ATDX_TERMMSK( ) function
Returns the reason for the last I/O function termination.

To determine whether your board supports FSK, use **dx_getfeaturelist( )** to return information about the features supported in the FEATURE_TABLE structure; the ft_play field, FT_ADSI bit, is used to indicate FSK support.

Two-way FSK transmit framing parameters for ETSI compatibility are set in the *voice.prm* file. For more information on these parameters, see the Configuration Guide for Springware boards.

10.8 Developing ADSI Applications

This section provides the following information on developing applications for one-way and two-way ADSI FSK:

- Technical Overview of One-Way ADSI Data Transfer
- Implementing One-Way ADSI Using dx_TxIottData( )
- Technical Overview of Two-Way ADSI Data Transfer
- Implementing Two-Way ADSI Using dx_TxIottData( )
- Implementing Two-Way ADSI Using dx_TxRxIottData( )

10.8.1 Technical Overview of One-Way ADSI Data Transfer

In one-way ADSI data transfer, the ADSI server sends ADSI messages to a CPE device, such as an ADSI-compliant telephone. The transactions that occur between the server and the CPE during one-way ADSI data transfer are as follows:

1. The server initiates the data transfer by sending a CPE Alerting Signal (CAS) to the CPE.
2. When the CPE receives the CAS, the device generates an ACK (DTMF ‘A’ signal) to the server. At this point the CPE device has switched from voice mode to data mode. (If the CPE device remains in data mode, subsequent transmissions do not require the CAS.)

   Note: Only ADSI-compliant CPE devices will respond to the CAS sent by the server. Check with your manufacturer to verify that your CPE device is a true ADSI-compliant device. ADSI-compliant devices are also referred to as "Type 3 CPE Devices" by Telcordia Technologies and the EIA/TIA.
3. Upon receipt of the ACK signal, the server initiates the FSK transmission sequence. Each FSK transmission sequence can contain anywhere from 1 to 5 messages.
4. The CPE receives the FSK data and uses the checksum included within the sequence to determine the number of messages successfully received.
5. The CPE device then responds to the server with an acknowledgment digit (DTMF ‘D’) followed by a DTMF of ‘0’ through ‘5,’ which indicates the number of messages successfully received.
6. The server interprets the DTMF as follows:
   - ACK = ‘D’ followed by a DTMF in the range of 1 – 5
   - NAK = ‘D’ followed by a DTMF ‘0’

10.8.2 Implementing One-Way ADSI Using dx_TxIottData( )

The **dx_TxIottData( )** function is used to send the CAS to the CPE and implement one-way ADSI data transfer. To transfer ADSI FSK data, configure the function parameters and structures as follows:

- Set the **wType** parameter DT_ADSI.
- Configure the DX_IOTT structure with the appropriate ADSI FSK data file(s). The application is responsible for constructing the messages and checksums for each transmission.
- Set the termination conditions with the DV_TPT structure.
- Set **dwTxDataMode** within the ADSI_XFERSTRUC referenced by **lpParams** to ADSI_ALERT to generate the CAS.

The following scenario illustrates the function calls that are required to generate an initial CAS to the CPE and begin one-way ADSI data transfer.

1. Prior to executing **dx_TxIottData( )**, clear the digit buffer for the desired voice channel using **dx_clrdigbuf( )**.
2. Issue **dx_TxIottData( )**. To generate an initial CAS to the CPE device, dwTxDataMode within ADSI_XFERSTRUC must be set to ADSI_ALERT.
3. The CAS is received by the CPE and the CPE sends an acknowledgment digit (DTMF 'A') to the voice device.

   Note: If the DTMF acknowledgment digit is not received from the CPE device within 500 ms following the end of the CAS, the function will return a 0 but the termination mask returned by **ATDX_TERMMSK( )** will be TM_MAXTIME to indicate an ADSI protocol error. (The function will return a -1 if a failure is due to a general transmission error.)
4. Upon receipt of the DTMF 'A' ACK, the voice device automatically transmits the data file referenced in the DX_IOTT structure.
5. After receiving the data file(s), the CPE responds with a DTMF ACK or NAK, indicating the number of messages successfully received. (The application is responsible for determining whether the message count acknowledgment matches the number of messages that were transmitted and for re-transmitting any messages.)

   Note: Upon successful completion, the function terminates with a TM_EOD (end of data) termination mask returned by **ATDX_TERMMSK( )**.
6. After completion of **dx_TxIottData( )**, the **dx_getdig( )** function retrieves the DTMF ACK sequence from the CPE device. Set the DV_TPT tp_termno field to DX_DIGTYPE to receive the DTMF string "adx," where "x" is the message count acknowledgment digit (1-5).

After the CAS is sent to the CPE, as described in the preceding scenario, the CPE is in data mode. Provided that the ADSI messages sent to the CPE instruct the CPE to remain in data mode, subsequent ADSI transmissions to the CPE do not require the CAS. To send ADSI data without the CAS, set the dwTxDataMode within the ADSI_XFERSTRUC referenced by lpParams to ADSI_NOALERT. All other settings are the same as above.

The following scenario illustrates the function calls that are required to transfer ADSI data when the CPE is already in data mode (without sending a CAS).

1. Prior to executing **dx_TxIottData( )**, issue **dx_clrdigbuf( )** to ensure that the voice channel digit buffer is empty.
2. Issue **dx_TxIottData( )** and set dwTxDataMode within the ADSI_XFERSTRUC data structure to ADSI_NOALERT. This initiates the immediate transfer of the data file(s) referenced in the DX_IOTT structure to the CPE device.
3. After receiving the data file(s), the CPE responds with a DTMF ACK or NAK, indicating the number of messages successfully received. (The application is responsible for determining whether the message count acknowledgment matches the number of messages that were transmitted and for re-transmitting any messages.)
4. After completion of **dx_TxIottData( )**, the **dx_getdig( )** function retrieves the DTMF ACK sequence from the CPE device. Set the DV_TPT tp_termno field to DX_DIGTYPE to receive the DTMF string "adx," where "x" is the message count acknowledgment digit (1-5).

10.8.3 Technical Overview of Two-Way ADSI Data Transfer

In two-way ADSI data transfer, both the ADSI server and CPE device can transmit and receive ADSI data messages. The CAS is used to initiate the transfer of ADSI FSK data and to return the CPE to voice mode after the data exchange is completed.

The transactions that occur between the server and the CPE in two-way ADSI data transfer are as follows:

1. The server initiates the data transfer by sending a CPE Alerting Signal (CAS) to the CPE equipment.
2. Upon receipt of the CAS, the CPE device generates an ACK (DTMF ‘A’ signal) to the server. At this point the CPE device has switched from voice mode to data mode. (Once the CPE device is in data mode, subsequent FSK data transmissions do not require the CAS.)

   Note: Only ADSI-compliant CPE devices will respond to the CAS sent by the server. Check with your manufacturer to verify that your CPE device is a true ADSI-compliant device. ADSI-compliant devices are also referred to as "Type 3 CPE Devices" by Telcordia Technologies.
3. When the ACK signal is received, the server initiates the FSK transmission sequence. Each FSK transmission sequence can contain anywhere from 1 to 5 messages. A "Switch to Peripheral Mode" message (using 0x0A as a ‘requested peripheral’ code) must be included within the FSK transmission sequence.
4. The CPE receives the FSK data and uses the checksum included within the sequence to determine the number of messages successfully received.
5. The CPE device then responds to the server with a DTMF ‘D’ followed by a DTMF ‘0’ through ‘5’ to indicate the number of messages successfully received. In addition, the CPE device acknowledges the "Switch to Peripheral Mode" message by responding with either
   - DTMF ‘B,’ indicating that the requested peripheral is available and on line
   - DTMF ‘A,’ indicating that the requested peripheral is not available
6. The server interprets the DTMF signals as follows:
   - ‘D’ followed by a DTMF in the range of 1 – 5 = ACK

- ‘D’ followed by a DTMF ‘0’ = NAK
- DTMF ‘B’ = requested peripheral available (ready to receive and transmit ADSI data)
- DTMF ‘A’ = requested peripheral unavailable (unable to transmit or receive ADSI data)

Once the CPE device has acknowledged the "Switch to Peripheral Mode" message, the CPE may transmit data to the server at any time. The server must be prepared to receive data at any time until the CPE peripheral is switched back to voice mode. To return the CPE peripheral to voice mode, the server sends a CAS to the CPE. Upon receipt of the CAS, the CPE responds with a DTMF ‘A’ signal. Receipt of DTMF ‘A’ at the server completes the return to voice mode transition.

10.8.4 Implementing Two-Way ADSI Using dx_TxIottData( )

The **dx_TxIottData( )** function is used to implement two-way ADSI data transfer. The **dx_TxIottData**( ) function transmits the CAS and the subsequent "Switch to Peripheral Mode Message" to the CPE. To transfer ADSI FSK data, set the parameters and configure the structures as described below:

- Set the **wType** parameter DT_ADSI.
- Configure the DX_IOTT structure with the appropriate ADSI FSK data file(s), including the "Switch to Peripheral Mode" message. The application is responsible for constructing the messages and checksums for each transmission.
- Set the termination conditions with the DV_TPT structure.
- Set dwTxDataMode within the ADSI_XFERSTRUC referenced by **lpParams** to ADSI_ALERT to generate the CAS.

The following scenario illustrates the function calls that are required to generate an initial CAS to the CPE and begin two-way ADSI data transfer.

1. Prior to executing **dx_TxIottData( )**, clear the digit buffer for the desired voice channel using **dx_clrdigbuf**( ).
2. Issue **dx_TxIottData**( ). To generate an initial CAS to the CPE device, set dwTxDataMode within ADSI_XFERSTRUC data structure to ADSI_ALERT.
3. The CAS is received by the CPE and the CPE sends an acknowledgment digit (DTMF ‘A’) to the voice device.

   Note: If the DTMF acknowledgment digit is not received from the CPE device within 500 ms following the end of the CAS, the function will return a 0 but the termination mask returned by **ATDX_TERMMSK( )** will be TM_MAXTIME to indicate an ADSI protocol error. (The function will return a -1 if a failure is due to a general transmission error.)
4. Upon receipt of the DTMF ‘A’ ACK, the voice device automatically transmits the data file referenced in the DX_IOTT structure, which must include the "Switch to Peripheral Mode" message.
5. After receiving the data file(s), the CPE responds with a DTMF ACK or NAK, indicating the number of messages successfully received. (The application is responsible for determining whether the message count acknowledgment matches the number of messages that were transmitted and for re-transmitting any messages.)

Note: Upon successful completion, the function terminates with a TM_EOD (end of data) termination mask returned by **ATDX_TERMMSK( )**.

6. The CPE responds to the "Switch to Peripheral Mode" message with either DTMF ‘B’ if the peripheral is available or DTMF ‘A’ if the peripheral is unavailable.
7. After completion of **dx_TxIottData( )**, the **dx_getdig( )** function retrieves the DTMF ACK sequence from the CPE device. Set the DV_TPT tp_termno parameter to DX_DIGTYPE to receive the DTMF string "adxb," where "x" is the message count acknowledgment digit (1-5). When the DTMF string is received, additional messages can be sent and received between the server and the CPE peripheral.

10.8.5 Implementing Two-Way ADSI Using dx_TxRxIottData( )

After the two-way ADSI transmission is implemented using the **dx_TxIottData( )** function, additional ADSI FSK messages are typically sent to the CPE peripheral to configure the display and soft keys. Since at this point the CPE peripheral has been configured to send data to the server, the **dx_TxRxIottData( )** function should be used to send the data to the CPE and then quickly turn around and be ready to receive data from the CPE.

To transfer ADSI FSK data using **dx_TxRxIottData( )**, set the function parameters and configure the structures as described below:

- Set **wTyp**e to DT_ADSI.
- Configure DX_IOTT structures referenced by **lpTxIott** and **lpRxIott** with the appropriate ADSI FSK data files. The application is responsible for constructing the messages and checksums for each transmission.
- Set the termination conditions for the transmit and receive portions of the function with the DV_TPT structures referenced by **lpTxTerminations** and **lpRxTerminations**, respectively.
- Set dwTxDataMode and dwRxDataMode within the ADSI_XFERSTRUC referenced by **lpParams** to ADSI_NOALERT to transmit and receive FSK ADSI data without generation of a CAS.

The following scenario illustrates the function calls that are required to send and receive FSK ADSI data between the server and the CPE.

1. Prior to executing **dx_TxIottData( )**, clear the digit buffer for the desired voice channel using **dx_clrdigbuf( )**.
2. Issue **dx_TxRxIottData( )** with dwTxDataMode and dwRxDataMode within ADSI_XFERSTRUC set to ADSI_NOALERT. This initiates the transmission of the data file referenced in the DX_IOTT structure to the CPE. The server voice channel is placed automatically in FSK ADSI data receive mode to receive data from the CPE.
3. After receiving the data file(s), the CPE responds with a DTMF ACK or NAK, indicating the number of messages successfully received. (The application is responsible for determining whether the message count acknowledgment matches the number of messages that were transmitted and for re-transmitting any messages.)
4. The server voice channel is ready and waiting for data from the CPE.

5. The CPE sends FSK ADSI data to the server. When an ADSI FSK message is successfully received or when the termination conditions set in lpRxTerminations are met, the **dx_TxRxIottData( )** function completes.
6. After completion of **dx_TxRxIottData( )**, the **dx_getdig( )** function retrieves the DTMF ACK sequence for the transmit portion of the function. When the DTMF string is received, additional messages can be sent and received between the server and the CPE peripheral.
7. In another thread of execution at the server, the received message(s) are processed by the application to determine the number of messages received and the integrity of the information.
8. Issue **dx_RxIottData( )** to receive messages from the CPE. This function should be issued as soon as possible because the CPE peripheral can send data to the server after a minimum of 100 msec following the completion of its transmission.

   If data needs to be transmitted to the CPE when the server is waiting to receive data, issue **dx_stopch( )** to terminate the current **dx_RxIottData( )** function. Upon confirmation of termination of **dx_RxIottData( )**, issue **dx_clrdigbuf( )** to clear the voice device channel buffer, and then issue **dx_TxIottData( )** to send the data to the CPE.

10.9 Modifying Older One-Way ADSI Applications

Prior to the release of the two-way ADSI, including two-way FSK, applications used the **dx_play( )** function to implement one-way ADSI applications. With two-way ADSI, transmit and receive data functions are introduced for data transfer. To take advantage of on-hook ADSI transfer in a one-way ADSI application, and/or to introduce two-way FSK concepts into applications, older applications need to be modified.

Applications developed prior to the release of the two-way ADSI use the following sequence of commands to generate a CAS tone followed by transmission of an ADSI file:

```
/* Setup DX_IOTT to play from disk */
/* Setup DV_TPT for termination conditions */

/* Initiate ADSI play when DTMF 'A' is received from CPE */
parmval = DM_A;
if (dx_setparm(Voice_Device, DXCH_DTINITSET, (void *)&parmval) == -1) {
   /* Process error */
}

/* Clear digit buffer for impending ADSI protocol DTMFs */
if (dx_clrdigbuf(Voice_Device) == -1) {
   /* Process error */
}

  /* Send CAS followed by ADSI data when DTMF 'A' is received */
if (dx_play(Voice_Device, &Iott_struct, &Tpt_struct, EV_SYNC | PM_ADSIALERT) == -1) {
   /* Process error */
}
```

In older applications, the use of **dx_play( )** for ADSI transmission can be replaced with the specialized **dx_TxIottData( )** data transfer function. The same DV_TPT and DX_IOTT are used by **dx_TxIottData( )** as for **dx_play( )**, however, the following additional parameters need to be configured:

wType
: specifies the data type transferred. To transfer ADSI FSK data, **wType** is set to DT_ADSI

lpParams
: specifies the data type specific information. To transmit CAS followed by the ADSI FSK message, dwTxDataMode within the ADSI_XFERSTRUC data structure pointed to by **lpParams** is set to ADSI_ALERT.

Using these parameters, the CAS will be transmitted and, upon receipt of DTMF 'A,' the ADSI FSK data will be sent to the CPE device.

The following sample code illustrates the use of the **dx_TxIottData( )** function to generate a CAS tone and transmit an ADSI file:

```
/* Setup DX_IOTT to play from disk */
/* Setup DV_TPT for termination conditions */

/* Setup ADSI_XFERSTRUC to send CAS followed by ADSI FSK upon receipt of DTMF 'A' */
adsimode.cbSize = sizeof(adsimode);
adsimode.dwTxDataMode = ADSI_ALERT;

/* Clear digit buffer for impending ADSI protocol DTMFs */
if (dx_clrdigbuf(Voice_Device) == -1) {
   /* Process error */
}

/* Send CAS followed by ADSI data when DTMF 'A' is received */
if (dx_TxIottData(Voice_Device, &IOTT, &TPT, DT_ADSI, &adsimode, EV_SYNC) == -1) {
   /* Process error */
}
```

Caller ID 11

This chapter provides information on caller ID. The following topics are covered:

11.1 Overview of Caller ID

The caller ID feature described in this chapter applies to Dialogic® Springware boards only.

Caller Identification (caller ID) is a service provided by local telephone companies to enable the subscriber to receive information about the caller. Caller ID information can include the calling party's directory number (DN), the date and time of the call, and the calling party's subscriber name. An application can enable the caller ID feature on specific channels to process caller ID information as it is received with an incoming call. The caller ID information is transmitted using FSK (frequency shift keying) to the subscriber from the service provider (telephone company Central Office) at 1200 baud.

If caller ID is enabled, on-hook detection (DTMF, MF, and global tone detection) will not function.

Caller ID is also available via the Global Call API. For more information, see the *Global Call Analog Technology User's Guide*.

11.2 Caller ID Formats

The following caller ID formats are supported:

CLASS (Custom Local Area Signaling Services)
: a set of standards published by Bellcore (now known as Telcordia Technologies) and supported on boards with loop-start capabilities in the following formats:

- Single Data Message (SDM) format
- Mltiple Data Message (MDM) format

ACLIP (Analog Calling Line Identity Presentation)
: a standard used in Singapore published by the Telecommunications Authority of Singapore and supported in the following formats:

- Single Data Message (SDM) format

- Multiple Data Message (MDM) format

CLIP (Calling Line Identity Presentation)
: a standard used in the United Kingdom published by British Telecommunications (BT)

JCLIP (Japanese Calling Line Identity Presentation)
: a standard for “Number Display” used in Japan published by Nippon Telegraph and Telephone Corporation (NTT).

Note: JCLIP operation requires that the Japanese country-specific parameter file be installed and configured (select Japan in the Dialogic country configuration).

Caller ID information is received from the Central Office (CO) between the first and second ring for CLASS and ACLIP, and before the first ring for CLIP. This information is supported as sent by the service provider in the format types described in Table 12.

Table 12. Supported CLASS Caller ID Information

Caller ID Information	CLASS and ACLIP		CLIP	JCLIP
	SDM *	MDM **		MDM **
Frame header (indicating SDM or MDM format type)	X	X		X
Calling line’s Directory Number (DN)	X	X	X	X
Date	X	X	X	
Time	X	X	X	
Calling line’s subscriber name		X	X	
Calling line’s DN (digits only)		X	X	
Dialed directory number (digits only)		X	X	X
Reason why caller DN is not available		X	X	X
Reason why calling subscriber name is not available		X	X	X
Indicate if the call is forwarded		X		
Indicate if the call is “long distance”		X		
Type of call (such as voice, ringback when free, message waiting call)			X	
Network Message System status (number of messages waiting)			X	
* Single Data Message ** Multiple Data Message				

Note: One or more of the caller ID features listed above may not be available from your service provider. Contact your service provider to determine the caller ID options available from your CO.

11.3 Accessing Caller ID Information

You can process caller ID information in your application in the following ways:

- For CLASS or ACLIP, the caller ID information is received from the service provider between the first and second ring. Set the ring event in the application to occur on or after the second ring. The ring event indicates reception of the CLASS or ACLIP caller ID information from the CO.
- For CLIP or JCLIP, the caller ID information is received from the service provider before the first ring. Set the ring event in the application to occur on or after the first ring. The ring event indicates reception of the CLIP caller ID information from the CO.

The caller ID information is available for the call from the moment the ring event is generated (if the ring event is set in your application as stated above) until one of the following occurs:

- If the call is answered (application channel goes off-hook), the caller ID information is available until the call is disconnected (application channel goes on-hook).
- If the call is unanswered (application channel remains on-hook), caller ID information is available until rings are no longer received from the CO (signaled by ring event, if enabled).

Notes: *1.* If the call is answered **before** the caller ID information has been received from the CO, caller ID information will not be available to the application.

2. If the application remains on-hook and the ring event is received **before** the caller ID information has been received from the CO, caller ID information will not be available until the beginning of the second ring.

The following voice API functions are used to access caller ID information received from the CO. These functions are not supported on Dialogic® HMP Software:

dx_gtcallid( )
Returns the calling line Directory Number (DN). Issue this function for applications that require only the calling line DN.

dx_gtextcallid( )
Returns the requested caller ID message. Issue this function for each type of caller ID message required.

dx_wtcallid( )
Waits for a specified number of rings and returns the calling station's DN. This convenience function combines the functionality of the **dx_setevtmsk( )**, **dx_getevt( )**, and **dx_gtcallid( )** functions.

Contact your service provider to determine the caller ID options available from your CO. Based on the options provided, you can determine which caller ID function best meets the application's needs.

To determine if caller ID information has been received from the CO, before issuing a **dx_gtcallid( )** or **dx_gtextcallid( )**, check the event data in the DX_EBLK event block structure. When the ring event is received (set by the application as stated above), the event data field in the event block is bit mapped and indicates that caller ID information is available when bit 0 (LSB) is set to 1 (see the function code examples in the *Voice API Library Reference*).

11.4 Enabling Channels to Use the Caller ID Feature

During Dialogic System Service startup, before the initial use of caller ID functions, the application must enable the caller ID feature on the channels requiring caller ID.

On Springware boards, caller ID is enabled by setting the DXCH_CALLID channel-based parameter to DX_CALLIDENABLE using **dx_setparm( )**. The default setting is caller ID disabled, DX_CALLIDDISABLE.

11.5 Error Handling

When the caller ID function completes, check the return code:

- If the caller ID function completes successfully, the buffer will contain the caller ID information.
- If the caller ID function fails, an error code will be returned that indicates the reason for the error. The call is still active when the error is returned.

When using the **dx_gtextcallid( )** function, error codes depend upon the Message Type ID argument (**infotype**) passed to the function. All Message Types can produce an EDX_CLIDINFO error. Message Type CLIDINFO_CALLID can also produce EDX_CLIDOOA and EDX_CLIDBLK errors.

When using the **dx_gtcallid( )** caller ID function, if an error is returned indicating that the caller's phone number (DN) is blocked or out of area, other information (for example, date or time) may be available by issuing the **dx_gtextcallid( )** caller ID function. The information that is available, other than the caller's phone number, is determined by the CO.

11.6 Caller ID Technical Specifications

For information about caller ID technical specifications, contact the appropriate authority and request the technical references you require:

CLASS

CLASS documents are published by Telcordia Technologies (previously Bellcore). To obtain a copy of these technical references, visit http://www.telcordia.com.

- TR-NWT-000031 (issue 4) CLASS Feature Calling Number Delivery
- TR-NWT-001188 CLASS Feature Calling Name Delivery Generic Requirements
- TR-NWT-000030 (issue 2) Voice Data Transmission Interface Generic Requirement

ACLIP

Contact the Telecommunications Authority of Singapore and Telcordia Technologies.

- TAS TS PSTN1 A-CLIP: 1994
- Bellcore specification TR-NWT-000030 (see Telcordia Technologies contact info provided in CLASS)

CLIP

Contact British Telecommunications.

- SIN (Supplier Information Note) 242 (issue 01)
- SIN (Supplier Information Note) 227 (issue 01)

JCLIP

Contact Nippon Telegraph and Telephone Corporation.

- Telephone Service Interfaces, Edition 5, Technical Reference

Global Tone Detection and Generation, and Cadenced Tone Generation 12

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

12.1 Global Tone Detection (GTD)

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

12.1.1 Overview of Global Tone Detection

Global tone detection (GTD) allows you to define single or dual frequency tones for detection. The characteristics of a tone are defined in a GTD tone template. A tone template contains 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**.

The typical use of global tone detection is to detect single and dual frequency tones other than those automatically provided with the voice software. This includes tones outside the standard DTMF set (0-9, a-d, *, and #), and the standard MF set (0-9, a-c, and *). GTD works simultaneously with DTMF and MF tone detection.

When GTD detects a tone, it responds by producing either a **tone event** on the event queue or a **digit** on the digit queue. The particular response depends on the GTD tone configuration.

12.1.2 Global Tone Detection on HMP Software versus Springware Boards

On Dialogic® HMP Software, tone templates are managed internally on a board basis, while on Springware boards, tone templates are managed internally on a channel basis.

On Dialogic® HMP Software, once a tone template is defined, it can be added to any number of channels for global tone detection on a board. The tone template is stored only once on the board. A counter in the tone template tracks the number of channels that are using this template; the template remains active until the very last channel of the board deletes the template.

On Dialogic® HMP Software, the memory available for all tone templates (including call progress analysis tones and user-defined tones) is pre-allocated and fixed. Each tone template takes up the same amount of memory. However, a limitation exists on the number of tone templates that can be active at one time on a channel, due to the number of events that are generated. See Section 12.1.8, “Guidelines for Creating User-Defined Tones”, on page 115 for more information.

On Dialogic® Springware boards, once a tone template is defined, it can be added to any number of channels for global tone detection as well. However, the tone template is stored for each channel that uses it.

12.1.3 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.

12.1.4 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.

API Library Functions

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.

dx_blddt( )
Defines a simple dual frequency tone. Subsequent calls to **dx_addtone( )** will use this tone until another tone is defined.

Note that the boards cannot 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. 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. 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 build-tone functions. The amplitudes remain valid for all tones built until **dx_setgtdamp( )** is called again and amplitudes are changed.

Instructions for Building Tone Templates

Follow these steps to build a tone template for global tone detection:

1. Determine the characteristics of the tone you wish to detect: single frequency or dual frequency tone, frequency range, cadence components, and so on.
2. Use the appropriate build-tone function, such as **dx_bldst( )**, **dx_blddt( )** and so on, to build and define the tone template. The functions require that you specify a unique tone ID for each tone template.
3. Use the **dx_addtone( )** function to add the tone template to a channel. Subsequent calls to **dx_addtone( )** will add this tone template until another tone is defined. Adding a tone template to a channel enables detection of a tone on that channel.
4. Repeat steps 1 - 3 as needed.

Tips and Hints for Building Tone Templates

The following are tips and hints when building a tone template for global tone detection:

- After using a build-tone function to define a new tone template, you must call **dx_addtone( )** to add the tone template to a channel and enable detection of that tone on a channel.
- After using a build-tone function to define a tone template, if the template is not added to a channel, the next call to a build-tone function will overwrite the tone definition contained in the previous template.
- Only one tone template can be created at a time; however, 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 disabled or enabled by using **dx_distone( )** and **dx_enbtone( )**. See Section 12.1.5, "Working with Tone Templates", on page 113 for more information.
- Each tone template must have a unique tone ID.
- On Windows®, do not use tone IDs 261, 262 and 263; they are reserved for library use.
- A particular tone template that has been added to a channel 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. See Section 12.1.5, "Working with Tone Templates", on page 113 for more information.

Table 13 lists some standard Bell System Network call progress tones. The frequencies are useful to know when creating tone templates.

Table 13. 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

12.1.5 Working with Tone Templates

After building a tone template, 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 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 detection of a user-defined tone on a channel as well as the DE_TONEON and/or DE_TONEOFF events for that tone. Detection capability for user-defined tones is enabled on a channel by default when **dx_addtone( )** is called.

dx_enbtone( )

Enables detection of a user-defined tone that was previously disabled by **dx_distone( )**. Also enables detection of DE_TONEON and/or DE_TONEOFF events for that tone. Detection capability for user-defined tones is enabled on a channel by default when **dx_addtone( )** is called.

12.1.6 Retrieving Tone Events

Tone-on events (DE_TONEON) and tone-off events (DE_TONEOFF) are call status transition (CST) events. Retrieval of these events is handled differently for asynchronous and synchronous applications. Table 14 outlines the different processes for retrieving tone events.

Table 14. 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 Dialogic® 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 Dialogic® HMP Software versus on Springware boards. On Dialogic® HMP Software, 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 Dialogic® HMP Software and on Dialogic® Springware boards, a DE_TONEOFF event is reported when the tone is no longer present.

12.1.7 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.

12.1.8 Guidelines for Creating User-Defined Tones

The following guidelines apply when creating user-defined tones:

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 user-defined 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.)
- On Dialogic® HMP Software, the maximum number of user-defined tone templates is 40.

  Note: In the case where the number of of tone templates is exhausted, no error is returned on **dx_addtone( )** and subsequent tone detection may fail.
- On Dialogic® HMP Software, the number of user-defined tone templates that are active and enabled on a channel is limited due to the number of events that a channel can report. The default maximum number of events for a channel is 20. (A tone-on event and a tone-off event in the same tone template count as two events.)
- On Dialogic® HMP Software, the default call progress analysis tones (which include tri-tone special information tone sequences or SIT sequences) are created at board initialization time and are available for use. If you create new call progress analysis tone templates using **dx_querytone( )**, **dx_deletetone( )** and **dx_createtone( )**, each tone template counts as a user-defined tone template, which reduces the number of user-defined tones you can create.
- On Dialogic® HMP Software, building and adding tones of zero frequency values to a tone template can cause firmware failures.
- On Dialogic® Springware boards, 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.
- On Springware boards, if you use call progress analysis to identify tri-tone special information tone (SIT) sequences, call progress analysis creates GTD tones, which reduces the number of user-defined tones you can create. Call progress analysis creates one GTD tone template for each single frequency tone that you define in the DX_CAP structure.
- On Springware boards, 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.
- On Springware boards, if you use a build tone function (such as **dx_blddt( )**) 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.
- On Springware boards 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 Springware boards on Linux, if you initiate call progress analysis and there is not enough memory to create the SIT sequences internally, you will get a T_CAERROR event and the **evtdata** field will contain MEMERR.
- On Springware boards on Windows, if you initiate call progress analysis and there is not enough memory to create the SIT sequences internally, you will get a CR_MEMERR.
- 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.

12.1.9 Global Tone Detection Application

A sample application for global tone detection (GTD) is 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 tone-on time, tone-on time deviation, tone-off time, tone-off time deviation, and repetition count parameters in the **dx_bldstcad( )** or **dx_blddtcad( )** functions.

On Dialogic® HMP Software, to detect leading edge debounce time, 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 **ontdev**, specify -1/2 of the desired debounce time
- For **offtime**, specify 0
- For **offtdev**, specify 0
- For **repcnt**, specify 0

On Dialogic® Springware boards, to detect leading edge debounce time, specify the following values for the **dx_bldstcad( )** or **dx_blddtcad( )** function parameters listed below:

- For **ontime**, specify the desired debounce time
- For **ontdev**, specify 3
- For **offtime**, specify 0
- For **offtdev**, specify 0
- For **repcnt**, specify 1

Note: The **dx_blddt( )** and **dx_bldst( )** functions cannot be used to detect leading edge debounce time because they do not have timing field parameters.

12.2 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

12.2.1 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

12.2.2 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.

12.2.3 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.

12.3 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

12.3.1 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.

12.3.2 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 14.

Figure 14. Example of Custom Cadenced Tone Generation

Define the Signal as a Dialogic Custom Cadence Tome

Description of the Cadence Tome Used in this Example

■ **Set the TN_GENCAD Parameters**

```
tngencad.cycles = 255;
tngencad.numsegs = 2;
tngencad.offtime (0) = 0;
tngencad.offtime (1) = 300;
dx_bidtngen(&tgencad.tone(0),440, 480, -16, -16,
100)
```

■ **Call thhe dx_playtoneEx( )**

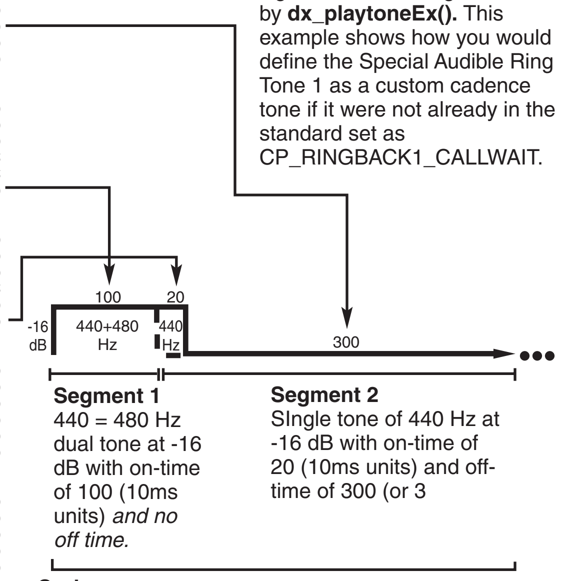
```
dx_playtoneEx (dxxxdev, &tngencad, tpt, EV_SYNC)
```

■ **The TN_GENCAD Definition and Resulting Signal**

```
TN_GENCAD      tngencad
cycles      =    255
numsegs     =      2
offtime(0)  =      0
offtime(1)  =    300
offtime(2)  =      0
offtime(3)  =      0

TN_GEN tone (0)
dflag(0)    =      0
tgfreq1(0)  =    440
tgfreq2(0)  =    480
tg_amp11(0) =    -16
tg_amp12(0) =    -16
tg_dur(0)   =    100

TN_GEN tone (1)
dflag(1)    =      0
tgfreq1(1)  =    440
tgfreq2(1)  =      0
tg_amp11(1) =    -16
tg_amp12(1) =      0
tg_dur(1)   =     20

TN_GEN tone (2)
dflag(2)    =      0
tgfreq1(2)  =      0
tgfreq2(2)  =      0
tg_amp11(2) =      0
tg_amp12(2) =      0
tg_dur(2)   =      0

TN_GEN tone (3)
dflag(3)    =      0
tgfreq1(3)  =      0
tgfreq2(3)  =      0
tg_amp11(3) =      0
tg_amp12(3) =      0
tg_dur(3)   =      0
```

Signal Description:
Repetition of combined tones (440 = 480 Hz) ON for 0.8 to 1.2 seconds, followed by 440 Hz tone ON for 0.2 seconds, and tone OFF for 2.7 to 3.3 seconds applied within a power range of -14.5 to 17.5 dBm.

Note:
Dialogic provides a predefined set of standard call progress signals that can be generated by **dx_playtoneEx().** This example shows how you would define the Special Audible Ring Tone 1 as a custom cadence tone if it were not already in the standard set as CP_RINGBACK1_CALLWAIT.



Segment 1
440 = 480 Hz dual tone at -16 dB with on-time of 100 (10ms units) *and no off time.*

Segment 2
SIngle tone of 440 Hz at -16 dB with on-time of 20 (10ms units) and off-time of 300 (or 3

Cycle:
2 segments repeating indefinitely, or until a tpt termination occurs.

12.3.3 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
```

12.3.4 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 examples of TN_GENCAD, see the definitions of standard call progress signals in Table 16, "TN_GENCAD Definitions for Standard PBX Call Progress Signals", on page 126.

12.3.5 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 15, "Standard PBX Call Progress Signals", on page 123 and note the signal ID (see also Figure 15, "Standard PBX Call Progress Signals (Part 1)", on page 124).
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)
   ```

12.3.6 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 Dialogic:

- Table 15, "Standard PBX Call Progress Signals", on page 123 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 15, "Standard PBX Call Progress Signals (Part 1)", on page 124 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 16, "TN_GENCAD Definitions for Standard PBX Call Progress Signals", on page 126 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 12.3.7, "Important Considerations for Using Predefined Call Progress Signals", on page 127 describes what standard was used, how the standard was implemented, information regarding the signal power levels, usage and other considerations.

Table 15. 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 15. Standard PBX Call Progress Signals (Part 1)

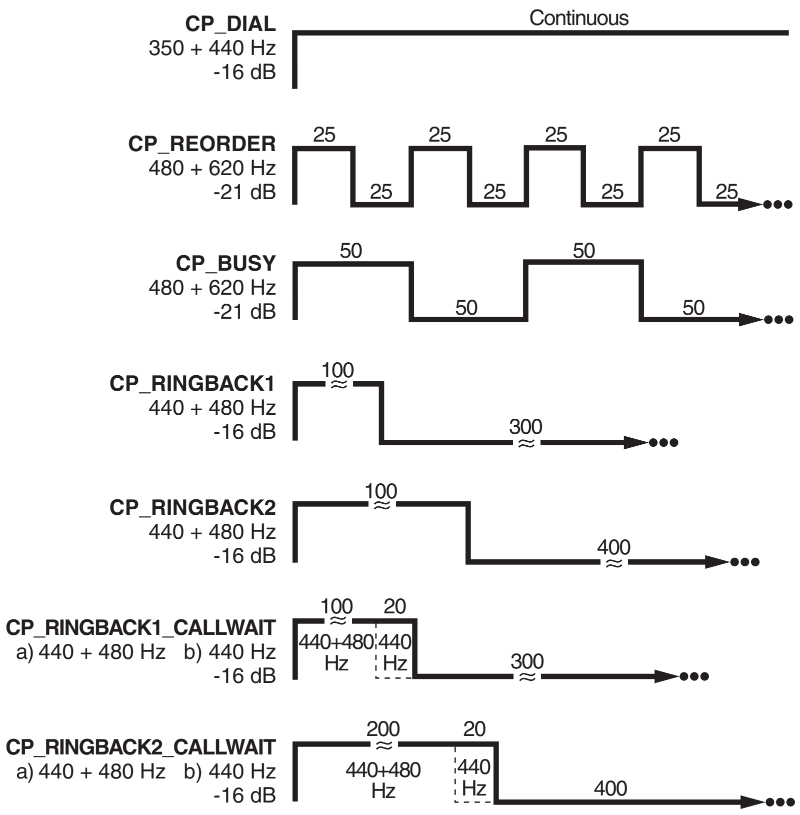


Figure 16. Standard PBX Call Progress Signals (Part 2)

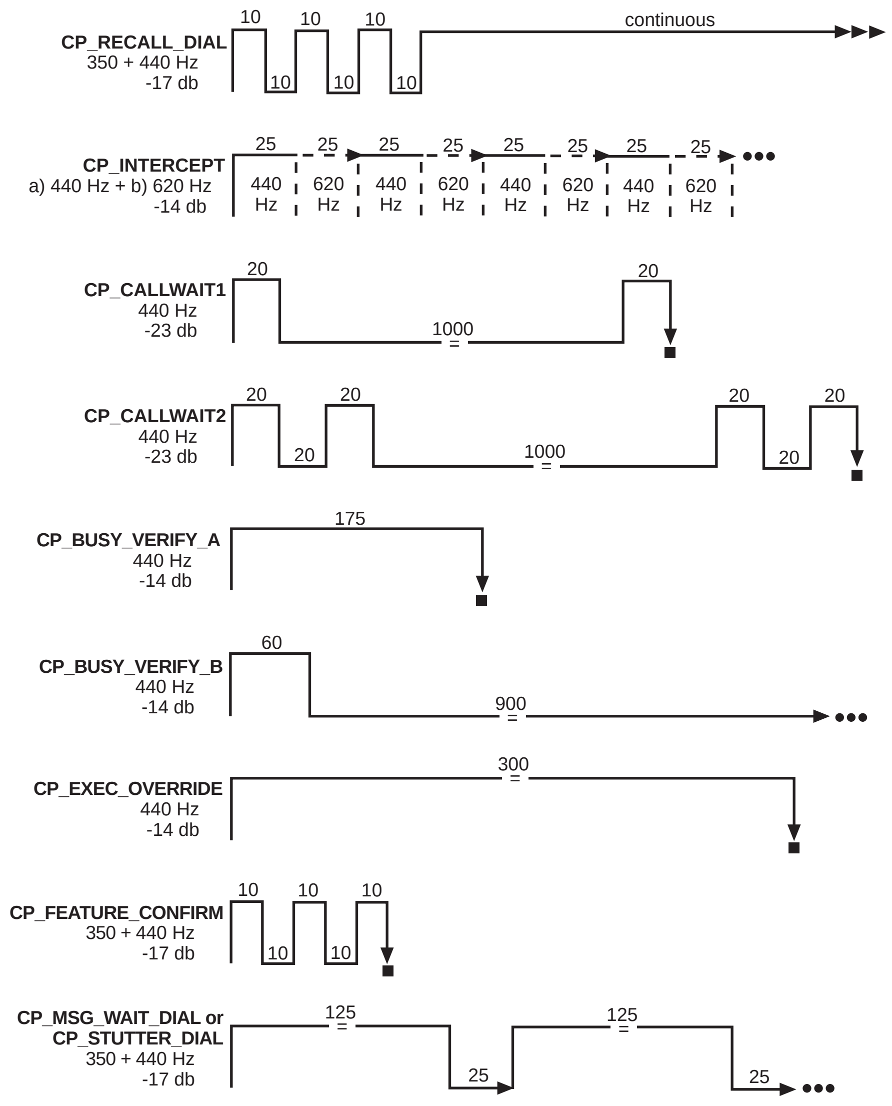


Table 16. TN_GENCAD Definitions for Standard PBX Call Progress Signals

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 16. 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**)

12.3.7 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 Dialogic® Springware board or 40 on Dialogic® DM3 board) 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 Dialogic 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 Dialogic 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.

Global Dial Pulse Detection 13

Global dial pulse detection (global DPD) is a signaling component of the voice library. The following topics provide more information on global DPD:

13.1 Overview

Global dial pulse detection is supported on Dialogic® Springware boards only.

Dial Pulse Detection (DPD) allows applications to detect dial pulses from rotary or pulse phones by detecting the audible clicks produced when a number is dialed, and to use these clicks as if they were DTMF digits. Dialogic global dial pulse detection, called global DPD, is a software-based dial pulse detection method that can use country-customized parameters for accurate performance.

Global DPD provides the following features and benefits:

- The algorithm is adaptive and can train on a DPD digit encountered, with the greatest accuracy produced from training on a digit that has 5 or more pulses. Global DPD does not require a leading “0” to train the global DPD algorithm.
- Can be performed simultaneously with DTMF detection. The application can determine whether the digit detected is a DTMF or DPD digit.
- Can be performed simultaneously with Global Tone Detection (GTD). For example, the application can use GTD to monitor for dial tone or busy tones simultaneously with DPD.
- Supports pulse-digit cut-through during a voice playback, with the correct digit returned in the digit buffer. Global DPD uses echo cancellation, which provides more accurate reporting of digits during voice playback.
- The application can enable global DPD and volume control. (Previously, there was a restriction that DPD digits had to be sent to the event queue instead of the digit queue if volume control was enabled.)

The following applications are supported by the global DPD feature:

- Analog applications using the loop-start telephone interface on a supported voice board
- Digital applications using a supported voice board

13.2 Global DPD Parameters

This feature is referred to as global DPD because its detection algorithm supports a wide range of dial pulses from 8 pulse-per-second (PPS) to 22 PPS telephones.

Customized global DPD download parameters are provided for several countries such as Argentina, Brazil, Colombia, India, Japan, Mexico and Venezuela.

On Linux, these parameters are contained in the *voice.prm* file. To download the global DPD parameters, select a specific country when the boards are configured.

On Windows®, configure DPD and select a specific country on the Country Property Sheet using the Dialogic® Configuration Manager (DCM).

13.3 Enabling Global DPD

Global Dial Pulse Detection (DPD) is available by default via software; separate GDPD enablement packages are no longer required.

Global DPD must be implemented on a call-by-call basis. Use **dx_setdigtyp( )** to enable DPD.

For any digit detected, you can determine the digit type such as DTMF or DPD by using the DV_DIGIT data structure in the application. When a **dx_getdig( )** or **dx_getdigEx( )** function call is performed, the digits are collected from the firmware and transferred to the user's digit buffer. The digits are stored as an array inside the DV_DIGIT structure.

You then use a pointer to the structure that contains a digit buffer. For an example, see Section 13.10, "Global DPD Example Code", on page 133. This method allows you to determine very quickly whether a pulse or DTMF telephone is being used.

13.4 Global DPD Programming Considerations

The global DPD algorithm will accurately detect digits in the supported regions without requesting a special training digit from the caller or requiring any other restrictions on the application. However, consider the following programming guidelines when designing the application:

- Talk-off rejection (the ability of the algorithm to distinguish between dial pulses and the human voice) will improve after the first digit is detected.
- Digit detection will be slightly more accurate (about 2%) after detecting a digit of 5 or greater. It is not necessary to dial a special training digit to do this. The application may simply restrict

the first menu to digits 5, 6, 7, 8, 9, and 0, and the training will be complete. Subsequent menus may be unrestricted.

- In general, detection accuracy is greater for higher digits than for lower. While detection accuracy is very high, it may be further improved by restricting menu selections, whenever convenient, to digits greater than 3.

13.5 Retrieving Digits from the Digit Buffer

To get the digits from the digit buffer, use the following synchronous programming model:

1. Define a data structure of type DV_DIGIT (the DV_DIGIT structure is defined by including the *dxxxlib.h* header file).
2. Enable DPD on the desired channels using the **dx_setdigtyp( )** function.
3. For each new connection, use **dx_setdigtyp( )** with the D_DPDZ mask, which initializes the DPD algorithm. After collecting the first DPD digit string, the mask can be set to D_DPD for the remainder of that connection. Each subsequent invocation of **dx_setdigtyp( )** must use the D_DPD mask.
4. Execute the **dx_getdig( )** function to collect and transfer the digits to the user's digit buffer. The digits are stored in the dg_value field of the DV_DIGIT structure. The corresponding digit types (dial pulse, DTMF, and so on) are stored in the dg_type field of the DV_DIGIT structure. For more information, see the DV_DIGIT structure in the *Voice API Library Reference*.

13.6 Retrieving Digits as Events

To get the digits as events, use the following asynchronous programming model using the **dx_setevtmsk( )**, **sr_waitevt( )**, and **sr_getevtdatap( )** functions and the DX_CST data structure.

1. Since the supported voice boards come with channels capable of global DPD, you must enable DPD on the desired channels using the **dx_setdigtyp( )** function.
2. For each new connection, use **dx_setdigtyp( )** with the D_DPDZ mask, which initializes the DPD algorithm. After collecting the first DPD digit string, the mask can be set to D_DPD for the remainder of that connection. Each subsequent invocation of **dx_setdigtyp( )** must use the D_DPD mask.
3. Use **dx_setevtmsk( )** to enable digit detection.
4. Use **sr_waitevt( )** to wait for events.
5. When a CST event occurs, use **sr_getevtdatap( )** to retrieve the pointer to the DX_CST structure.
6. The cst_data field (DX_CST structure) for a DE_DIGITS event contains an ASCII digit (low byte) and the digit type (high byte).

13.7 Dial Pulse Detection Digit Type Reporting

Two defines are provided for identifying the dial pulse detection digit type, depending upon how the digit type is retrieved:

DG_DPD
: Dial pulse detection digit from the DX_EBLK event queue data (cst_data) through a DE_DIGITS Call Status Transition event

DG_DPD_ASCII
: Dial pulse detection digit from the DV_DIGIT dg_type digit buffer using **dx_getdig( )**

Obtaining the digit type for DPD digits is valid only in the case when the voice and DPD capabilities are both present on the same board. In the case where a voice board does not support DPD, you cannot detect DPD digits or obtain the DPD digit type even though you can enable DPD and digit type reporting without an error.

13.8 Defines for Digit Type Reporting

Use the defines as shown here to determine the digit type from the value returned in the dg_type (digit type) field from the DV_DIGIT digit buffer. If you get the digit from the DV_DIGIT dg_type digit buffer using **dx_getdig( )**, you should use the digit type define that has the "_ASCII" suffix. Otherwise, if you get the digit from the DX_EBLK event queue data (cst_data) through a DE_DIGITS Call Status Transition event, you should use the digit type define without the "_ASCII" suffix.

	Defines for dg_type from	
Digit Type	**Digit Buffer**	**Event Queue**
DTMF	DG_DTMF_ASCII	DG_DTMF
DPD	DG_DPD_ASCII	DG_DPD
MF	DG_MF_ASCII	DG_MF
GTD	DG_USER1_ASCII	DG_USER1
(user-defined)	DG_USER2_ASCII	DG_USER2
	DG_USER3_ASCII	DG_USER3
	DG_USER4_ASCII	DG_USER4
	DG_USER5_ASCII	DG_USER5

13.9 Implementing Global DPD

Use the following procedure to implement global DPD:

1. Define a data structure of type DV_DIGIT (this structure is specified in the *DXDIGIT.H* file).
2. Enable DPD on the desired channels using the **dx_setdigtyp( )** function. For new calls you must use the D_DPDZ mask that initializes the DPD detector for new calls.

3. Execute the **dx_getdig( )** function to collect and transfer the digits to the user's digit buffer. The digits are stored in the dg_value field of the DV_DIGIT structure with the corresponding digit types stored in the dg_type field of the DV_DIGIT structure.

13.10 Global DPD Example Code

The following example illustrates how to set up and use global DPD. The code uses the synchronous model.

```
/*$ dx_setdigtyp( )and dx_getdig( ) example for global dial pulse detection $*/

#include        <stdio.h>
#include        "srllib.h"
#include        "dxxxlib.h"

void main(int argc, char **argv)
{

   int     dev;                    /* Dialogic device handle */
   DV_DIGIT        dig;
   DV_TPT tpt;

   /*
    * Open device, make or accept call
    */

   /* set up TPT to wait for 3 digits and terminate */
   dx_clrtpt(&tpt, 1);
   tpt.tp_type =   IO_EOT;
   tpt.tp_termno = DX_MAXDTMF;
   tpt.tp_length = 3;
   tpt.tp_flags =  TF_MAXDTMF;

   /* enable DPD and DTMF digits */
   dx_setdigtyp(dev, D_DPDZ|D_DTMF);

   /* clear the digit buffer */
   dx_clrdigbuf(dev);

   /* collect 3 digits from the user */
   if (dx_getdig(dev, &tpt, &dig, EV_SYNC) == -1) {
      /* error, display error message */
      printf("dx_getdig error %d, %s\n", ATDV_LASTERR(dev), ATDV_ERRMSGP(dev));
   } else {
      /* display digits received and digit type */
      printf("Received \"%s\"\n", dig.dg_value);
      printf("Digit type is ");
      /*
       * digit types have 0x30 ORed with them strip it off
       * so that we can use the DG_xxx equates from the header files
       */
      switch ((dig.dg_type[0] & 0x000f)) {
         case DG_DTMF:
            printf("DTMF\n");
            break;
         case DG_DPD:
            printf("DPD\n");
            break;
         default:
            printf("Unknown, %d\n", (dig.dg_type[0] &0x000f));
      }
}

/*
 * continue processing call
 */
```

Building Applications 14

This chapter provides information on building applications using the Dialogic® Voice API library. The following topics are discussed:

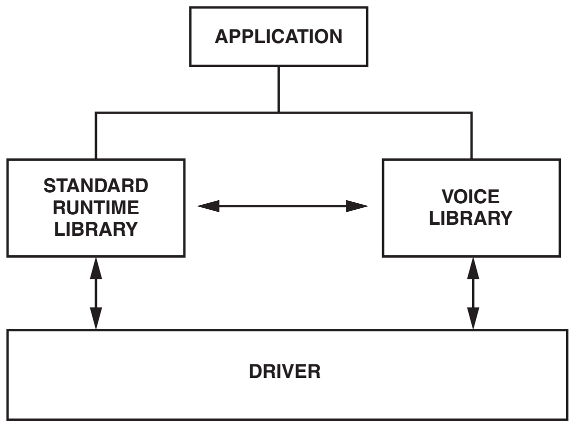
14.1 Dialogic® Voice and SRL API Libraries

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

The Dialogic® Voice API library and the Dialogic® 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 *Dialogic® Standard Runtime Library API Programming Guide* and *Dialogic® Standard Runtime Library API Programming Guide*.

Figure 17 illustrates how the Dialogic® Voice and SRL API libraries interface with the driver.

Figure 17. Dialogic® Voice and SRL API Libraries



14.2 Compiling and Linking

The following topics discuss compiling and linking requirements:

- Include Files
- Required Libraries for Linux
- Required Libraries for Windows®
- Variables for Compiling and Linking

14.2.1 Include Files

Function prototypes and equates are defined in include files, also known as header files. Applications that use the Dialogic® Voice API 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 Dialogic® Voice API library functions:

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

dxxxlib.h
: Contains function prototypes and equates for the Dialogic® Voice API library. Used for voice processing applications.

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

14.2.2 Required Libraries for Linux

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

You must link the following shared object library files **in the order shown** when building 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.

Note: When building an application, list Dialogic® libraries before all other libraries.

14.2.3 Required Libraries for Windows®

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

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

libdxxmt.lib
: Main voice library file.

libsrlmt.lib
: Standard Runtime Library file.

14.2.4 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_DIR
: Variable that points to the directory in which the release software is installed.

INTEL_DIALOGIC_INC
: Variable that points to the directory in which header files are stored.

INTEL_DIALOGIC_LIB
: Variable that points to the directory in which shared library files are stored.

These variables are automatically set at login and should be used in compiling and linking commands.

Index