Voice Programmer's Guide

for Windows NT

Voice and Loop-Start Boards
Release 4.25SC

System Release 4.25SC software only supports hardware configurations that use SCbus

COPYRIGHT NOTICE

The contents of this document are subject to change without notice. Every effort has been made to ensure the accuracy of this document. However, due to ongoing Product improvements and revisions, Dialogic cannot guarantee the accuracy of printed material after the date of publication nor can it accept responsibility for errors or omissions. Dialogic will publish updates and revisions to this document as needed.

The software referred to in this document is provided under a Software License Agreement. Refer to the Software License Agreement for complete details governing the use of the software.

Publication Date: September, 1996
Dialogic Corporation
1515 Route Ten
Parsippany, NJ 07054

Table of Contents

List of Tables

1. Voice Software Reference Overview

1.1. Voice Product Terminology

The following product naming conventions are used throughout this guide:

D/2x refers to any model of the Dialogic DIALOG series of 2-channel voice-store-and-forward expansion boards. This series includes **D/21D**, **D41/E**, and **D41ESC** and boards.

D/4x refers to any model of the Dialogic DIALOG series of 4-channel voice store-and-forward expansion boards. This series includes the **D/41D**, **D/41E**, and **D/41ESC** boards.

D/12x refers to any model of the Dialogic series of 12-channel voice-store-and-forward expansion boards. **D/120, D/121, D/121A,** and **D/121B** are specific models of this board.

D/81A refers to the Dialogic 8-channel voice-store-and-forward expansion board.

D/160SC refers to the Dialogic 16-channel voice board with onboard analog loop start interface.

D/240SC refers to the Dialogic 24-channel voice board for use with a network interface board.

D/240SC-T1 refers to the Dialogic 24-channel voice board with onboard T-1 digital interface.

D/300SC-E1 refers to the Dialogic 30-channel voice board with onboard E-1 digital interface.

D/320SC refers to the Dialogic 32-channel voice board for use with a network interface board.

D/xxx refers to D/2x, D/4x, D/81A and D/12x expansion boards.

D/xxxSC refers to voice and telephone network interface resource boards that communicate via the SCbus. These boards include **D/41ESC, D/160SC-LS, D/240SC, D/240SC-T1, D/300SC-E1**, and **D/320SC**.

DIALOG/HD or **Spancard** refers to voice and telephone network interface resource boards that communicate via the SCbus. These boards include **D/160SC-LS, D/240SC, D/240SC-T1, D/300SC-E1**, and **D/320SC**.

DTI/xxx refers to any of Dialogic's digital telephony interface expansion boards for the AT-bus architecture. These boards include: **DTI/101, DTI/211, DTI/212, DTI/240SC,** and **DTI/300SC** boards.

FAX/xxx refers to Dialogic's FAX resource expansion boards. FAX/120 is a 12-channel model that connects to a D/121A or D/121B board.

Firmware Load File refers to the firmware file that is downloaded to a Voice board. This file has a *.fwl* extension.

LSI refers to Dialogic's PEB-based loop start interface expansion boards. The **LSI/120** is a specific model of this board. **LSI/80-int** refers to the international versions of Dialogic's loop start interface expansion boards.

PEB is the PCM expansion bus connecting the D/81A or D/12x voice boards to the network interface boards.

SCbus is the TDM (Time Division Multiplexed) bus connecting SCSA (Signal Computing System Architecture) voice, telephone network interface and other technology resource boards together.

Spancard same as **DIALOG/HD**.

SpringBoard refers to the hardware platform used with the D/21D, D/41D, D/21E, D/41E, D/81A, D/121, D/121A, and D/121B board.

SpringWare refers to the software algorithms built into the downloadable firmware that provides the voice processing features available on all Dialogic voice boards.

VFX/40ESC is a Dialogic SCbus voice and FAX resource board with on-board loop-start interfaces. The **VFX/40ESC** board provides 4-channels of enhanced voice and FAX services in a single slot.

Voice hardware and software refers to D/2x, D/4x, D/81A, D/12x, and D/xxxSC expansion boards and associated software.

1.2. Organization of This Voice Reference Guide

The *Voice Programmer's Guide for Windows NT* describes the voice software for Windows NT and provides instructions for using the Voice Driver and Voice Libraries.

Chapter *1. Voice Software Reference Overview* provides an overview of the voice software. It lists each of the components and supported Dialogic boards and describes the Voice Driver and Voice Libraries.

Chapter *2. Using the Voice Reference Library* provides general information about the Voice Library *libdxxmt.lib*. It provides an overview of the function categories, and describes the programming requirements when using the library.

Chapter *3. Voice Function Reference* provides a complete function reference (in alphabetical order) for all of the functions in the Voice Library.

Chapter *4. Voice Data Structures and Device Parameters* describes the following topics:

- data structures and tables contained in the Voice Library.
- parameter defines for Voice Devices that can be set or retrieved using **dx_getparm( )** and **dx_setparm( )**.

Chapter *5. Voice Programming Conventions* lists programming techniques that simplify programming with the Dialogic Voice Library.

Appendix A lists the voice device entries and returns for the Standard Runtime Library.

Appendix B list the Voice Library error defines.

Appendix C describes differences and similarities between DTMF and MF tones.

Appendix D provides a list of related Dialogic publications.

A **Glossary** and an **Index** are also provided.

This chapter provides an overview of the voice software for Windows NT. The voice software consists of the following:

- Voice Driver
- Voice Library of C functions
- Standard Runtime Library

The Voice Driver and Voice Libraries are described in this chapter. For information about the remaining voice software see the following:

- For installation of the voice software and demonstration program, see the *System Release Software Installation Reference for Windows NT*
- For the Standard Runtime Library, see Appendix A and the *Standard Runtime Library Programmer's Guide for Windows NT*
- For Voice Driver Features and a description of the Demonstration program, see the *Voice Features Guide for Windows NT*

1.3. Voice Driver

The Voice Driver communicates with and controls the voice hardware. Voice hardware consists of voice store-and-forward boards which include the following boards:

For SCbus-based applications:

- D/41ESC
- D/160SC-LS
- D/240SC, D/320SC
- D/240SC-T1, D/300SC-E1
- DTI/241SC, DTI/301SC
- LSI/81SC, LSI/161SC

The D/41ESC, D/160SC-LS, D/240SC, D/320SC, D/240SC-T1, and D300SC-E1 boards support a range of Voice Processing features such as:

- Record and playback of voice data
- Speed and volume control of play
- Call handling
- Call Analysis - Basic and Enhanced
- DTMF, MF, and R2MF tone generation and detection
- Global Tone Generation and Detection

PerfectCall Call Analysis, Speed and Volume Control, Global Tone Detection, and Global Tone Generation are supported on the DSP-based D/xxx boards (D/21D, D41D, D/21E, D/41ESC, D81A, D/121, D/121A, D/121B, D/160SC-LS, D/240SC, D240SC-T1, D300SC-E1, and D/320SC).

The DTI/241SC, DTI/301SC, LSI/81SC, and LSI/161SC boards support the following Voice Processing features:

- Call handling
- Call Analysis - Basic and Enhanced
- DTMF, MF, and R2MF tone generation and detection
- Global Tone Generation and Detection

PerfectCall Call Analysis, Speed and Volume Control, Global Tone Detection, and Global Tone Generation are supported on the DSP-based D/xxx boards (D/21D, D41D, D/21E, D/41ESC, D81A, D/121, D/121A, D/121B, D/160SC-LS, D/240SC, D240SC-T1, D300SC-E1, and D/320SC).

User-defined tones are CST events, but detection for these events is enabled using **dx_addtone( )** or **dx_enbtone( )**. See the *Voice Features Guide* for information on Global Tone Detection functions.

Boards are treated as *board devices* and channels are treated as *channel devices* or *board subdevices* by the Voice Driver.

The SCbus is a real-time, high speed, time division multiplexed (TDM) communications bus connecting Signal Computing System Architecture (SCSA) voice, telephone network interface and other technology resource boards together. SCbus boards are treated as board devices with on-board voice and/or telephone network interface devices that are identified by a board and channel (time slot for

digital network channels) designation, such as a voice channel, analog channel or digital channel.

For more information on the SCbus and SCbus routing, refer to the *SCbus Routing Guide* and the *SCbus Routing Function Reference for Windows NT*.

1.4. Voice Libraries

The voice libraries provide the interface to the Voice Driver. The voice libraries for single threaded and multithreaded applications include:

- *libdxxmt.lib* - the main Voice Library
- *libsrlmt.lib* - the Standard Runtime Library

These "C" function libraries can be used to:

- Utilize all the voice board features
- Write applications using a **Single Threaded Synchronous** or Multithreaded **Asynchronous** programming model
- Configure devices
- Handle events that occur on the devices
- Return device information

The voice library *libdxxmt.lib*, which contains most of these functions, is described in more detail in *2. Using the Voice Reference Library*.

The Standard Runtime Library *libsrlmt.lib* is described in the *Standard Runtime Library Programmer's Guide for Windows NT.* This library provides a set of common system functions that are device independent and are applicable to all Dialogic devices (e.g., D/240SC-T1 and FAX/120 boards). You can use these functions to simplify application development by writing common event handlers to be used by all devices.

1.4.1. Single Threaded Asynchronous Programming Model

Single threaded asynchronous programming enables a single program to control multiple voice channels within a single thread. This allows the development of complex applications where multiple tasks must be coordinated simultaneously. The asynchronous programming model supports both polled and callback event management.

The *Standard Runtime Library Programmer's Guide for Windows NT* contains a full discussion of the asynchronous programming models.

1.4.2. Multithreaded Synchronous Programming Model

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

The *Standard Runtime Library Programmer's Guide for Windows NT* contains a full discussion of the synchronous programming models.

1.4.3. Extended Asynchronous Programming Model

This model is similar to the asynchronous except it is implemented using the **sr_waitevtEx( )** function. This allows an application to have different threads waiting on events on different devices. As with the basic asynchronous model, functions initiated asynchronously from a different thread and the completion event picked up the **sr_waitevtEx( )** thread.

The *Standard Runtime Library Programmer's Guide for Windows NT* contains a full discussion of the extended asynchronous programming model.

2. Using the Voice Reference Library

This chapter provides a description of the voice library and the programming requirements. The following topics are included:

- Voice Library and its function categories (*Section 2.1*)
- Programming requirements for the Voice Library (*Section 2.2*).

2.1. Voice Library

The Voice Library functions provide an interface to the Voice Device Driver. The functions can be divided into the following major categories:

Device Management	• open and close devices
Configuration	• alter configuration of devices
I/O	• transfer data to and from devices
Convenience	• simplify play and record
Call Status Transition Event	• set and monitor events on devices
Route	• for SCbus boards, connect the receive (listen) channel of an SCbus board to an SCbus time slot; the transmit of each channel device is connected to a unique and unchangeable SCbus time slot at system initiation and download.
Global Tone Detection	• enable user-defined tone detection
Global Tone Generation	• enable user-defined tone generation
R2MF Convenience	• detect and generate R2MF tones
Speed and Volume Convenience	• enable play-speed and play-volume control • convenience functions for adjusting speed and volume control
Structure Clearance	• clear data structures
Extended Attribute	• retrieve device information

This section lists the functions that belong to each category and describes the characteristics of each category.

In the *Function Reference* (*3. Voice Function Reference*) each function is described in detail, and the function header includes the category to which the function belongs.

2.1.1. Device Management Functions

dx_close( )	• close a board or channel
dx_open( )	• open a board or channel

The Device Management functions open and close devices (boards and channels). For SCbus configurations using a D/240SC-T1 or D/300SC-E1 board, each board comprises a digital interface device with independent channels/time slots (dtiBxTx) and a voice device with independent channels (dxxxBxCx); where B is followed by the unique board number, C is followed by the number of the voice device channel (1 to 4) and T is followed by the number of the digital interface device time slot (digital channel)(1 to 24 for T-1; 1 to 30 for E-1).

Before you can use any of the other library functions on a device, that device must be opened. When the device is opened using **dx_open( )** the function returns a unique Dialogic device handle. The handle is the only way the device can be identified once it has been opened. The **dx_close( )** function closes a device via its handle.

Device Management functions do not cause a device to be busy. In addition, the Device Management functions will work on a device whether the device is busy or idle.

NOTES:

1. Issuing a **dt_open( )**, **dx_open( )**, **dt_close( )** or **dx_close( )** while the device is being used by another process will not affect the current operation of the device.
2. The device handle which is returned is Dialogic defined. The device handle is not a standard Windows NT file descriptor. Any attempts to use operating system commands such as **read( )**, **write( )**, or **ioctl( )** will produce unexpected results.
3. In an application that starts a process, the device handle is not inheritable by the child process. Devices must be opened in the child process.

2.1.2. Configuration Functions

dx_clrdigbuf( )	• clear the firmware digit buffer
dx_getparm( )	• get a board/channel device parameter
dx_setdigtyp( )	• set digit collection type
dx_sethook( )	• set hookswitch state
dx_setparm( )	• set device parameters
dx_wtring( )	• wait for number of rings

Configuration functions allow you to alter, examine, and control the physical configuration of an open device. The configuration functions operate on a device only if the device is idle. All configuration functions cause a device to be busy and return the device to an idle state when the configuration is complete. See *Section 2.2.3. Busy and Idle States* for information about busy and idle states.

NOTE: The **dx_sethook( )** function can also be classified as an I/O function and can be run asynchronously or synchronously.

2.1.3. I/O Functions

dx_dial( ) (enable/disable call analysis)	• dial an ASCIIZ string of digits
dx_getdig( )	• get digits from channel digit buffer
dx_play( )	• play voice data from one or more sources
dx_playiottdata( )	• play voice data from multiple sources
dx_rec( )	• record voice data to one or more destinations
dx_reciottdata( )	• record voice data to multiple destinations
dx_setdigbuf( )	• set digit buffering mode
dx_stopch( )	• stop current I/O
dx_wink( )	• wink a channel

NOTES: 1. dx_playtone( ), which is grouped with the Global Tone eneration

functions, is also an I/O function and all I/O characteristics apply.

2. **dx_sethook( )**, which is grouped with the Configuration functions, is also an I/O function and all I/O characteristics apply.
3. **dx_wink( )**, cannot be called for a digital T-1 configuration that includes a D/240SC-T1 board. Transparent signaling for SCbus digital interface devices is not supported in System Release 4.1SC.

The purpose of an I/O function is to transfer data to and from an open idle channel. All I/O functions cause a channel to be busy while data transfer is taking place and return the channel to an idle state when data transfer is complete. The **dx_stopch( )** function stops any other I/O function, except **dx_dial( )** (see **dx_dial( )** and **dx_stopch( )** in the *Chapter 3. Voice Function Reference* for information).

I/O functions can be run synchronously or asynchronously. When running synchronously, they return after completing successfully or after an error. When running asynchronously they will return immediately to indicate successful initiation (or an error), and continue processing until a termination condition is satisfied. See the *Standard Runtime Library Programmer's Guide for Windows NT*, for a full discussion on asynchronous and synchronous operation.

A set of termination conditions can be specified for I/O functions (except **dx_stopch( )** and **dx_wink( )**). These conditions dictate what events will cause an I/O function to terminate. The termination conditions are specified just before the I/O function call is made. Obtain termination reasons for I/O functions by calling the Extended Attribute function **ATDX_TERMMSK( )**. See *Section* 2.2.4. I/O Terminations for information on I/O terminations.

NOTE: The **dx_stopch( )** function will not stop all I/O functions. Do not use this function to stop **dx_wink( )** or **dx_dial( )**(without Call Analysis enabled). See *Chapter 3. Voice Function Reference* for more information on these functions.

2.1.4. Convenience Functions

dx_playf( )	• play voice data from a single file
dx_playvox( )	• play a VOX file
dx_playwav( )	• play a WAVE file
dx_recf( )	• record voice data to a single file
dx_recvox( )	• record voice data to a single VOX file
dx_recwav( )	• record voice data to a single WAVE file

These functions simplify synchronous play and record.

dx_playf( ) performs a playback from a single file by specifying the filename. The same operation can be done by using **dx_play( )** and supplying 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_playvox( )**, **dx_playwav( )**, **dx_recvox( )**, **dx_recwav( )**, and **dx_recf( )** provide the same single-file convenience for the **dx_playiottdata( )**, **dx_reciottdata( )**, and **dx_rec( )** function.

Source code is included for **dx_playf( )** and **dx_recf( )** in the function descriptions in *Chapter 3. Voice Function Reference*.

NOTE: **dx_playf( ), dx_playvox( ), dx_playwav( ), dx_recf( ), dx_recvox( )** and **dx_recwav( )** run synchronously only.

2.1.5. Call Status Transition Event Functions

dx_getevt( )	• get call status transition event
dx_setevtmsk( )	• set call status transition event notification

Call Status Transition (CST) Event functions set and monitor Call Status Transition events that can occur on a device. Call Status Transition events indicate changes in the status of the call. For example, if rings were detected, if the line went onhook or offhook, or if a tone was detected. The full list of Call Status

Transition events is contained in *Section* 4.1.3. DX_CST - *call status transition structure* which describes the Call Status Transition structure (DX_CST).

dx_setevtmsk( ) enables detection of CST event(s).

dx_getevt( ) retrieves events in a synchronous environment. To retrieve CST events in an asynchronous environment, use the Standard Runtime Library's Event Management functions.

2.1.6. SCbus Routing Functions

See the *SCbus Routing Function Reference for Windows NT* for function descriptions and the nomenclature used to identify devices, channels and time slots in an SCbus configuration. The SCbus routing functions can only be used in SCbus configurations.

2.1.7. Global Tone Detection Functions

dx_addtone( )	• add a user-defined tone
dx_blddt( )	• build a dual frequency tone description
dx_blddtcad( )	• build a dual freqency tone cadence description
dx_bldst( )	• build a single frequency tone description
dx_bldstcad( )	• build a single frequency tone cadence description
dx_deltones( )	• delete user-defined tones
dx_enbtone( )	• enable detection of user-defined tones
dx_distone( )	• disable detection of user-defined tones
dx_setgtdamp( )	• sets amplitudes used by Global Tone Detection (GTD)

Use the Global Tone Detection (GTD) functions to define and enable detection of single and dual frequency tones that fall outside those automatically provided with the Voice Driver. This includes tones outside the standard DTMF range of 0-9, a-d, * and #.

The GTD **dx_blddt( )**, **dx_blddtcad(**), **dx_bldst( )**, and **dx_bldstcad( )** functions define tones which can then be added to the channel using **dx_addtone( )**. This enables detection of the tone on that channel.

See the *Voice Features Guide for Windows NT* for a full description of Global Tone Detection.

2.1.8. Global Tone Generation Functions

dx_bldtngen( )	• build a user-defined tone generation template
dx_playtone( )	• play a user-defined tone

Use Global Tone Generation functions to define and play single and dual tones other than those automatically provided with the Voice driver.

dx_bldtngen( ) defines a tone template structure, *TN_GEN*. **dx_playtone( )** can then be used to generate the tone.

See the *Voice Features Guide for Windows NT* for a full description of Global Tone Generation, and see 4. Voice Data Structures and Device Parameters for a description of the TN_GEN structure.

2.1.9. R2MF Convenience Functions

r2_creatfsig( )	• create R2MF forward signal tone
r2_playbsig( )	• play R2MF backward signal tone

These are convenience functions which enable detection of R2MF forward signals on a channel, and play R2MF backward signals in response. For more information about Voice Support for R2MF, see the *Voice Features Guide for Windows NT*.

2.1.10. Speed and Volume Functions

dx_adjsv( )	• adjust speed or volume
dx_clrsvcond( )	• clear speed or volume digit adjustment conditions

dx_setsvcond( )	• set speed or volume digit adjustment conditions
dx_getcursv( )	• get current speed and volume settings
dx_getsvmt( )	• get Speed/Volume Modification Table
dx_setsvmt( )	• set Speed/Volume Modification Table

NOTE: Speed and Volume Control are available on D/41ESC, D/160SC-LS, D/240SC, D/240SC-T1, D/300SC-E1, and D/320SC boards.

Use these functions to adjust the speed and volume of the play. A 21-entry Speed Modification Table and Volume Modification Table is associated with each channel. This table can be used for increasing or decreasing the speed or volume. This table has default values which can be changed using the **dx_setsvmt( )** function.

dx_adjsv( ) and **dx_setsvcond( )** both use the Modification Table to adjust speed or volume; **dx_adjsv( )** adjusts speed or volume immediately, and **dx_setsvcond( )** sets conditions (such as a digit) for speed or volume adjustment. **dx_clrsvcond( )** to clear the speed or volume conditions.

dx_getcursv( ) retrieves the current speed or volume settings. **dx_getsvmt( )** retrieves the settings of the current Speed or Volume Adjustment Table.

See the *Voice Features Guide for Windows NT* for more information about voice software support for speed and volume.

2.1.11. Speed and Volume Convenience Functions

dx_addspddig( )	• add speed adjustment digit
dx_addvoldig( )	• add volume adjustment digit

dx_addspdig( ) and **dx_addvoldig( )** are convenience functions that specify a digit and an adjustment to occur on that digit, without having to set any data structures. These functions use the default settings of the Speed/Volume Modification Tables.

2.1.12. PerfectCall Call Analysis Functions

dx_chgdur( )	• change PerfectCall Call Analysis signal duration
dx_chgfreq( )	• change PerfectCall Call Analysis signal frequency
dx_chgrepcnt( )	• change PerfectCall Call Analysis signal repetition count
dx_initcallp( )	• initialize PerfectCall Call Analysis on a channel
dx_chg( )	• functions can be used to change the definition of default PerfectCall Call Analysis tones.
dx_initcallp( )	• enables PerfectCall Call Analysis.

2.1.13. Structure Clearance Functions

dx_clrcap( )	• clear DX_CAP structure
dx_clrtpt( )	• clear DV_TPT structure

These functions do not affect a device. The **dx_clrcap( )** and **dx_clrtpt( )** functions provide a convenient method for clearing the DX_CAP and DV_TPT Voice Library data structures. These structures are discussed in *Chapter 4. Voice Data Structures and Device Parameters*.

2.1.14. Extended Attribute Functions

ATDX_ANSRSIZ( )	• Returns duration of answer detected during
Call Analysis	
ATDX_BDNAMEP( )	• Returns pointer to the device name string
ATDX_BDTYPE( )	• Returns board type
ATDX_BUFDIGS( )	• Returns number of digits in firmware since last dx_getdig( ) for a given channel
ATDX_CHNAMES( )	• Returns pointer to an array of channel name strings
ATDX_CHNUM( )	• Returns channel number on board associated with the channel device handle

ATDX_CONNTYPE( )	• Returns connection type for a call
ATDX_CPERROR( )	• Returns call analysis error
ATDX_CPTERM( )	• Returns last call analysis termination
ATDX_CRTNID( )	• Returns the identifier of the tone that causedthe most recent Call Analysis termination
ATDX_DEVTYPE( )	• Returns device type
ATDX_DTNFAIL( )	• Returns the dial tone character that indicates which dial tone Call Analysis failed to detect
ATDX_FRQDUR( )	• Returns duration of first frequency
ATDX_FRQDUR2( )	• Returns duration of 2nd SIT tone frequency
ATDX_FRQDUR3( )	• Returns duration of 3rd SIT tone frequency detected
ATDX_FRQHZ( )	• Returns frequency of first detected tone
ATDX_FRQHZ2( )	• Returns frequency of second detected SIT tone
ATDX_FRQHZ3( )	• Returns frequency of third detected SIT tone
ATDX_FRQOUT( )	• Returns % of frequency out of bounds detected during Call Analysis
ATDX_FWVER( )	• Returns firmware version
ATDX_HOOKST( )	• Returns current hook status
ATDX_LINEST( )	• Returns current line status
ATDX_LONGLOW( )	• Returns duration of longer silence detected during Call Analysis
ATDX_PHYADDR( )	• Returns physical address of board
ATDX_SHORTLOW( )	• Returns duration of shorter silence detected during Call Analysis
ATDX_SIZEHI( )	• Returns duration of non-silence detected during Call Analysis
ATDX_STATE( )	• Returns current state of the device
ATDX_TERMMSK( )	• Returns termination bitmap

ATDX_TONEID( )	• Returns the tone id
ATDX_TRCOUNT( )	• Returns last record or play transfer count

Voice Library Extended Attribute functions return information specific to the Voice device indicated in the function call. Many are related to specific Voice features:

Basic Call Analysis uses:

ATDX_ANSRSIZ( )
ATDX_CPERROR( )
ATDX_CPTERM( )
ATDX_FRQ( )
ATDX_LONGLOW( )
ATDX_SHORTLOW( )
ATDX_SIZEHI( )

PerfectCall Call Analysis uses:

ATDX_ANSRSIZ( )
ATDX_CPERROR( )
ATDX_CPTERM( )
ATDX_FRQ( )
ATDX_CRTNID( )
ATDX_DTNFAIL( )

The Call Status Transition event detection uses:

ATDX_HOOKST( )

Global Tone Detection uses:

ATDX_TONEID( )

2.2. Voice Programming Requirements

This section contains information that is required when using the Voice Library and many of its functions. The following topics are covered:

- Opening and Using Devices (*Section 2.2.1*)
- Opening and Using Voice Channels (*Section 2.2.2*)
- Busy and Idle Device States (*Section 2.2.3*)
- I/O Terminations (*Section 2.2.4*)
- Error Handling (*Section 2.2.5*)
- Voice Library Include Files (*Section 2.2.6*)
- Compiling Applications (*Section 2.2.7*)

2.2.1. Opening and Using Devices

When you open a file under Windows NT, it returns a unique file descriptor for that file. The following is an example of a file descriptor:

```
int file_descriptor;
file_descriptor = open(filename, mode);
```

Any subsequent action you wish to perform on that file is accomplished by identifying the file using **file_descriptor**. No action can be performed on the file until it is first opened.

Dialogic boards and channels work in a similar manner. You must first open a Voice device using **dx_open( )** before you can perform any operation on it. When you open a channel using **dx_open( ),** the value returned is a unique Dialogic device handle for that particular open process on that channel. The Dialogic channel device handle is referred to as **chdev**, where

```
int chdev;
chdev = dx_open(channel_name,mode)
```

Any time you wish to use a Voice library function on the channel, you must identify the channel with its Dialogic channel device handle, **chdev**. The channel name is used only when opening a channel, and all actions after opening must use

the handle **chdev**. Board devices are opened by following the same procedure, where **bddev** refers to the Dialogic board device handle.

NOTE: As stated above, boards and channels are considered separate devices under Windows NT. It is possible to open and use a channel without ever opening the board it is on. There is no board-channel hierarchy imposed by the driver.

To enable users to control the boards and the channels under the Windows NT operating system, Dialogic provides a library of C language functions. For details on opening and closing channels and boards refer to the function references for **dx_open( )** and **dx_close( )** in *Section 3.2. Voice Library Function* Descriptions.

CAUTION

Dialogic devices should **never** be opened using the Windows NT **open( ).**

2.2.2. Opening and Using Voice Files

The Voice library provides a set of standard I/O routines. Although applications may use the routines provided with the *Microsoft C Runtime Library*, Dialogic recommends that the application use the Dialogic file handling routines when manipulating voice files. These routines are **dx_fileopen( )** for opening voice files, **dx_fileclose( )** for closing voice files, and **dx_fileseek( ), dx_fileread( ),** and **dx_filewrite( )** for searching for, reading, or writing directly to a file. The arguments for these functions are identical to the equivalent "C" runtime functions.

2.2.3. Busy and Idle States

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. The categories of functions and their state dependencies are described in the following sections.

2.2.4. I/O Terminations

Pass a set of termination conditions as one of the function parameters when an I/O function is issued. Termination conditions are events monitored during the I/O process that cause an I/O function to terminate. When the termination condition is met, **ATDX_TERMMSK( )** returns the reason for termination. I/O functions can terminate under the following conditions:

- byte transfer count is satisfied
- device has stopped due to **dx_stopch( )**
- end of file is reached during a play
- loop current has dropped for a period of time
- maximum delay between DTMF digits is detected
- maximum number of digits has been received
- maximum period of non-silence (noise or meaningful sound) has been detected
- maximum period of silence has been detected
- pattern of silence and non-silence (noise or meaningful sound) has been detected
- specific digit has been received
- I/O function has been executing for a maximum period of time
- user-defined digit has been received
- user-defined tone-on or tone-off has been detected (GTD)

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

To set the termination conditions, values are placed in fields of a DV_TPT structure. If you set more than one termination condition, the first one that occurs will terminate the I/O function. The DV_TPT structures can be configured as a linked list or array, with each DV_TPT specifying a single terminating condition. The DV_TPT structure, which is defined in *srllib.h*, is described in detail in the *Standard Runtime Library Programmer's Guide for Windows NT.* Voice board

values for the DV_TPT are contained *Appendix A*. The termination conditions are described in the following paragraphs.

Byte Transfer Count - This termination condition applies when playing or recording a file with **dx_play( )** or **dx_rec( ).** The maximum number of bytes is set in the *DX_IOTT*structure. This condition will cause termination if the maximum number of bytes is used before one of the termination conditions specified in the DV_TPT occurs. See *Section 4.1.5. DX_IOTT - I/O transfer table*, for information about setting the number of bytes in the DX_IOTT.

Stop Occurred - dx_stopch( ) terminates any I/O function, except for **dx_dial( )** without Call Analysis enabled, and **dx_wink( ).** See the **dx_stopch( )** function description for more detailed information about this function.

End of File Reached - This termination condition applies when playing a file. This condition causes 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. See *Section 4.1.5. DX_IOTT - I/O transfer table* for information about setting the DX_IOTT. **ATDX_TERMMSK( )** returns the termination reason TM_EOD when this termination condition is met.

Loop Current Drop - In some central offices, switches, and PBX's, 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. Specify the amount of time in the **tp_length** field of a DV_TPT structure in 100 ms units (default) or 10 ms units. Specify 10 ms in the **tp_flags** field of the DV_TPT structure. **ATDX_TERMMSK( )** returns the termination reason TM_LCOFF when this termination condition is met.

Maximum Delay Between Digits - This termination condition monitors the length of time between the digits being received. A specific length of time can be placed in the **tp_length** field of a DV_TPT. If the time between receiving digits is more than this period of time, the function terminates. Specify the amount of time in 100 ms units (default) or 10 ms units for the **tp_length** field or 10 ms units for the **tp_flags** field. **ATDX_TERMMSK( )** returns the termination reason TM_IDDTIME when this termination condition is met.

Maximum Digits Received - This termination condition counts the number of digits in the channel's digit buffer. If the buffer is not empty before the I/O

function is called, the condition counts the digits remaining in the buffer as well. To set the maximum number of digits received before termination, place a number from 1 to 31 in the **tp_length** field of a DV_TPT. **ATDX_TERMMSK( )** returns the termination reason TM_MAXDTMF when this termination condition is met.

Maximum Length of Non-silence - Non-silence is the absence of silence: noise or meaningful sound, such as a person speaking. Enable this condition by setting the **tp_length** field of a DV_TPT to a specific period of time. When the application detects non-silence for this length of time, the I/O function terminates. 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. Specify the amount of time in 100 ms units (default) or 10 ms units in the **tp_length** field or 10 ms units in the **tp_flags** field of the DV_TPT structure**. ATDX_TERMMSK( )** returns the termination reason TM_MAXNOSIL when this termination condition is met.

Maximum Length of Silence - Enable this termination condition 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 ms units (default) or 10 ms units for the **tp_length** field or 10 ms units in the **tp_flags** field of the DV_TPT structure. **ATDX_TERMMSK( )** returns the termination reason TM_MAXSIL when this termination condition is met.

Pattern of Silence and Non-silence - A known pattern of silence and non-silence can terminate a function. A pattern can be specified by specifying 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. **ATDX_TERMMSK( )** returns the termination reason TM_PATTERN when this termination condition is met.

DX_PMOFF/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. A detailed description of how to set these termination conditions is described in *Appendix A* in a section called *"Using DX_PMON and DX_PMOFF."*

Specific Digit Received - An application collects the digits received during an I/O function in a channel's digit buffer. If the buffer is not empty before an I/O function executes, the application treats the digits in the buffer as if the digits were received during the I/O execution. Enable this termination condition 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. **ATDX_TERMMSK( )** returns the termination reason TM_DIGIT when this termination condition is met.

Maximum Function Time - Place a time limit on the I/O function by seeting the **tp_length** field of a DV_TPT to a specific length of time in 100 ms units. The I/O function terminates when it executes longer than this period of time. Specify the amount of time in 100 ms units (default) or 10 ms units for the **tp_length** field and 10 ms units in the **tp_flags** field of the DV_TPT. **ATDX_TERMMSK( )** returns the termination reason TM_MAXTIME when this termination condition is met.

User-Defined Digit Received - An application collects user-defined digits in a channel's digit buffer during an I/O function. If the buffer is not empty before an I/O function executes, the application treats the digits in the buffer as if 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 terminates. **ATDX_TERMMSK( )** returns the termination reason TM_DIGIT when this termination condition is met.

User-Defined Tone On/Off Event Detected - Use this termination condition with Global Tone Detection. Before specifying a user-defined tone as a termination condition, define the tone using the GTD **dx_bld...( )** functions, and enable the tone detection on the channel using the **dx_addtone( )** or **dx_enbtone( )** functions. 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. **ATDX_TERMMSK( )** returns the termination reason TM_TONE when this termination condition is met.

The application may clear the DV_TPT structure using **dx_clrtpt( )** before initializing the structure and passing a pointer to it as a function parameter.

Refer to the *Standard Runtime Library Programmer's Guide for Windows NT* for a complete discussion of the DV_TPT structure, and to *Appendix A* for a description of Voice software values for the DV_TPT.

2.2.5. Error Handling

All the Dialogic Voice Library functions return a value to indicate success or failure of the function. All Voice Library functions indicate success by a return value of zero or a non-negative number.

Extended Attribute functions that return pointers return a pointer to the ASCIIZ string "Unknown device" if they fail.

Extended Attribute functions that don't return pointers, return a value of AT_FAILURE if they fail.

All other functions return a value of -1 to indicate a failure.

If a function fails, call the Standard Attribute functions **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** for the reason for failure. These functions are described in the *Standard Runtime Library Programmer's Guide for Windows NT.*

The errors that can be returned by a Voice Library function are listed in Table 1. These errors are also listed for reference in *Appendix B*.

NOTES:

1. **dx_open( )** and **dx_close( )** are exceptions to the above error handling rules. If these functions fail, the return code is -1 and the specific error is found in the **errno** variable contained in *errno.h.*
2. If **ATDV_LASTERR( )** returns the error EDX_SYSTEM, a Windows NT system error has occurred. Check the global variable errno contained in *errno.h.*

Table 1. Voice Library Function Errors

Error Define	Error String
EDX_AMPLGEN	Invalid Amplitude Value in Tone Generation Template [+2dB - -40dB]

Error Define	Error String
EDX_ASCII	Invalid ASCII Value in Tone Template Description
EDX_BADDEV	Invalid Device Descriptor
EDX_BADIOTT	Invalid Entry in the *DX_IOTT*
EDX_BADPARM	Invalid Parameter in Function Call
EDX_BADPROD	Function Not Supported on this Board
EDX_BADTPT	Invalid Entry in the DX_TPT
EDX_BADWAVFILE	Invalid WAV file
EDX_BUSY	Device is Already Busy
EDX_CADENCE	Invalid Cadence Component Values in Tone Template Description
EDX_CHANNUM	Invalid Channel Number Specified
EDX_DIGTYPE	Invalid Dig_type Value in Tone Template Description
EDX_FLAGGEN	Invalid tn_dflag field in Tone Generation Template
EDX_FREQDET	Invalid Freq Component Values in Tone Template Description
EDX_FREQGEN	Invalid Frequency Component in Tone Generation Template [200hz - 2000hz]
EDX_FWERROR	Firmware Error
EDX_IDLE	Device is Idle
EDX_INVSUBCM	Invalid Sub Command Number
EDX_MAXTMPLT	Max number of Templates Exists
EDX_MSGSTATUS	Invalid Message Status Setting

Error Define	Error String
EDX_NOERROR	No Errors
EDX_NONZEROSIZE	Reset to Default was Requested but size was non-zero
EDX_SH_BADCMD	Unsupported command or WAV file format
EDX_SPDVOL	Must Specify either SV_SPEEDTBL or SV_VOLUMETBL
EDX_SVADJBLKS	Invalid Number of Speed/Volume Adjustment Blocks
EDX_SVMTRANGE	An Entry in *DX_SVMT* was out of Range
EDX_SVMTSIZE	Invalid Table Size Specified
EDX_SYSTEM	System Error
EDX_TIMEOUT	Function Timed Out
EDX_TONEID	Bad Tone Template ID

2.2.6. Voice Library Include Files

The following lines must be included in application code prior to calling Voice Library functions:

```
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
```

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

The libraries are located in the following default directories:

<*install drive:*>\<*install directory*> *dialogic*\ *inc*

2.2.7. Compiling Applications

Application programs developed using the Voice Library for Windows NT should be linked with the following libraries.

Libraries for multithreaded applications are located in the following default directories:

> *<install drive:>\<install directory> \dialogic\i386\lib\libdxxmt.*lib - the main Voice Library
> *<install drive:>\<install directory> \dialogic\i386\lib\libsrlmt.lib* - the Standard Runtime Library

They should be linked in the order specified above.

3. Voice Function Reference

3.1. Voice Function Reference Overview

This chapter provides a description of the Voice Library functions provided with the voice software for Windows NT systems. These functions are listed alphabetically for ease of use.

Some Voice Library functions use special structures. These structures are defined in the *dxxxlib.h* and *srllib.h* header files and are described in *Chapter 4. Voice Data Structures and Device Parameters*. They include the following:

DV_DIGIT	• User Digit Buffer Structure
DV_TPT	• Termination Parameter Table Structure
DX_CAP	• Call Analysis Parameter Structure
DX_EBLK	• Event Block Structure
DX_IOTT	• I/O Transfer Table Structure
DX_UIO	• I/O User-definable I/O Structure

The **dx_getparm( )** and **dx_setparm( )** functions use defined masks to specify parameters. The definitions of the masks are found in *Chapter 4. Voice Data Structures and Device Parameters*.

Applications that use the Voice Library must include the *dxxxlib.h* and *srllib.h* header files.

NOTE: The *srllib.h* header file must always be listed before any other Dialogic header file.

SCbus Functions

See the *SCbus Routing Function Reference for Windows NT* for function descriptions and the nomenclature used to identify devices, channels and time slots in an SCbus configuration. The SCbus routing functions can only be used in SCbus configurations.

3.2. Voice Library Function Descriptions

Name:	long ATDX_ANSRSIZ(chdev)
Inputs:	int chdev • valid Dialogic channel device handle
Returns:	answer duration in 10 ms units if successful AT_FAILURE if error
Includes:	srllib.h dxxxlib.h
Category:	Extended Attribute

■ Description

The **ATDX_ANSRSIZ( )** function returns the duration of the answer that occurs when **dx_dial( )** with Basic Call Analysis enabled is called on a channel. An answer is considered the period of non-silence that begins after cadence is broken and a connection is made. This measurement is taken before a connect event is returned. The duration of the answer can be used to determine if the call was answered by a person or an answering machine. This feature is based on the assumption that an answering machine typically answers a call with a longer greeting than a live person does.

See the section called "*Cadence Detection Parameters' Affect on a Connect*" in the *Voice Features Guide for Windows NT* for information about distinguishing between a person and answering machine.

The function parameter is defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.

■ Cautions

None.

■ Example

```
/* Call Analysis with user-specified parameters */
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
```

```
#include <windows.h>

main()
{
   int cares, chdev;
   DX_CAP capp;
   .
   .
   /* open the channel using dx_open( ). Obtain channel device descriptor in
    * chdev
    */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1)  {
     /* process error */
   }

   /* take the phone off-hook */
   if (dx_sethook(chdev,DX_OFFHOOK,EV_SYNC) == -1) {
     /* process error */
   }

   /* Set the DX_CAP structure as needed for call analysis. Perform the
    * outbound dial with call analysis enabled
    */
   if ((cares = dx_dial(chdev,"5551212",&capp,DX_CALLP|EV_SYNC)) == -1) {
     /* perform error routine */
   }

   switch (cares) {
   case CR_CNCT:     /* Call Connected, get some additional info */
      printf("\nDuration of short low - %ld ms",ATDX_SHORTLOW(chdev)*10);
      printf("\nDuration of long low  - %ld ms",ATDX_LONGLOW(chdev)*10);
      printf("\nDuration of answer    - %ld ms",ATDX_ANSRSIZ(chdev)*10);
      break;
   case CR_CEPT:     /* Operator Intercept detected */
      printf("\nFrequency detected - %ld Hz",ATDX_FRQHZ(chdev));
      printf("\n%% of Frequency out of bounds - %ld Hz",ATDX_FRQOUT(chdev));
      break;
   case CR_BUSY:
   .
   .
   }
}
```

■ Errors

This function will fail and return AT_FAILURE if an invalid channel device handle is specified in **chdev**.

■ See Also

Related to Call Analysis:

- **dx_dial( )**
- DX_CAP. (*Chapter 4. Voice Data Structures and Device Parameters*)
- "Call Analysis" (*Voice Features Guide for Windows NT*)

Name:	char * ATDX_BDNAMEP(chdev)
Inputs:	int chdev • valid Dialogic channel device handle
Returns:	pointer to Board device name string if successful pointer to ASCIIZ string "Unknown device" if error
Includes:	srllib.h dxxxlib.h
Category:	Extended Attribute

■ Description

The **ATDX_BDNAMEP( )** function returns a pointer to the name of the board device on which the channel accessed by **chdev** resides.

As illustrated in the example, this may be used to open the board device that corresponds to a particular channel device prior to setting board parameters.

The function parameter is defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.

■ Cautions

None.

■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   int chdev, bddev;
   char *bdnamep;
   .
   .
   /* Open the channel device */
   if ((chdev = dx_open("dxxxB1C1", NULL)) == -1) {
     /* Process error */
   }
```

```
   /* Display board name */
   bdnamep = ATDX_BDNAMEP(chdev);
   printf("The board device is: %s\n", bdnamep);


   /* Open the board device */
   if ((bddev = dx_open(bdnamep, NULL)) == -1) {
     /* Process error */
   }
  .
  .
}
```

■ Errors

This function will fail and return a pointer to "Unknown device" if an invalid channel device handle is specified in **chdev**.

Name: long ATDX_BDTYPE(dev)
Inputs: int dev • valid Dialogic board or channel device handle
Returns: board or channel device type if successful
AT_FAILURE if error
Includes: srllib.h
dxxxlib.h
Category: Extended Attribute

■ Description

The **ATDX_BDTYPE( )** function returns the device type of the board or channel **dev**.

A typical use would be to determine whether or not the device can support particular features, such as Call Analysis.

The function parameter is defined as follows:

Parameter	Description
dev:	specifies the valid Dialogic device handle obtained when a board or channel was opened using **dx_open( )**.

Possible return values are the following:

DI_D20BD	• D/20 Board Device
DI_D21BD	• D/21 Board Device
DI_D40BD	• D/40 Board Device
DI_D41BD	• D/41 Board Device
DI_D20CH	• D/20 Channel Device
DI_D21CH	• D/21 Channel Device
DI_D40CH	• D/40 Channel Device
DI_D41CH	• D/41 Channel Device

NOTE: DI_41BD and DI_41CH will be returned for the D/121 board, which emulates three D/41 boards. DI_41BD and DI_41CH will be returned: for the D/160SC-LS board which emulates four D/41 boards; for the D/240SC and D/240SC-T1 boards which emulate six D/41 boards; and for the D/300SC-E1 and D/320SC boards which emulate eight D/41

boards

■ .Cautions

None.

■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
#define ON 1

main()
{
   int bddev;
   long bdtype;
   int call_analysis=0;


   /* Open the board device */
   if ((bddev = dx_open("dxxxB1",NULL)) == -1) {
     /* Process error */
   }


   if((bdtype = ATDX_BDTYPE(bddev)) == AT_FAILURE) {
     /* Process error */
   }


   if(bdtype == DI_D41BD) {
      printf("Device is a D/41 Board\n");
      call_analysis = ON;
   }
  .
  .
}
```

■ Errors

This function will fail and return AT_FAILURE if an invalid board or channel device handle is specified in **dev**.

Name:	long ATDX_BUFDIGS(chdev)
Inputs:	int chdev • valid Dialogic channel device handle
Returns:	number of uncollected digits in the firmware buffer if successful AT_FAILURE if error
Includes:	srllib.h dxxxlib.h
Category:	Extended Attribute

■ Description

The **ATDX_BUFDIGS( )** function returns the number of uncollected digits in the firmware buffer for channel **chdev**. This is the number of digits that have arrived since the last call to **dx_getdig( )** or the last time the buffer was cleared using **dx_clrdigbuf( )**. The digit buffer contains a maximum of 31 digits and a null terminator.

The function parameter is defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.

■ Cautions

Digits that adjust speed and volume (see **dx_setsvcond( )**) will not be passed to the digit buffer.

■ Example

```
#include <fcntl.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   int chdev;
   long bufdigs;
   DX_IOTT iott;
   DV_TPT  tpt[2];
```

```
   /* Open the device using dx_open( ). Get channel device descriptor in
    * chdev. */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
     /* process error */
   }

   /* set up DX_IOTT */
   iott.io_type = IO_DEV|IO_EOT;
   iott.io_bufp = 0;
   iott.io_offset = 0;
   iott.io_length = -1;  /* play till end of file */

   if((iott.io_fhandle = dx_fileopen("prompt.vox", O_RDONLY)) == -1)  {
     /* process error */
   }

   /* set up DV_TPT */
   dx_clrtpt(tpt,2);
   tpt[0].tp_type   = IO_CONT;
   tpt[0].tp_termno = DX_MAXDTMF;     /* Maximum digits */
   tpt[0].tp_length = 4;              /* terminate on 4 digits */
   tpt[0].tp_flags  = TF_MAXDTMF;     /* Use the default flags */
   tpt[1].tp_type   = IO_EOT;
   tpt[1].tp_termno = DX_DIGMASK;     /* Digit termination */
   tpt[1].tp_length = DM_5;           /* terminate on the digit "5" */
   tpt[1].tp_flags  = TF_DIGMASK;     /* Use the default flags */

   /* Play a voice file. Terminate on receiving 4 digits, the digit "5" or
    * at end of file.*/
   if (dx_play(chdev,&iott,tpt,EV_SYNC) == -1) {
     /* process error */
   }
   /* Check # of digits collected and continue processing. */
   if((bufdigs=ATDX_BUFDIGS(chdev))==AT_FAILURE)  {
     /* process error */
   }
   .
   .
   .
}
```

■ Errors

This function will fail and return AT_FAILURE if an invalid channel device handle is specified in **chdev**.

■ See Also

Other digit functions:

- **dx_getdig( )**
- **dx_clrdigbuf( )**

Name: char ** ATDX_CHNAMES(bddev)
Inputs: int bddev • valid Dialogic board device handle
Returns: pointer to array of channel names if successful
pointer to array of pointers that point to "Unknown device" if error
Includes: srllib.h
dxxxlib.h
Category: Extended Attribute

■ Description

The **ATDX_CHNAMES( )** function returns a pointer to an array of channel names associated with the designated board device handle **bddev**.

A possible use for this attribute would be to display the names of the channel devices associated with a particular board device.

The function parameter is defined as follows:

Parameter	Description
bddev:	specifies the valid board device handle obtained when the board was opened using **dx_open( )**.

■ Cautions

None.

■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   int bddev, cnt;
   char **chnames;
   long subdevs;
   .
   .
   /* Open the board device */
```

```
   if ((bddev = dx_open("dxxxB1",NULL)) == -1) {
     /* Process error */
   }
   .
   .
   /* Display channels on board */
   chnames = ATDX_CHNAMES(bddev);
   subdevs = ATDV_SUBDEVS(bddev);  /* number of sub-devices on board */


   printf("Channels on this board are:\n");
   for(cnt=0; cnt<subdevs; cnt++)  {
      printf("%s\n",*(chnames + cnt));
   }


   /* Call dx_open( ) to open each of the
    * channels and store the device descriptors
    */
   .
   .
}
```

■ Errors

This function will fail and return the address of a pointer to "Unknown device" if an invalid board device handle is specified in **bddev**.

Name:	long ATDX_CHNUM(chdev)
Inputs:	int chdev • valid Dialogic channel device handle
Returns:	channel number if successful AT_FAILURE if error
Includes:	srllib.h dxxxlib.h
Category:	Extended Attribute

■ Description

The **ATDX_CHNUM( )** function returns the channel number of the channel **chdev** on the Voice board. Channel numbering starts at 1.

For example, use the channel as an index into an array of channel-specific information.

■ Cautions

None.

■ Example

```
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
main()
{
   int chdev;
   long chno;
   .
   .
   /* Open the channel device */
   if ((chdev = dx_open("dxxxB1C1", NULL)) == -1) {
     /* Process error */
   }
   /* Get Channel number */
   if((chno = ATDX_CHNUM(chdev)) == AT_FAILURE) {
     /* Process error */
   }
   /* Use chno for application-specific purposes */
   .
   .
}
```

■ Errors

This function will fail and return AT_FAILURE if an invalid channel device handle is specified in **chdev**.

Name:	long ATDX_CONNTYPE(chdev)
Inputs:	int chdev • valid Dialogic channel device handle
Returns:	connection type AT_FAILURE if error
Includes:	srllib.h dxxxlib.h
Category:	Extended Attribute

■ Description

The **ATDX_CONNTYPE( )** function returns the connection for a call on the channel chdev. Use this function when a CR_CNCT is returned by **ATDX_CPTERM( )** after termination of **dx_dial( )** with Call Analysis enabled.

Possible return values are the following:

CON_CAD	• Connection due to cadence break
CON_LPC	• Connection due to loop current
CON_PVD	• Connection due to Positive Voice Detection
CON_PAMD	• Connection due to Positive Answering Machine Detection

The function parameter is defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.

■ Cautions

None.

■ Example

```
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
```

```
main()
{
   int  dxxxdev;
   int  cares;


   /*
    * Open the Voice Channel Device and Enable a Handler
    */
   if ( ( dxxxdev = dx_open( "dxxxB1C1", NULL) ) == -1 ) {
      perror( "dxxxB1C1" );
      exit( 1 );
   }

/*
    *  Delete any previous tones
    */
   if ( dx_deltones(dxxxdev) < 0 ) {
      /* handle error */
   }


   /*
    * Now enable Enhanced call progress with above changed settings.
    */
   if (dx_initcallp( dxxxdev )) {
      /* handle error */
   }


   /*
    * Take the phone off-hook
    */
   if ( dx_sethook( dxxxdev, DX_OFFHOOK, EV_SYNC ) == -1 ) {
      printf( "Unable to set the phone off-hook\n" );
      printf( "Lasterror = %d  Err Msg = %s\n",
         ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
      dx_close( dxxxdev );
      exit( 1 );
   }


   /*
    * Perform an outbound dial with call analysis, using
    * the default call analysis parameters.
    */
   if ((cares=dx_dial( dxxxdev, ",84",(DX_CAP *)NULL, DX_CALLP ) ) == -1 ) {
      printf( "Outbound dial failed - reason = %d\n",
          ATDX_CPERROR( dxxxdev ) );
      dx_close( dxxxdev );
      exit( 1 );
   }


   printf( "Call Analysis returned %d\n", cares );
   if ( cares == CR_CNCT ) {
      switch ( ATDX_CONNTYPE( dxxxdev ) ) {
      case CON_CAD:
         printf( "Cadence Break\n" );
         break;
      case CON_LPC:
         printf( "Loop Current Drop\n" );
         break;


      case CON_PVD:
         printf( "Positive Voice Detection\n" );
```

```
         break;

      case CON_PAMD:
         printf( "Positive Answering Machine Detection\n" );
         break;

      default:
         printf( "Unknown connection type\n" );
         break;
      }
   }

   /*
    * Continue Processing
    *   .
    *   .
    *   .
    */

   /*
    * Close the opened Voice Channel Device
    */
   if ( dx_close( dxxxdev ) != 0 ) {
      perror( "close" );
   }

   /* Terminate the Program */
   exit( 0 );
}
```

■ Errors

This function will fail and return AT_FAILURE if an invalid channel device handle is specified in **chdev**.

■ See Also

Related to Call Analysis:

- **dx_dial( )**
- **ATDX_CPTERM( )**
- DX_CAP structure (*Chapter 4. Voice Data Structures and Device Parameters*)
- "Call Analysis" (*Voice Features Guide for Windows NT*)

Name:	long ATDX_CPERROR(chdev)	
Inputs:	int chdev	• valid Dialogic channel device handle
Returns:	Call Analysis error AT_FAILURE if function fails	
Includes:	srllib.h dxxxlib.h	
Category:	Extended Attribute	

■ Description

The **ATDX_CPERROR( )** function returns the error that caused **dx_dial( )** to terminate when checking for operator intercept SIT tones.

When **dx_dial( )** terminates due to a Call Analysis error, CR_ERROR is returned by **ATDX_CPTERM( )**.

If CR_ERROR is returned, use **ATDX_CPERROR( )** to determine the Call Analysis error. One of the following values will be returned:

CR_LGTUERR	• lower frequency greater than upper frequency
CR_MEMERR	• out of memory when creating temporary SIT tone templates
CR_MXFRQERR	• invalid ca_maxtimefrq field in DX_CAP
CR_OVRLPERR	• overlap in selected SIT tones
CR_TMOUTOFF	• timeout waiting for SIT tone to terminate
CR_TMOUTON	• timeout waiting for SIT tone to commence
CR_UNEXPTN	• unexpected SIT tone
CR_UPFRQERR	• invalid upper frequency selection

The function parameter is defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.

■ Cautions

None.

■ Example

```
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   int  dxxxdev;
   int  cares;

   /*
    * Open the Voice Channel Device and Enable a Handler
    */
   if ( ( dxxxdev = dx_open( "dxxxB1C1", NULL) ) == -1 ) {
      perror( "dxxxB1C1" );
      exit( 1 );
   }

   /*
    * Take the phone off-hook
    */
   if ( dx_sethook( dxxxdev, DX_OFFHOOK, EV_SYNC ) == -1 ) {
      printf( "Unable to set the phone off-hook\n" );
      printf( "Lasterror = %d  Err Msg = %s\n",
          ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
      dx_close( dxxxdev );
      exit( 1 );
   }

   /*
    * Perform an outbound dial with call analysis, using
    * the default call analysis parameters.
    */
   if((cares = dx_dial( dxxxdev,",84",(DX_CAP *) NULL, DX_CALLP )) == -1 ) {
      printf( "Outbound dial failed - reason = %d\n",
          ATDX_CPERROR( dxxxdev ) );
      dx_close( dxxxdev );
      exit( 1 );
   }

   /*
```

```
    * Continue Processing
    *   .
    *   .
    *   .
    */


   /*
    * Close the opened Voice Channel Device
    */
   if ( dx_close( dxxxdev ) != 0 ) {
      perror( "close" );
   }

   /* Terminate the Program */
   exit( 0 );
}
```

See Also

Related to Call Analysis:

- **dx_dial( )**
- **ATDX_CPTERM( )**
- DX_CAP structure (*Chapter 4. Voice Data Structures and Device Parameters*)

Name: long ATDX_CPTERM(chdev)
Inputs: int chdev • valid Dialogic channel device handle
Returns: last Call Analysis termination if successful
AT_FAILURE if error
Includes: srllib.h
dxxxlib.h
Category: Extended Attribute

■ Description

The **ATDX_CPTERM( )** function returns last Call Analysis termination on the channel **chdev**. Call this function to determine the call status after dialing out with Call Analysis enabled.

The function parameter is defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.

Possible return values are the following:

- CR_BUSY — Busy
- CR_CEPT — Operator intercept
- CR_CNCT — Connect
- CR_FAXTONE — Called line answered by fax machine or modem
- CR_NOANS — No answer
- CR_NODIALTONE — Timeout occurred while waiting for dial tone
- CR_NORB — No ringback
- CR_STOPD — Stopped
- CR_ERROR — Error

■ Cautions

None.

Example

```
 /* Call Analysis with user-specified parameters */
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   int chdev;
   DX_CAP capp;
   .
   .
   /* open the channel using dx_open( ).  Obtain channel device descriptor
      in
    * chdev
    */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1)  {
     /* process error */
   }

   /* take the phone off-hook */
   if (dx_sethook(chdev,DX_OFFHOOK,EV_SYNC) == -1) {
     /* process error */
   } else {

      /* Clear DX_CAP structure */
      dx_clrcap(&capp);

      /* Set the DX_CAP structure as needed for call analysis.
       * Allow 3 rings before no answer.
       */
      capp.ca_nbrdna = 3;

      /* Perform the outbound dial with call analysis enabled. */
      if (dx_dial(chdev,"5551212",&capp,DX_CALLP|EV_SYNC) == -1) {
      /* perform error routine */
      }
   }
   .
   .

   /* Examine last call progress termination on the device */
   switch (ATDX_CPTERM(chdev)) {
   case CR_CNCT:    /* Call Connected, get some additional info */
   .
   .
      break;
   case CR_CEPT:    /* Operator Intercept detected */
   .
   .
      break;
      .
      .
   case AT_FAILURE:   /* Error */

   }
}
```

Errors

This function will fail and return AT_FAILURE if an invalid channel device handle is specified in **chdev**.

See Also

Related to Call Analysis:

- **dx_dial( )**
- DX_CAP structure (*Chapter 4. Voice Data Structures and Device Parameters*)
- "Call Analysis" (*Voice Features Guide for Windows NT*)

Name: long ATDX_CRTNID(chdev)
Inputs: int chdev • valid channel device handle
Returns: identifier of the tone that caused the most recent Call Analysis termination, if successful
AT_FAILURE if error
Includes: srllib.h
dxxxlib.h
Category: Extended Attribute

■ Description

The **ATDX_CRTNID( )** function returns the tone identifier of the tone that caused the most recent Call Analysis termination of the channel device. This function is supported under PerfectCall Call Analysis on DSP boards only. See the *Voice Features Guide for Windows NT* for a description of PerfectCall Call Analysis.

The function parameter is defined as follows:

Parameter	Description
chdev:	specifies the valid Dialogic device handle obtained when a board or channel was opened using **dx_open( )**.

Possible return values are the following:

- TID_DIAL_LCL • Local dial tone
- TID_DIAL_INTL • International dial tone
- TID_DIAL_XTRA • Special (“Extra”) dial tone
- TID_BUSY1 • First signal busy
- TID_BUSY2 • Second signal busy
- TID_RINGBK1 • Ringback
- TID_FAX1 • First fax or modem tone
- TID_FAX2 • Second fax or modem tone

■ Cautions

None.

Example

```
#include <stdio.h>

#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   DX_CAP  cap_s;
   int             ddd, car;
   char   *chnam, *dialstrg;

   chnam    = "dxxxB1C1";
   dialstrg = "L1234";

   /*
    *  Open channel
    */
   if ((ddd = dx_open( chnam, NULL )) == -1 ) {
      /* handle error */
   }

   /*
    *  Delete any previous tones
    */
   if ( dx_deltones(ddd) < 0 ) {
      /* handle error */
   }

   /*
    * Now enable Enhanced call progress with above changed settings.
    */
   if (dx_initcallp( ddd )) {
      /* handle error */
   }

   /*
    *  Set off Hook
    */
   if ((dx_sethook( ddd, DX_OFFHOOK, EV_SYNC )) == -1) {
      /* handle error */
   }

   /*
    *  Dial
    */
   printf("Dialing %s\n", dialstrg );
   car = dx_dial(ddd,dialstrg,(DX_CAP *)&cap_s,DX_CALLP|EV_SYNC);
   if (car == -1) {
      /* handle error */
   }

   switch( car ) {
   case CR_NODIALTONE:
      switch( ATDX_DTNFAIL(ddd) ) {
      case 'L':
         printf(" Unable to get Local dial tone\n");
```

```
         break;
      case 'I':
         printf(" Unable to get International dial tone\n");
         break;
      case 'X':
         printf(" Unable to get special eXtra dial tone\n");
         break;
      }
      break;


   case CR_BUSY:
      printf(" %s engaged - %s detected\n", dialstrg,
            (ATDX_CRTNID(ddd) == TID_BUSY1 ? "Busy 1" : "Busy 2") );
      break;


   case CR_CNCT:
      printf(" Successful connection to %s\n", dialstrg );
      break;


   default:
      break;
   }


   /*
    *  Set on Hook
    */
   if ((dx_sethook( ddd, DX_ONHOOK, EV_SYNC )) == -1) {
      /* handle error */
   }

   dx_close( ddd );
}
```

Name: long ATDX_DEVTYPE(dev)
Inputs: int dev • valid Dialogic board or channel device handle
Returns: device type if successful
AT_FAILURE if error
Includes: srllib.h
dxxxlib.h
Category: Extended Attribute

■ Description

The **ATDX_DEVTYPE( )** function returns device type of the board or channel **dev**.

The function parameter is defined as follows:

Parameter	Description
dev:	specifies the valid Dialogic device handle obtained when a board or channel was opened using **dx_open( )**.

Possible return values are the following:

DT_DXBD	• Board device
DT_DXCH	• Channel

■ Cautions

None.

■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   int bddev;
```

```
   long devtype;

   /* Open the board device */
   if ((bddev = dx_open("dxxxB1",NULL)) == -1) {
     /* Process error */
   }

   if((devtype = ATDX_DEVTYPE(bddev)) == AT_FAILURE) {
     /* Process error */
   }

   if(devtype == DT_DXBD) {
      printf("Device is a Board\n");
   }

   /* Continue processing */
      .
      .
}
```

■ Errors

This function will fail and return AT_FAILURE if an invalid board or channel device handle is specified in **dev**.

Name:	long ATDX_DTNFAIL(chdev)
Inputs:	int chdev • valid channel device handle
Returns:	code for the dial tone that failed to appear AT_FAILURE if error
Includes:	srllib.h dxxxlib.h
Category:	Extended Attribute

■ Description

The **ATDX_DTNFAIL( )** function returns character for dial tone that PerfectCall Call Analysis failed to detect. This attribute is supported under PerfectCall Call Analysis only.

The function parameter is defined as follows:

Parameter	Description
chdev:	specifies the valid Dialogic device handle obtained when a board or channel was opened using **dx_open( )**.

Possible return values are the following:

L	• Local dial tone
I	• International dial tone
X	• Special ("extra") dial tone

■ Cautions

None.

■ Example

```
#include <stdio.h>

#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
```

```
{
   DX_CAP  cap_s;
   int    ddd, car;
   char   *chnam, *dialstrg;


   chnam    = "dxxxB1C1";
   dialstrg = "L1234";


   /*
    *  Open channel
    */
   if ((ddd = dx_open( chnam, NULL )) == -1 ) {
      /* handle error */
   }


   /*
    *  Delete any previous tones
    */
   if ( dx_deltones(ddd) < 0 ) {
      /* handle error */
   }


   /*
    * Now enable Enhanced call progress with above changed settings.
    */
   if (dx_initcallp( ddd )) {
      /* handle error */
   }


   /*
    *  Set off Hook
    */
   if ((dx_sethook( ddd, DX_OFFHOOK, EV_SYNC )) == -1) {
      /* handle error */
   }

   /*
    *  Dial
    */

   printf("Dialing %s\n", dialstrg );
   car = dx_dial(ddd,dialstrg,(DX_CAP *)&cap_s,DX_CALLP|EV_SYNC);
   if (car == -1) {
      /* handle error */
   }


   switch( car ) {
   case CR_NODIALTONE:
      switch( ATDX_DTNFAIL(ddd) ) {
      case 'L':
         printf(" Unable to get Local dial tone\n");
         break;
      case 'I':
         printf(" Unable to get International dial tone\n");
         break;
      case 'X':
         printf(" Unable to get special eXtra dial tone\n");
         break;
      }
      break;
```

```
   case CR_BUSY:
      printf(" %s engaged - %s detected\n", dialstrg,
          ATDX_CRTNID(ddd) == TID_BUSY1 ? "Busy 1" : "Busy 2") );
      break;


   case CR_CNCT:
      printf(" Successful connection to %s\n", dialstrg );
      break;


   default:
      break;
   }


   /*
    *  Set on Hook
    */
   if ((dx_sethook( ddd, DX_ONHOOK, EV_SYNC )) == -1) {
      /* handle error */
   }

   dx_close( ddd );
}
```

Name: long ATDX_FRQDUR(chdev)
Inputs: int chdev • valid Dialogic channel device handle
Returns: first frequency duration in 10ms units
AT_FAILURE if error
Includes: srllib.h
dxxxlib.h
Category: Extended Attribute

■ Description

When **dx_dial( )** terminates due to Operator Intercept; the **ATDX_FRQDUR( )** function can be used to return the duration of the first detected SIT tone frequency in 10 ms units.

Termination due to Operator Intercept is indicated by **ATDX_CPTERM( )** returning CR_CEPT.

The function parameter is defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.

■ Cautions

None.

■ Example

This example illustrates **ATDX_FRQDUR( )**, **ATDX_FRQDUR2( )**, and **ATDX_FRQDUR3( )**.

```
/* Call Analysis with user-specified parameters */
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
```

```
{
   int cares, chdev;
   DX_CAP capp;
   .
   .
   /* open the channel using dx_open( ). Obtain channel device descriptor in
    * chdev
    */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1)  {
     /* process error */
   }

   /* take the phone off-hook */
   if (dx_sethook(chdev,DX_OFFHOOK,EV_SYNC) == -1) {
     /* process error */
   }

   /* Set the DX_CAP structure as needed for call analysis. Perform the
    * outbound dial with call analysis enabled
    */
   if ((cares = dx_dial(chdev,"5551212",&capp,DX_CALLP|EV_SYNC)) == -1) {
     /* perform error routine */
   }

   switch (cares) {
     case CR_CNCT:  /* Call Connected, get some additional info */
      printf("\nDuration of short low - %ld ms",ATDX_SHORTLOW(chdev)*10);
      printf("\nDuration of long low  - %ld ms",ATDX_LONGLOW(chdev)*10);
      printf("\nDuration of answer    - %ld ms",ATDX_ANSRSIZ(chdev)*10);
      break;
     case CR_CEPT:  /* Operator Intercept detected */
      printf("\nFirst frequency detected - %ld Hz",ATDX_FRQHZ(chdev));
      printf("\nSecond frequency detected - %ld Hz", ATDX_FRQHZ2(chdev));
      printf("\nThird frequency detected - %ld Hz", ATDX_FRQHZ3(chdev));

      printf("\nDuration of first frequency - %ld ms", ATDX_FRQDUR(chdev));
      printf("\nDuration of second frequency - %ld ms",
             ATDX_FRQDUR2(chdev));
      printf("\nDuration of third frequency - %ld ms", ATDX_FRQDUR3(chdev));
      break;
     case CR_BUSY:
      break;
      .
      .
   }
}
```

■ See Also

Related to Call Analysis:

- **dx_dial( )**
- **ATDX_CPTERM( )**
- DX_CAP structure (*Chapter 4. Voice Data Structures and Device Parameters*)
- "Call Analysis" (*Voice Features Guide for Windows NT*)

SIT Tone Detection - All Boards:

- **ATDX_FRQHZ( )**

SIT Tone Detection - DSP Boards

- **ATDX_FRQDUR2( )**
- **ATDX_FRQDUR3( )**
- **ATDX_FRQHZ2( )**
- **ATDX_FRQHZ3( )**

Name:	long ATDX_FRQDUR2(chdev)
Inputs:	int chdev • valid Dialogic channel device handle
Returns:	second frequency duration in 10 ms units AT_FAILURE if error
Includes:	srllib.h dxxxlib.h
Category:	Extended Attribute

■ Description

When **dx_dial( )** terminates due to Operator Intercept, the **ATDX_FRQDUR2( )** function can be used to return the duration of the second detected frequency in 10 ms units.

NOTE: For more information on tri-tone SIT sequences, see *Frequency Detection* in the *Voice Features Guide for Windows NT*.

Termination due to Operator Intercept is indicated by **ATDX_CPTERM( )** returning CR_CEPT.

The function parameter is defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.

■ Cautions

None.

■ Example

See the example for **ATDX_FRQDUR( )**.

■ See Also

Related to Call Analysis:

- **dx_dial( )**
- **ATDX_CPTERM( )**
- DX_CAP structure (*Chapter 4. Voice Data Structures and Device Parameters*)
- "Call Analysis" (*Voice Features Guide for Windows NT*)

SIT Tone Detection - All Boards:

- **ATDX_FRQHZ( )**
- **ATDX_FRQDUR( )**
- "Frequency Detection" (*Voice Features Guide for Windows NT*)

SIT Tone Detection - DSP Boards:

- **ATDX_FRQDUR3( )**
- **ATDX_FRQHZ2( )**
- **ATDX_FRQHZ3( )**

Name:	long ATDX_FRQDUR3(chdev)
Inputs:	int chdev • valid Dialogic channel device handle
Returns:	third frequency duration in 10 ms units AT_FAILURE if error
Includes:	srllib.h dxxxlib.h
Category:	Extended Attribute

■ Description

When **dx_dial( )** terminates due to Operator Intercept, the **ATDX_FRQDUR3( )** function can be used to return the duration of the third detected frequency in 10 ms units.

NOTE: For more information about tri-tone SIT tone detection, see *Frequency Detection* in the *Voice Features Guide for Windows NT*.

Termination due to Operator Intercept is indicated by **ATDX_CPTERM( )** returning CR_CEPT.

The function parameter is defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.

■ Cautions

None.

■ Example

See the example for **ATDX_FRQDUR( )**.

■ See Also

Related to Call Analysis:

- **dx_dial( )**
- **ATDX_CPTERM( )**
- DX_CAP structure (*Chapter 4. Voice Data Structures and Device Parameters*)
- "Call Analysis" (*Voice Features Guide for Windows NT*)

SIT Tone Detection - All Boards:

- **ATDX_FRQHZ( )**
- **ATDX_FRQDUR( )**
- "Frequency Detection" (*Voice Features Guide for Windows NT*)

SIT Tone Detection - DSP Boards

- **ATDX_FRQDUR2( )**
- **ATDX_FRQHZ2( )**
- **ATDX_FRQHZ3( )**

Name: long ATDX_FRQHZ(chdev)
Inputs: int chdev • valid Dialogic channel device handle
Returns: first tone frequency in hz
AT_FAILURE if error
Includes: srllib.h
dxxxlib.h
Category: Extended Attribute

■ Description

When **dx_dial( )** with Call Analysis terminates due to Operator Intercept, the **ATDX_FRQHZ( )** function can return frequency of answered signal in Hz, (such as the first detected SIT tone that occurs due to operator intercept).

NOTE: Termination due to Operator Intercept is indicated by **ATDX_CPTERM( )** returning CR_CEPT.

The function parameter is defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.

■ Cautions

None.

■ Example

This example illustrates the use of **ATDX_FRQHZ( )**, **ATDX_FRQHZ2( )**, and **ATDX_FRQHZ3( )**.

```
/* Call Analysis with user-specified parameters */
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   int cares, chdev;
```

```
   DX_CAP capp;
   .
   .
   /* open the channel using dx_open( ). Obtain channel device descriptor in
    * chdev
    */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1)  {
     /* process error */
   }

   /* take the phone off-hook */
   if (dx_sethook(chdev,DX_OFFHOOK,EV_SYNC) == -1) {
     /* process error */
   }

   /* Set the DX_CAP structure as needed for call analysis. Perform the
    * outbound dial with call analysis enabled
    */
   if ((cares = dx_dial(chdev,"5551212",&capp,DX_CALLP|EV_SYNC)) == -1) {
     /* perform error routine */
   }

   switch (cares) {
     case CR_CNCT:  /* Call Connected, get some additional info */
      printf("\nDuration of short low - %ld ms",ATDX_SHORTLOW(chdev)*10);
      printf("\nDuration of long low  - %ld ms",ATDX_LONGLOW(chdev)*10);
      printf("\nDuration of answer    - %ld ms",ATDX_ANSRSIZ(chdev)*10);
      break;
     case CR_CEPT:  /* Operator Intercept detected */
      printf("\nFirst frequency detected - %ld Hz",ATDX_FRQHZ(chdev));
      printf("\nSecond frequency detected - %ld Hz", ATDX_FRQHZ2(chdev));
      printf("\nThird frequency detected - %ld Hz", ATDX_FRQHZ3(chdev));

      printf("\nDuration of first frequency - %ld ms", ATDX_FRQDUR(chdev));
      printf("\nDuration of second frequency - %ld ms",
             ATDX_FRQDUR2(chdev));
      printf("\nDuration of third frequency - %ld ms", ATDX_FRQDUR3(chdev));
      break;
     case CR_BUSY:
      break;
      .
      .
   }
}
```

See Also

Related to Call Analysis:

- **dx_dial( )**
- **ATDX_CPTERM( )**
- DX_CAP structure (*Chapter 4. Voice Data Structures and Device Parameters*)
- "Call Analysis" (*Voice Features Guide for Windows NT*)

SIT Tone Detection - All Boards:

- **ATDX_FRQDUR( )**

SIT Tone Detection - DSP Boards:

- **ATDX_FRQDUR2( )**
- **ATDX_FRQDUR3( )**
- **ATDX_FRQHZ2( )**
- **ATDX_FRQHZ3( )**

Name: long ATDX_FRQHZ2(chdev)
Inputs: int chdev • valid Dialogic channel device handle
Returns: second tone frequency in hz
AT_FAILURE if error
Includes: srllib.h
dxxxlib.h
Category: Extended Attribute

■ Description

When **dx_dial( )** terminates due to Operator Intercept, the **ATDX_FRQHZ2( )** function can return frequency of second detected tone in Hz.

Termination due to Operator Intercept is indicated by **ATDX_CPTERM( )** returning CR_CEPT.

NOTE: For more information about tri-tone SIT tone detection, see *Frequency Detection* in the *Voice Features Guide for Windows NT*.

The function parameter is defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.

■ Cautions

None.

■ Example

See the example for **ATDX_FRQHZ( )**.

■ See Also

Related to Call Analysis:

- **dx_dial( )**
- **ATDX_CPTERM( )**
- DX_CAP structure (*Chapter 4. Voice Data Structures and Device Parameters*)
- "Call Analysis" (*Voice Features Guide for Windows NT*)

SIT Tone Detection - All Boards:

- **ATDX_FRQDUR( )**
- **ATDX_FRQHZ( )**
- "Frequency Detection" (*Voice Features Guide for Windows NT*)

SIT Tone Detection - DSP Boards:

- **ATDX_FRQDUR2( )**
- **ATDX_FRQDUR3( )**
- **ATDX_FRQHZ3( )**

Name: long ATDX_FRQHZ3(chdev)
Inputs: int chdev • valid Dialogic channel device handle
Returns: third tone frequency in hz
AT_FAILURE if error
Includes: srllib.h
dxxxlib.h
Category: Extended Attribute

■ Description

When **dx_dial( )** terminates due to Operator Intercept, the **ATDX_FRQHZ3( )** function can return frequency of third detected tone in Hz.

Termination due to Operator Intercept is indicated by **ATDX_CPTERM( )** returning CR_CEPT.

NOTE: For more information about tri-tone SIT tone detection, see *Frequency Detection* in the *Voice Features Guide for Windows NT*.

The function parameter is defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.

■ Cautions

None.

■ Example

See the example for **ATDX_FRQHZ( )**.

■ See Also

Related to Call Analysis:

- **dx_dial( )**
- **ATDX_CPTERM( )**
- DX_CAP structure (*Chapter 4. Voice Data Structures and Device Parameters*)
- "Call Analysis" (*Voice Features Guide for Windows NT*)

SIT Tone Detection - DSP Boards

- **ATDX_FRQDUR2( )**
- **ATDX_FRQDUR3( )**
- **ATDX_FRQHZ2( )**

Name: long ATDX_FRQOUT(chdev)
Inputs: int chdev • valid Dialogic channel device handle
Returns: percentage frequency out-of bounds
AT_FAILURE if error
Includes: srllib.h
dxxxlib.h
Category: Extended Attribute

■ Description

The **ATDX_FRQOUT( )** function returns percentage of a single tone frequency that was not within the specified range in the DX_CAP structure.

Upon detection of a frequency within the range specified by **ca_upperfrq** and lower **ca_lowerfrq**, use this function to optimize the **ca_refctfrq** parameter (which sets the percentage of *time* that the frequency can be out of bounds) in the DX_CAP structure.

The function parameter is defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.

■ Cautions

This function is only for use with non-DSP boards. If you call it on a DSP board, it will return zero.

■ Example

```
/* Call Analysis with user-specified parameters */
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   int cares, chdev;
```

```
   DX_CAP capp;
   .
   .
   /* open the channel using dx_open( ).  Obtain channel device descriptor in
    * chdev
    */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1)  {
     /* process error */
   }


   /* take the phone off-hook */
   if (dx_sethook(chdev,DX_OFFHOOK,EV_SYNC) == -1) {
     /* process error */
   }


   /* Set the DX_CAP structure as needed for call analysis. Perform the
    * outbound dial with call analysis enabled.
    */
   if ((cares = dx_dial(chdev,"5551212",&capp,DX_CALLP|EV_SYNC)) == -1) {
     /* perform error routine */
   }
   switch (cares) {
     case CR_CNCT: /* Call Connected, get some additional info */
      printf("\nDuration of short low - %ld ms",ATDX_SHORTLOW(chdev)*10);
      printf("\nDuration of long low  - %ld ms",ATDX_LONGLOW(chdev)*10);
      printf("\nDuration of answer    - %ld ms",ATDX_ANSRSIZ(chdev)*10);
      break;
     case CR_CEPT: /* Operator Intercept detected */
      printf("\nFrequency detected - %ld Hz",ATDX_FRQHZ(chdev));
      printf("\n%% of Frequency out of bounds - %ld Hz",ATDX_FRQOUT(chdev));
      break;
     case CR_BUSY:
      break;
      .
      .
   }
}
```

■ Errors

This function will fail and return AT_FAILURE if an invalid channel device handle is specified in **chdev**.

■ See Also

Related to Call Analysis:

- **dx_dial( )**
- **ATDX_CPTERM( )**
- DX_CAP structure (*Chapter 4. Voice Data Structures and Device Parameters*)
- "Call Analysis" *(Voice Features Guide for Windows NT)*

Name: long ATDX_FWVER(bddev)
Inputs: int bddev • valid Dialogic board device handle
Returns: D/4x Firmware version if successful
AT_FAILURE if error
Includes: srllib.h
dxxxlib.h
Category: Extended Attribute

■ Description

The **ATDX_FWVER( )** function returns version number of D/4x firmware. On a D/41ESC or a D/xxxSC board the emulated D/4x firmware version is returned.

The function parameter is defined as follows:

Parameter	Description
bddev:	specifies the valid board device handle obtained when the board was opened using **dx_open( )**.

■ Cautions

None.

■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   int bddev;
   long fwver;
   .
   .
   /* Open the board device */
   if ((bddev = dx_open("dxxxB1",NULL)) == -1) {
     /* Process error */
   }
   .
   .
   /* Display Firmware version number */
   if ((fwver = ATDX_FWVER(bddev))==AT_FAILURE) {
```

```
    /* Process error */
  }
  printf("Firmware version %ld\n",fwver);
  .
  .
}
```

■ Errors

This function will fail and return AT_FAILURE if an invalid device handle is specified in **bddev**.

Name: long ATDX_HOOKST(chdev)
Inputs: int chdev • valid Dialogic channel device handle
Returns: current hook state of channel if successful
AT_FAILURE if error
Includes: srllib.h
dxxxlib.h
Category: Extended Attribute

■ Description

The **ATDX_HOOKST( )** function returns the current hook state of the channel **chdev**.

NOTE: Do not call this function for a digital T-1 or E-1 SCbus configuration that includes a D/240SC, D/240SC-T1, D/320SC D/300SC-E1, DTI/241SC, or DTI/301SC board. Transparent signaling for SCbus digital interface devices is not supported in System Release 4.1SC.

The function parameter is defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.

Possible return values are the following:

DX_OFFHOOK	• Channel is off-hook
DX_ONHOOK	• Channel is on-hook

■ Cautions

None.

■ Example

```
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
```

```
main()
{
   int chdev;
   long hookst;


   /* Open the channel device */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
     /* Process error */
   }
   .
   .
   /* Examine Hook state of the channel. Perform application specific action */
   if((hookst = ATDX_HOOKST(chdev)) == AT_FAILURE) {
     /* Process error */
   }


   if(hookst == DX_OFFHOOK) {
     /* Channel is Off-hook */
   }
   .
   .
}
```

■ Errors

This function will fail and return AT_FAILURE if an invalid channel device handle is specified in **chdev**.

■ See Also

- **dx_sethook( )**
- **DX_CST( )** (*Chapter 4. Voice Data Structures and Device Parameters*)

Name:	long ATDX_LINEST(chdev)
Inputs:	int chdev • valid Dialogic channel device handle
Returns:	current line status of channel if successful AT_FAILURE if error
Includes:	srllib.h dxxxlib.h
Category:	Extended Attribute

■ Description

The **ATDX_LINEST( )** function returns a bitmapped representation of activity on the line at that instant connected to the channel **chdev**.

The function parameter is defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.

Possible return values are the following:

RLS_SILENCE	• Silence on the line
RLS_DTMF	• present
RLS_LCSENSE	• not present
RLS_RING	• Ring not present
RLS_HOOK	• Channel is on-hook
RLS_RINGBK	• Audible ringback detected

■ Cautions

None.

■ Example

```
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
```

```
{
   int chdev;
   long linest;

   /* Open the channel device */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
     /* Process error */
   }

   /* Examine line status bitmap of the channel. Perform application-specific
    * action
    */
   if((linest = ATDX_LINEST(chdev)) == AT_FAILURE) {
     /* Process error */
   }

   if(linest & RLS_LCSENSE) {
     /* No loop current */
   }
   .
   .
}
```

■ Errors

This function will fail and return AT_FAILURE if an invalid channel device handle is specified in **chdev**.

Name: long ATDX_LONGLOW(chdev)
Inputs: int chdev • valid Dialogic channel device handle
Returns: duration of longer silence if successful
AT_FAILURE if error
Includes: srllib.h
dxxxlib.h
Category: Extended Attribute

■ Description

The **ATDX_LONGLOW( )** function returns duration of the longer silence, in 10 ms units, of the initial signal that occurred during Call Analysis on the channel **chdev.** This function can be used in conjunction with **ATDX_SIZEHI( )** and **ATDX_SHORTLOW( )** to determine the elements of an established cadence. See the *Voice Features Guide for Windows NT* for further information.

The function parameter is defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.

■ Cautions

None.

■ Example

```
/* Call Analysis with user-specified parameters */
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   int cares, chdev;
   DX_CAP capp;
   .
   .
   /* open the channel using dx_open( ).  Obtain channel device descriptor in
```

```
    * chdev
    */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1)  {
     /* process error */
   }

   /* take the phone off-hook */
   if (dx_sethook(chdev,DX_OFFHOOK,EV_SYNC) == -1) {
     /* process error */
   }

   /* Set the DX_CAP structure as needed for call analysis. Perform the
    * outbound dial with call analysis enabled
    */
   if ((cares = dx_dial(chdev,"5551212",&capp,DX_CALLP|EV_SYNC)) == -1) {
     /* perform error routine */
   }
   switch (cares) {
     case CR_CNCT: /* Call Connected, get some additional info */
      printf("\nDuration of short low - %ld ms",ATDX_SHORTLOW(chdev)*10);
      printf("\nDuration of long low  - %ld ms",ATDX_LONGLOW(chdev)*10);
      printf("\nDuration of answer    - %ld ms",ATDX_ANSRSIZ(chdev)*10);
      break;
     case CR_CEPT: /* Operator Intercept detected */
      printf("\nFrequency detected - %ld Hz",ATDX_FRQHZ(chdev));
      printf("\n%% of Frequency out of bounds - %ld Hz",ATDX_FRQOUT(chdev));
      break;
     case CR_BUSY:
      .
      .
   }
}
```

■ Errors

This function will fail and return AT_FAILURE if an invalid channel device handle is specified in **chdev**.

■ See Also

Related to Call Analysis::

- **dx_dial( )**
- **ATDX_CPTERM( )**
- DX_CAP structure (*Chapter 4. Voice Data Structures and Device Parameters*)
- "Call Analysis" (*Voice Features Guide for Windows NT*)
- "Cadence Detection" (*Voice Features Guide for Windows NT*)

Name: long ATDX_PHYADDR(bddev)
Inputs: int bddev
- valid Dialogic board device handle

Returns: physical address of board if successful
AT_FAILURE if error
Includes: srllib.h
dxxxlib.h
Category: Extended Attribute

■ Description

The **ATDX_PHYADDR( )** function returns the physical address of the board **bddev**.

The function parameter is defined as follows:

Parameter	Description
bddev:	specifies the valid board device handle obtained when the board was opened using **dx_open( )**.

■ Cautions

None.

■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   int bddev;
   long phyaddr;

   /* Open the board device */
   if ((bddev = dx_open("dxxxB1",NULL)) == -1) {
     /* Process error */
   }

   if((phyaddr = ATDX_PHYADDR(bddev)) == AT_FAILURE) {
     /* Process error */
```

```
   }

   printf("Board is at address %X\n",phyaddr);
   .
   .
}
```

■ Errors

This function will fail and return AT_FAILURE if an invalid board device handle is specified in **bddev**.

Name: long ATDX_SHORTLOW(chdev)
Inputs: int chdev • valid Dialogic channel device handle
Returns: duration of shorter silence if successful
AT_FAILURE if error
Includes: srllib.h
dxxxlib.h
Category: Extended Attribute

■ Description

The **ATDX_SHORTLOW( )** function returns duration of shorter silence of the initial signal that occurred during Call Analysis on the channel **chdev**. This function can be used in conjunction with **ATDX_SIZEHI( )** and **ATDX_LONGLOW( )** to determine the elements of an established cadence. See the *Voice Features Guide for Windows NT* for further information.

Compare the results of this function with the DX_CAP field ca_lo2rmin to determine whether the cadence is a double or single ring.

If the result of **ATDX_SHORTLOW( )** is less than the ca_lo2rmin field this indicates a double ring cadence.

If the result of **ATDX_SHORTLOW( )** is greater than the ca_lo2rmin field this indicates a single ring.

The function parameter is defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.

■ Cautions

None.

■ Example

```
/* Call Analysis with user-specified parameters */
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   int cares, chdev;
   DX_CAP capp;
   .
   .
   /* open the channel using dx_open( ).  Obtain channel device descriptor
    * in chdev
    */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1)  {
     /* process error */
   }

   /* take the phone off-hook */
   if (dx_sethook(chdev,DX_OFFHOOK,EV_SYNC) == -1) {
     /* process error */
   }

   /* Set the DX_CAP structure as needed for call analysis. Perform the
    * outbound dial with call analysis enabled
    */
   if ((cares = dx_dial(chdev,"5551212",&capp,DX_CALLP|EV_SYNC)) == -1) {
     /* perform error routine */
   }
   switch (cares) {
     case CR_CNCT: /* Call Connected, get some additional info */
      printf("\nDuration of short low - %ld ms",ATDX_SHORTLOW(chdev)*10);
      printf("\nDuration of long low  - %ld ms",ATDX_LONGLOW(chdev)*10);
      printf("\nDuration of answer    - %ld ms",ATDX_ANSRSIZ(chdev)*10);
      break;
     case CR_CEPT:     /* Operator Intercept detected */
      printf("\nFrequency detected - %ld Hz",ATDX_FRQHZ(chdev));
      printf("\n%% of Frequency out of bounds - %ld Hz",ATDX_FRQOUT(chdev));
      break;
     case CR_BUSY:
      .
      .
   }

}
```

■ Errors

This function will fail and return AT_FAILURE if an invalid channel device handle is specified in **chdev**.

See Also

- **dx_dial( )**
- **ATDX_LONGLOW( )**
- **ATDX_SIZEHI( )**
- **ATDX_CPTERM( )**
- DX_CAP structure (*Chapter 4. Voice Data Structures and Device Parameters*)
- "Call Analysis" (*Voice Features Guide for Windows NT*)
- "Cadence Detection" (*Voice Features Guide for Windows NT*)

Name: long ATDX_SIZEHI(chdev)
Inputs: int chdev • valid Dialogic channel device handle
Returns: non-silence duration in 10 ms units if successful
AT_FAILURE if error
Includes: srllib.h
dxxxlib.h
Category: Extended Attribute

■ Description

The **ATDX_SIZEHI( )** function returns duration of initial non-silence, in 10 ms units, during Call Analysis on the channel **chdev**. This function can be used in conjunction with **ATDX_SIZEHI( )** and **ATDX_LONGLOW( )** to determine the elements of an established cadence. See the *Voice Features Guide for Windows NT* for further information.

The function parameter is defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.

■ Cautions

None.

■ Example

```
/* Call Analysis with user-specified parameters */
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   int cares, chdev;
   DX_CAP capp;
   .
   .
   /* open the channel using dx_open( ).  Obtain channel device descriptor
```

```
    * in chdev
    */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1)  {
     /* process error */
   }

   /* take the phone off-hook */
   if (dx_sethook(chdev,DX_OFFHOOK,EV_SYNC) == -1) {
     /* process error */
   }

   /* Set the DX_CAP structure as needed for call analysis. Perform the
    * outbound dial with call analysis enabled
    */
   if ((cares = dx_dial(chdev,"5551212",&capp,DX_CALLP|EV_SYNC)) == -1) {
     /* perform error routine */
   }
   switch (cares) {
     case CR_CNCT:     /* Call Connected, get some additional info */
      printf("\nDuration of short low - %ld ms",ATDX_SHORTLOW(chdev)*10);
      printf("\nDuration of long low  - %ld ms",ATDX_LONGLOW(chdev)*10);
      printf("\nDuration of non-silence  - %ld ms",ATDX_SIZEHI(chdev)*10);
      break;
     case CR_CEPT:     /* Operator Intercept detected */
      printf("\nFrequency detected - %ld Hz",ATDX_FRQHZ(chdev));
      printf("\n%% of Frequency out of bounds - %ld Hz",ATDX_FRQOUT(chdev));
      break;
     case CR_BUSY:
      .
      .
   }

}
```

■ Errors

This function will fail and return AT_FAILURE if an invalid channel device handle is specified in **chdev**.

■ See Also

- **dx_dial( )**
- **ATDX_LONGLOW( )**
- **ATDX_SHORTLOW( )**
- **ATDX_CPTERM( )**
- DX_CAP structure (*Chapter 4. Voice Data Structures and Device Parameters*)
- "Call Analysis" (*Voice Features Guide for Windows NT*)
- "Cadence Detection" (*Voice Features Guide for Windows NT*)

Name: long ATDX_STATE(chdev)
Inputs: int chdev • valid Dialogic channel device handle
Returns: current state of channel if successful
AT_FAILURE if error
Includes: srllib.h
dxxxlib.h
Category: Extended Attribute

■ Description

The **ATDX_STATE( )** function returns the current state of the channel **chdev**.

The function parameter is defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.

Possible return values are the following:

- CS_DIAL • Dial state
- CS_CALL • Call state
- CS_GTDIG • Get Digit state
- CS_HOOK • Hook state
- CS_IDLE • Idle state
- CS_PLAY • Play state
- CS_RECD • Record state
- CS_STOPD • Stopped state
- CS_TONE • Playing tone state
- CS_WINK • Wink state

NOTE: When a Voice board is being used with a FAX/xxx board to send and receive faxes the following states may be returned:

- CS_SENDFAX • Channel is in a fax transmission state.
- CS_RECVFAX • Channel is in a fax reception state.

NOTE: A device is idle if there is no I/O function active on it.

■ Cautions

None.

■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
m#include <windows.h>

main()
{
   int chdev;
   long chstate;

   /* Open the channel device */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
     /* Process error */
   }
   .
   .
   /* Examine state of the channel. Perform application specific action based
    * on state of the channel
    */
   if((chstate = ATDX_STATE(chdev)) == AT_FAILURE) {
     /* Process error */
   }

   printf("current state of channel %s = %ld\n", ATDX_NAMEP(chdev), chstate);
   .
   .

}
```

■ Errors

This function will fail and return AT_FAILURE if an invalid channel device handle is specified in **chdev**.

Name:	long ATDX_TERMMSK(chdev)
Inputs:	int chdev • valid Dialogic channel device handle
Returns:	channel's last termination bitmap if successful AT_FAILURE if error
Includes:	srllib.h dxxxlib.h
Category:	Extended Attribute

■ Description

The **ATDX_TERMMSK( )** function returns a bitmap containing the reason(s) for the last termination on the channel **chdev**. The bitmap is reset when an I/O function terminates.

The function parameter is defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.

Possible return values are the following:

TM_NORMTERM	• Normal Termination (for **dx_dial( )**, **dx_sethook( )**)
TM_MAXDTMF	• Maximum DTMF count
TM_MAXSIL	• Maximum period of silence
TM_MAXNOSIL	• Maximum period of non-silence
TM_LCOFF	• Loop current off
TM_IDDTIME	• Inter-digit delay
TM_MAXTIME	• Maximum function time
TM_DIGIT	• Specific digit received
TM_PATTERN	• Pattern matched silence off
TM_USRSTOP	• Function stopped by user
TM_EOD	• End of Data reached on playback
TM_TONE	• Tone-on/off event
TM_ERROR	• I/O Device Error

■ Cautions

If several termination conditions are met at the same time, several bits will be set in the termination bitmap.

■ Example

```
#include <stdio.h>
#include <fcntl.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   int chdev;
   long term;
   DX_IOTT iott;
   DV_TPT  tpt[4];

   /* Open the channel device */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
     /* Process error */
   }

   /* Record a voice file. Terminate on receiving a digit, silence, loop
    * current drop, max time, or reaching a byte count of 50000 bytes.
    */

   /* set up DX_IOTT */

   iott.io_type = IO_DEV|IO_EOT;
   iott.io_bufp = 0;
   iott.io_offset = 0;
   iott.io_length = 50000;

   if((iott.io_fhandle = dx_fileopen("file.vox", O_RDWR)) == -1)  {
     /* process error */
   }

   /* set up DV_TPTs for the required terminating conditions */

   dx_clrtpt(tpt,4);
   tpt[0].tp_type   = IO_CONT;
   tpt[0].tp_termno = DX_MAXDTMF;           /* Maximum digits */
   tpt[0].tp_length = 1;                     * terminate on the first digit */
   tpt[0].tp_flags  = TF_MAXDTMF;           /* Use the default flags */
   tpt[1].tp_type   = IO_CONT;
   tpt[1].tp_termno = DX_MAXTIME;           /* Maximum time */
   tpt[1].tp_length = 100;                  /* terminate after 10 secs */
   tpt[1].tp_flags  = TF_MAXTIME;           /* Use the default flags */
   tpt[2].tp_type   = IO_CONT;
   tpt[2].tp_termno = DX_MAXSIL;            /* Maximum Silence */
   tpt[2].tp_length = 30;                   /* terminate on 3 sec silence */
   tpt[2].tp_flags  = TF_MAXSIL;            /* Use the default flags */
```

```
   tpt[3].tp_type   = IO_EOT;               /* last entry in the table */
   tpt[3].tp_termno = DX_LCOFF;             /* terminate on loop current drop */
   tpt[3].tp_length = 10;                   /* terminate on 1 sec silence */
   tpt[3].tp_flags  = TF_LCOFF;             /* Use the default flags */


   /* Now record to the file */
   if (dx_rec(chdev,&iott,tpt,EV_SYNC) == -1) {
     /* process error */
   }

   /* Examine bitmap to determine if digits caused termination */
   if((term = ATDX_TERMMSK(chdev)) == AT_FAILURE) {
     /* Process error */
   }

   if(term & TM_MAXDTMF) {
     printf("Terminated on digits\n");
       .
       .
   }

}
```

■ Errors

This function will fail and return AT_FAILURE if an invalid channel device handle is specified in **chdev**.

■ See Also

Setting Termination Conditions:

- DV_TPT (*Appendix A*)

Retrieving Termination Events - asynchronously:

- Event Management functions (*Standard Runtime Library Programmer's Guide for Windows NT*)

Name: long ATDX_TONEID(chdev)
Inputs: int chdev • valid Dialogic channel device handle
Returns: channel's last termination bitmap if successful
AT_FAILURE if error
Includes: srllib.h
dxxxlib.h
Category: Extended Attribute

■ Description

The **ATDX_TONEID( )** function returns the user-defined tone id. Use this function to determine which tone occurred when **ATDX_TERMMSK( )** returns DX_TONE to indicate that an I/O function terminated due to the occurrence of a user-specified tone.

The function parameter is defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.

■ Cautions

None.

■ Example

```
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

#define TID_1   101

main()
{
   TN_GEN          tngen;
   DV_TPT   tpt[ 5 ];
   int      chdev;
```

```
/*
 * Open the D/xxx Channel Device and Enable a Handler
 */
if ( ( chdev = dx_open( "dxxxB1C1", NULL ) ) == -1 ) {
   perror( "dxxxB1C1" );
   exit( 1 );
}

/*
 * Describe a Simple Dual Tone Frequency Tone of 950-
 * 1050 Hz and 475-525 Hz using leading edge detection.
 */
if ( dx_blddt( TID_1, 1000, 50, 500, 25, TN_LEADING )== -1 ) {
   printf( "Unable to build a Dual Tone Template\n" );
}

/*
 * Add the Tone to the Channel
 */
if ( dx_addtone( chdev, NULL, 0 ) == -1 ) {
   printf( "Unable to Add the Tone %d\n", TID_1 );
   printf( "Lasterror = %d  Err Msg = %s\n",
     ATDV_LASTERR( chdev ), ATDV_ERRMSGP( chdev ) );
   dx_close( chdev );
   exit( 1 );
}

/*
 * Build a Tone Generation Template.
 * This template has Frequency1 = 1140,
 * Frequency2 = 1020, amplitute at -10dB for
 * both frequencies and duration of 100 * 10 msecs.
 */
dx_bldtngen( &tngen, 1140, 1020, -10, -10, 100 );

/*
 * Set up the Terminating Conditions
 */
tpt[0].tp_type = IO_CONT;
tpt[0].tp_termno = DX_TONE;
tpt[0].tp_length = TID_1;
tpt[0].tp_flags = TF_TONE;
tpt[0].tp_data = DX_TONEON;


tpt[1].tp_type = IO_CONT;
tpt[1].tp_termno = DX_TONE;
tpt[1].tp_length = TID_1;
tpt[1].tp_flags = TF_TONE;
tpt[1].tp_data = DX_TONEOFF;


tpt[2].tp_type = IO_EOT;
tpt[2].tp_termno = DX_MAXTIME;
tpt[2].tp_length = 6000;
tpt[2].tp_flags = TF_MAXTIME;


if (dx_playtone( chdev, &tngen, tpt, EV_SYNC ) == -1 ){
   printf( "Unable to Play the Tone\n" );
   printf( "Lasterror = %d  Err Msg = %s\n",
     ATDV_LASTERR( chdev ), ATDV_ERRMSGP( chdev ) );
   dx_close( chdev );
   exit( 1 );
```

```
   }

   if ( ATDX_TERMMSK( chdev ) & TM_TONE ) {
      printf( "Terminated by Tone Id = %d\n", ATDX_TONEID( chdev ) );
   }

   /*
    * Continue Processing
    *   .
    *   .
    *   .
    */

   /*
    * Close the opened D/xxx Channel Device
    */
   if ( dx_close( chdev ) != 0 ) {
      perror( "close" );
   }

   /* Terminate the Program */
   exit( 0 );
}
```

■ Errors

This function will fail and return AT_FAILURE if an invalid channel device handle is specified in **chdev**.

Name: long ATDX_TRCOUNT(chdev)
Inputs: int chdev • valid Dialogic channel device handle
Returns: last play/record transfer count if successful
AT_FAILURE if error
Includes: srllib.h
dxxxlib.h
Category: Extended Attribute

■ Description

The **ATDX_TRCOUNT( )** function returns number of bytes transferred during the last play or record on the channel **chdev**.

The function parameter is defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.

■ Cautions

None.

■ Example

```
#include <stdio.h>
#include <fcntl.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   int chdev;
   long trcount;
   DX_IOTT iott;
   DV_TPT  tpt[2];

   /* Open the channel device */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
     /* Process error */
   }
```

```
   /* Record a voice file. Terminate on receiving a digit, max time,
    * or reaching a byte count of 50000 bytes.
    */
   .
   .
   /* set up DX_IOTT */
   iott.io_type = IO_DEV|IO_EOT;
   iott.io_bufp = 0;
   iott.io_offset = 0L;
   iott.io_length = 50000L;
   if((iott.io_fhandle = dx_fileopen("file.vox", O_RDWR)) == -1)  {
     /* process error */
   }

   /* set up DV_TPTs for the required terminating conditions */
   dx_clrtpt(tpt,2);
   tpt[0].tp_type   = IO_CONT;
   tpt[0].tp_termno = DX_MAXDTMF;     /* Maximum digits */
   tpt[0].tp_length = 1;              /* terminate on the first digit */
   tpt[0].tp_flags  = TF_MAXDTMF;     /* Use the default flags */

   tpt[1].tp_type   = IO_EOT;
   tpt[1].tp_termno = DX_MAXTIME;     /* Maximum time */
   tpt[1].tp_length = 100;            /* terminate after 10 secs */
   tpt[1].tp_flags  = TF_MAXTIME;     /* Use the default flags */

   /* Now record to the file */
   if (dx_rec(chdev,&iott,tpt,EV_SYNC) == -1) {
     /* process error */
   }

   /* Examine transfer count */
   if((trcount = ATDX_TRCOUNT(chdev)) == AT_FAILURE) {
     /* Process error */
   }

   printf("%ld bytes recorded\n", trcount);
   .
   .
}
```

■ Errors

This function will fail and return AT_FAILURE if an invalid channel device handle is specified in **chdev**.

Name:	int dx_addspddig(chdev, digit, adjval)	
Inputs:	int chdev	• valid Dialogic channel device handle
	char digit	• DTMF digit
	short adjval	• speed adjustment value
Returns:	0 if success -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	Speed and Volume Convenience	

■ Description

dx_addspddig( ) is a convenience function that sets a DTMF digit to adjust speed by a specified amount, immediately and for all subsequent plays on the specified channel (until changed or cancelled).

NOTES: **1.** Calls to this function are cumulative. To reset a digit condition, you need to clear all adjustment conditions using a **dx_clrsvcond( )**, and then reset the new condition.

2. Speed control is supported on D/21D, D/21E, D/41D, D/41ESC, D/41E, D/81A, D/121B, D/160SC-LS, D/240SC, D/240SC-T1, D/300SC-E1 and D/320SC boards.

This function assumes that the Speed Modification Table has not been modified using the **dx_setsvmt( )** function.

For information about the speed and volume functions and the Speed and Volume Modification Tables, see the *Voice Features Guide for Windows NT*. For information about the speed and volume data structures see *Sections* DV_TPT

Parameter	Description
chdev:	specifies the valid channel device handle obtained by a call to **dx_open( )**.
digit:	specifies a DTMF digit (0-9, *,#) that will modify speed by the amount specified in **adjval**. To start play-speed at the origin, set **digit** to NULL and set **adjval** to SV_NORMAL.

Parameter	Description	
adjval:	specifies one of the following the speed adjustment values to take effect whenever the digit specified in **digit** occurs:	
	SV_ADD10PCT	Increase play - speed by 10%
	SV_ADD20PCT	Increase play - speed by 20%
	SV_ADD30PCT	Increase play - speed by 30%
	SV_ADD40PCT	Increase play - speed by 40%
	SV_ADD50PCT	Increase play - speed by 50%
	SV_SUB10PCT	Decrease play - speed by 10%
	SV_SUB20PCT	Decrease play - speed by 20%
	SV_SUB30PCT	Decrease play - speed by 30%
	SV_SUB40PCT	Decrease play - speed by 40%
	SV_NORMAL	Set play - speed to origin (regular speed) when the play begins. **digit** must be set to NULL.

■ Cautions

1. This function is cumulative. To reset or remove any condition, you should clear all conditions, and reset if required (e.g., If DTMF digit "1" has already been set to increase play-speed by one step, a second call that attempts to redefine "1" to the origin, will have no effect on speed or volume but it will be added to the array of conditions. The digit will retain its original setting).
2. The digit that causes the play adjustment will not be passed to the digit buffer, so it cannot be retrieved using **dx_getdig( )** or **ATDX_BUFDIGS( )**
3. 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.
4. Speed control is supported on all the D/21D, D/21E, D/41D, D/41E, D/41ESC, D/81A, D/121B, D/160SC-LS, D/240SC, D/240SC-T1, D/300SC-E1, and D/320SC boards.

■ Example

```
#include <stdio.h>
```

```
#include <errno.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

/*
 * Global Variables
 */

main()
{
   int  dxxxdev;

   /*
    * Open the Voice Channel Device and Enable a Handler
    */
   if ( ( dxxxdev = dx_open( "dxxxB1C1", NULL) ) == -1 ) {
      perror( "dxxxB1C1" );
      exit( 1 );
   }

   /*
    * Add a Speed Adjustment Condition - increase the
    * playback speed by 30% whenever DTMF key 1 is pressed.
    */
   if ( dx_addspddig( dxxxdev, '1', SV_ADD30PCT ) == -1 ) {
      printf("Unable to Add a Speed Adjustment Condition\n");
      printf( "Lasterror = %d  Err Msg = %s\n",
         ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
      dx_close( dxxxdev );
      exit( 1 );
   }

   /*
    * Continue Processing
    *   .
    *   .
    *   .
    */

   /*
    * Close the opened Voice Channel Device
    */
   if ( dx_close( dxxxdev ) != 0 ) {
      perror( "close" );
   }

   /* Terminate the Program */
   exit( 0 );
}
```

■ Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	• Invalid Parameter
EDX_BADPROD	• Function not supported on this board
EDX_SYSTEM	• Windows NT system error - check **errno**
EDX_SVADJBLK	• Invalid Number of Play Adjustment Blocks

See Also

- **dx_addvoldig( )**
- **dx_adjsv( )**
- **dx_clrsvcond( )**
- **dx_getcursv( )**
- **dx_getsvmt( )**
- **dx_setsvcond( )**
- **dx_setsvmt( )**
- "Speed and Volume Modification Tables" (*Voice Features Guide for Windows NT*)
- *DX_SVCB* data structure (*Chapter 4. Voice Data Structures and Device Parameters*)

Name:	int dx_addtone(chdev,digit,digtype)	
Inputs:	int chdev	• valid Dialogic channel device handle
	unsigned char digit	• optional digit associated with the bound tone
	unsigned char digtype	• digit type
Returns:	0 if success -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	Global Tone Detection	

■ Description

The **dx_addtone( )** function adds the tone that was defined by the most recent **dx_blddt( )** (or other Global Tone Detection build-tone) function call, to the specified channel. Adding a user-defined tone to a channel downloads it to the board and enables detection of tone-on and tone-off events for that tone by default.

Use **dx_distone( )** to disable detection of the tone, without removing the tone from the channel. Detection can be enabled again using **dx_enbtone( )**. For example, if you only want to be notified of tone-on events, you should call **dx_distone( )** to disable detection of tone-off events.

■ Detection Notification

Tone-on and tone-off events are call status transition events. Retrieval of these events is handled differently for asynchronous and synchronous applications. Table 2 outlines the different processes:

Table 2. Asynchronous/Synchronous CST Event Handling

	Synchronous	Asynchronous
1.	Call **dx_addtone( )**, or **dx_enbtone( )**	Call **dx_addtone( )** or **dx_enbtone( )** to enable tone-on/off detection.
2.	Call **dx_getevt( )** to wait for CST	Use SRL to asynchronously wait for

	event(s). Events are returned in the *DX_EBLK* structure	TDX_CST event(s)
3.	N/A	Use **sr_getevtdatap( )** to retrieve *DX_CST* structure

NOTE: These procedures are the sa me 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. When the digit is detected, it is placed in the digit buffer and can be used for termination.

■ Setting User-Defined Tones as Termination Conditions

Detection of a user-defined tone can be specified as a termination condition for I/O functions. Set the **tp_termno** field in the DV_TPT to DX_TONE, and specify DX_TONEON or DX_TONEOFF in the **tp_data** field.

The function parameters are described below.

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.
digit:	(optional) specifies the digit to associate with the tone. When the tone is detected, the digit will be placed in the *DV_DIGIT* digit buffer. These digits can be retrieved using **dx_getdig( )** (i.e., they can be used in the same way as DTMF digits, for example). If you do not specify a digit, the tone will be indicated by a DE_TONEON event or DE_TONEOFF event.
digtype:	specifies the type of digit the channel will detect. Specify one of the following values. • DG_USER1 • DG_USER2 • DG_USER3 • DG_USER4 • DG_USER5

Parameter	Description
	Up to twenty digits can be associated with each of these digit types.

NOTE: These types can be specified in addition to the digit types already defined for the Voice Library (e.g., DTMF, MF) which are specified using **dx_setdigtyp( )**.

■ Cautions

1. Ensure that **dx_blddt( )** (or another appropriate "build tone" function) has been called to define a tone prior to adding it to the channel using **dx_addtone( )**, otherwise an error will occur.
2. The **dx_addtone( )** function may not be used to change a tone that has previously been added.
3. The number of tones that can be added to a channel is dependent on the type of board. See the *Voice Features Guide for Windows NT* for details.

■ Example

```
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

#define TID_1   101
#define TID_2   102
#define TID_3   103
#define TID_4   104

main()
{
   int  dxxxdev;

   /*
    * Open the Voice Channel Device and Enable a Handler
    */
   if ( ( dxxxdev = dx_open( "dxxxB1C1", NULL) ) == -1 ) {
      perror( "dxxxB1C1" );
      exit( 1 );
   }

   /*
    * Describe a Simple Dual Tone Frequency Tone of 950-
    * 1050 Hz and 475-525 Hz using leading edge detection.
    */
```

```
if ( dx_blddt( TID_1, 1000, 50, 500, 25, TN_LEADING ) == -1 ) {
   printf( "Unable to build a Dual Tone Template\n" );
}

/*
 * Bind the Tone to the Channel
 */
if ( dx_addtone( dxxxdev, NULL, 0 ) == -1 ) {
   printf( "Unable to Bind the Tone %d\n", TID_1 );
   printf( "Lasterror = %d  Err Msg = %s\n",
       ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ));
   dx_close( dxxxdev );
   exit( 1 );
}

/*
 * Describe a Dual Tone Frequency Tone of 950-1050 Hz
 * and 475-525 Hz. On between 190-210 msecs and off
 * 990-1010 msecs and a cadence of 3.
 */
if ( dx_blddtcad( TID_2, 1000, 50, 500, 25, 20, 1, 100, 1, 3 ) == -1 ) {
   printf("Unable to build a Dual Tone Cadence Template\n" );
}

/*
 * Bind the Tone to the Channel
 */
if ( dx_addtone( dxxxdev, 'A', DG_USER1 ) == -1 ) {
   printf( "Unable to Bind the Tone %d\n", TID_2 );
   printf( "Lasterror = %d  Err Msg = %s\n",
       ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ));
   dx_close( dxxxdev );
   exit( 1 );
}

/*
 * Describe a Simple Single Tone Frequency Tone of
 * 950-1050 Hz using trailing edge detection.
 */
if ( dx_bldst( TID_3, 1000, 50, TN_TRAILING ) == -1 ) {
   printf( "Unable to build a Single Tone Template\n" );
}

/*
 * Bind the Tone to the Channel
 */
if ( dx_addtone( dxxxdev, 'D', DG_USER2 ) == -1 ) {
   printf( "Unable to Bind the Tone %d\n", TID_3 );
   printf( "Lasterror = %d  Err Msg = %s\n",
       ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
   dx_close( dxxxdev );
   exit( 1 );
}

/*
 * Describe a Single Tone Frequency Tone of 950-1050 Hz.
 * On between 190-210 msecs and off 990-1010 msecs and
 * a cadence of 3.
 */
if ( dx_bldstcad( TID_4, 1000, 50, 20, 1, 100, 1, 3 ) == -1 ) {
   printf("Unable to build a Single Tone Cadence Template\n");
}
```

```
   /*
    * Bind the Tone to the Channel
    */
   if ( dx_addtone( dxxxdev, NULL, 0 ) == -1 ) {
      printf( "Unable to Bind the Tone %d\n", TID_4 );
      printf( "Lasterror = %d  Err Msg = %s\n",
          ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
      dx_close( dxxxdev );
      exit( 1 );
   }

   /*
    * Continue Processing
    *   .
    *   .
    *   .
    */

   /*
    * Close the opened Voice Channel Device
    */
   if ( dx_close( dxxxdev ) != 0 ) {
      perror( "close" );
   }

   /* Terminate the Program */
   exit( 0 );
}
```

■ Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_ASCII	• Invalid ASCII value in tone template description
EDX_BADPARM	• Invalid parameter
EDX_BADPROD	• Function not supported on this board
EDX_CADENCE	• Invalid cadence component value
EDX_DIGTYPE	• Invalid Dig_Type value in tone template description
EDX_FREQDET	• Invalid tone frequency
EDX_INVSUBCMD	• Invalid sub-command
EDX_MAXTMPLT	• Maximum number of user-defined tones for the board
EDX_SYSTEM	• Windows NT System error - check **errno**
EDX_TONEID	• Invalid tone template ID

See Also

Global Tone Detection functions:

- **dx_blddt( )**, **dx_bldst( )**, **dx_blddtcad( )**, **dx_bldstcad( )**
- **dx_distone( )**
- **dx_enbtone( )**
- "Global Tone Detection" (*Voice Features Guide for Windows NT*)

Event Retrieval:

- **dx_getevt( )**
- *DX_CST* data structure
- **sr_getevtdatap( )** (in the *Standard Runtime Library Programmer's Guide for Windows NT*)

Digit Retrieval:

- **dx_getdig( )**
- **dx_setdigtyp( )**
- *DV_DIGIT*

Name:	int dx_addvoldig(chdev,digit,adjval)	
Inputs:	int chdev	• valid Dialogic channel device handle
	char digit	• DTMF digit
	short adjval	• volume adjustment value
Returns:	0 if success -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	Speed and Volume Convenience	

■ Description

dx_addvoldig( ) is a convenience function that sets a DTMF digit to immediately adjust volume by a specified amount for all subsequent plays on the specified channel (until changed or cancelled).

NOTES: **1.** Calls to this function are cumulative. To reset a digit condition, you need to clear all adjustment conditions using a **dx_clrsvcond( )**, and then reset the new condition.

2. Volume control is supported on D/21D, D/21E, D/41D, D/41E, D/41ESC, D/81A, D/121, D/121A, D/121B, D/160SC-LS, D/240SC, D/240SC-T1, D/300SC-E1, and D/320SC boards.

This function assumes that the Volume Modification Table has not been modified using the **dx_setsvmt( )** function.

For information about the speed and volume functions and the Speed and Volume Modification Tables, see the *Voice Features Guide for Windows NT*.

Parameter	Description
chdev:	specifies the valid channel device handle obtained by a call to **dx_open( )**.
digit:	specifies a DTMF digit (0-9, *,#) that will modify volume by the amount specified in **adjval**. To start play-volume at the origin, set **digit** to NULL and set **adjval** to SV_NORMAL.
adjval:	specifies one of the following the speed adjustment values

Parameter	Description	
	to take effect whenever the digit specified in **digit** occurs:	
	SV_ADD2DB	Increase play-volume by 2DB
	SV_ADD4DB	Increase play-volume by 4DB
	SV_ADD6DB	Increase play-volume by 6DB
	SV_ADD8DB	Increase play-volume by 8DB
	SV_SUB2DB	Decrease play-volume by 2DB
	SV_SUB4DB	Decrease play-volume by 4DB
	SV_SUB6DB	Decrease play-volume by 6DB
	SV_SUB8DB	Decrease play-volume by 8DB
	SV_NORMAL	Set play-volume to origin when the play begins (**digit** must be set to NULL)

Cautions

1. This function is cumulative. To reset or remove any condition, you should clear all adjustment conditions, and reset them if required. (e.g., If DTMF digit "1" has already been set to increase play-volume by one step, a second call that attempts to redefine "1" to the origin (regular volume), will have no effect on the volume, but will add it to the condition array. The digit will retain its original setting).
2. The digit that causes the play adjustment will not be passed to the digit buffer, so it cannot be retrieved using **dx_getdig( )** and will not be included in the result of **ATDX_BUFDIGS( )** which retrieves the number of digits in the buffer.
3. 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.
4. Volume control is supported on the D/21D, D/21E, D/41D, D/41E, D/41ESC, D/81A, D/121, D/121A, D/121B, D/160SC-LS, D/240SC, D/240SC-T1, D/300SC-E1, and D/320SC boards only.

Example

```
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
```

```
#include <dxxxlib.h>
#include <windows.h>

/*
 * Global Variables
 */
main()
{
   int  dxxxdev;

   /*
    * Open the Voice Channel Device and Enable a Handler
    */
   if ( ( dxxxdev = dx_open( "dxxxB1C1", NULL) ) == -1 ) {
      perror( "dxxxB1C1" );
      exit( 1 );
   }

   /*
    * Add a Speed Adjustment Condition - decrease the
    * playback volume by 2dB whenever DTMF key 2 is pressed.     */
   if ( dx_addvoldig( dxxxdev, '2', SV_SUB2DB ) == -1 ) {
      printf( "Unable to Add a Volume Adjustment" );
      printf( " Condition\n");
      printf( "Lasterror = %d  Err Msg = %s\n",
         ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
      dx_close( dxxxdev );
      exit( 1 );
   }

   /*
    * Continue Processing
    *   .
    *   .
    *   .
    */

   /*
    * Close the opened Voice Channel Device
    */
   if ( dx_close( dxxxdev ) != 0 ) {
      perror( "close" );
   }

   /* Terminate the Program */
   exit( 0 );
}
```

■ Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	• Invalid Parameter
EDX_BADPROD	• Function not supported on this board

EDX_SVADJBLKS	• Invalid Number of Play Adjustment Blocks
EDX_SYSTEM	• Windows NT system error - check **errno**

■ See Also

Related Speed and Volume functions:

- **dx_addspddig( )**
- **dx_adjsv( )**
- **dx_clrsvcond( )**
- **dx_getcursv( )**
- **dx_getsvmt( )**
- **dx_setsvcond( )**
- **dx_setsvmt( )**

Name: int dx_adjsv(chdev,tabletype,action,adjsize)

Inputs:

int chdev	• valid channel device handle
unsigned short tabletype	• table to set (speed or volume)
unsigned short action	• how to adjust (absolute position, relative change or toggle)
unsigned short adjsize	• adjustment size

Returns: 0 if successful
-1 if failure

Includes: srllib.h
dxxxlib.h

Category: Speed and Volume

■ Description

The **dx_adjsv( )** function adjusts speed or volume immediately, and for all subsequent plays on a specified channel (until changed or cancelled). Speed or volume can be set to a specific value, adjusted incrementally, or can be set to toggle. See the **action** parameter description for information.

dx_adjsv( ) utilizes the Speed and Volume Modification Tables to make adjustments to play-speed or play-volume. These tables have 21 entries that represent different levels of speed or volume. There are up to ten levels above and below the regular speed or volume. These tables can be set with explicit values using **dx_setsvmt( )** or default values can be used. Refer to the *Voice Features Guide for Windows NT* for detailed information about these tables.

NOTES:

1. This function is similar to **dx_setsvcond( )**. Use **dx_adjsv( )** to explicitly adjust the play immediately, and use **dx_setsvcond( )** to adjust the play in response to specified conditions. See the description of **dx_setsvcond( )** for more information.
2. Whenever a play is started its speed and volume is based on the most recent modification.

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.
tabletype:	specifies whether to modify the play-back using a value from the Speed or the Volume Modification Table.

Parameter	Description	
	SV_SPEEDTBL	Use the Speed Modification Table
	SV_VOLUMETBL	Use the Volume Modification Table
action:	specifies the type of adjustment to make. Set to one of the following:	
	SV_ABSPOS	Set speed or volume to a specified position in the appropriate table. (The position is set using the **adjsize** parameter).
	SV_RELCURPOS	Adjust speed or volume by the number of steps specified using the **adjsize** parameter.
	SV_TOGGLE	Toggle between values specified using the **adjsize** parameter.
adjsize:	specifies the size of the adjustment. **adjsize** has a different value depending on how the adjustment type is set using the action parameter. Set **adjsize** to one of the following:	

For this *action* value	**Choose this *adjsize* value**
SV_ABSPOS	Specify the position between -10 to +10 in the Speed/Volume Modification Table that contains the required speed or volume adjustment. The origin (regular speed or volume) has a value of 0 in the table.
SV_RELCURPOS	Specify how many positive or negative "steps" in the Speed/Volume Modification Table by which to adjust the speed or volume. For example, specify -2 to lower the speed or volume by 2 steps in the Speed/Volume Modification Table.
SV_TOGGLE	Set the values between which speed or volume will toggle.

SV_TOGORIGIN - sets the speed/volume to toggle between the origin and the last modified level of speed/volume.

SV_CURORIGIN - resets the current speed/volume level to the origin (i.e. regular speed/volume).

SV_CURLASTMOD - sets the current speed/volume to the last modified speed volume level.

SV_RESETORIG - resets the current speed/volume to the origin and the last modified speed/volume to the origin

■ Cautions

Speed and volume control are supported on the D/21D, D/21E, D/41D, D/41E, D/41ESC, D/81A, D/121B, D/160SC-LS, D/240SC, D/240SC-T1, D/300SC-E1 and D/320SC boards only. Do not use the Speed and Volume control functions to control speed on the D/120, D/121, or D/121A boards.

■ Example

```
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   int  dxxxdev;

   /*
    * Open the Voice Channel Device and Enable a Handler
    */
   if ( ( dxxxdev = dx_open( "dxxxB1C1", NULL) ) == -1 ) {
      perror( "dxxxB1C1" );
      exit( 1 );
   }

   /*
    * Modify the Volume of the playback so that it is 4dB
     * higher than normal.
    */
   if ( dx_adjsv( dxxxdev, SV_VOLUMETBL, SV_ABSPOS, SV_ADD4DB ) == -1 ) {
      printf( "Unable to Increase Volume by 4dB\n" );
      printf( "Lasterror = %d  Err Msg = %s\n",
```

```
         ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
      dx_close( dxxxdev );
      exit( 1 );
   }

   /*
    * Continue Processing
    *   .
    *   .
    *   .
    */


   /*
    * Close the opened Voice Channel Device
    */
   if ( dx_close( dxxxdev ) != 0 ) {
      perror( "close" );
   }

   /* Terminate the Program */
   exit( 0 );
}
```

■ Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	• Invalid Parameter
EDX_BADPROD	• Function not supported on this board
EDX_SYSTEM	• Windows NT system error - check **errno**

■ See Also

Related to Speed and Volume:

- **dx_setsvmt( )**
- **dx_getcursv( )**
- **dx_getsvmt( )**
- "Speed and Volume Modification Tables" *(Voice Features Guide for Windows NT)*
- *DX_SVMT* structure (*Chapter 4. Voice Data Structures and Device Parameters*)

Setting Speed and Volume conditions:

- **dx_setsvcond( )**

- **dx_clrsvcond( )**

Name:	int dx_blddt(tid,freq1,fq1dev,freq2,fq2dev,mode)	
Inputs:	unsigned int tid	• tone ID to assign.
	unsigned int freq1	• frequency 1 in Hz
	unsigned int fq1dev	• frequency 1 deviation in Hz
	unsigned int freq2	• frequency 2 in Hz
	unsigned int fq2dev	• frequency 2 deviation in Hz
	unsigned int mode	• leading or trailing edge
Returns:	0 if success -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	Global Tone Detection	

■ Description

The **dx_blddt( )** function defines a simple dual frequency tone. Subsequent calls to **dx_addtone( )** will enable detection of this tone, until another tone is defined.

Issuing a **dx_blddt( )** defines a new tone but **dx_addtone( )** must be used to add the tone to the channel.

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

Parameter	Description
tid:	specifies a unique identifier for the tone. **NOTE:** If you are using R2MF tone detection, reserve the use of tone IDs 101 to 115 for the R2MF tones. See **r2_creatfsig( )** for further information.
freq1:	specifies the first frequency (in Hz) for the tone
frq1dev:	specifies the allowable deviation for the first frequency (in Hz).
freq2:	specifies the second frequency (in Hz) for the tone
frq2dev:	specifies the allowable deviation for the second frequency (in Hz)
mode:	specifies whether tone detection notification will occur on the leading or trailing edge of the tone. Set to one of the following: • TN_LEADING

Parameter	Description
	• TN_TRAILING

■ Cautions

1. Only one tone per process can be defined at any time. Ensure that **dx_blddt( )** is called for each **dx_addtone( )**. The tone is not created until **dx_addtone( )** is called, and a second consecutive call to **dx_blddt( )** will replace the previous tone definition for the channel. If you call **dx_addtone( )** without calling **dx_blddt( )** an error will occur.
2. If you are using R2MF tone detection, reserve the use of tone ID's 101 to 115 for the R2MF tones. See **r2_creatfsig( )** for further information.

■ Example

```
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>


#define TID_1   101
main()
{
   int  dxxxdev;
   /*
    * Open the Voice Channel Device and Enable a Handler
    */
   if ( ( dxxxdev = dx_open( "dxxxB1C1", NULL) ) == -1 ) {
      perror( "dxxxB1C1" );
      exit( 1 );
   }

   /*
    * Describe a Simple Dual Tone Frequency Tone of 950-
    * 1050 Hz and 475-525 Hz using leading edge detection.
    */
   if ( dx_blddt( TID_1, 1000, 50, 500, 25, TN_LEADING ) == -1 ) {
      printf( "Unable to build a Dual Tone Template\n" );
   }

   /*
    * Continue Processing
    *   .
    *   .
    *   .
    */


   /*
    * Close the opened Voice Channel Device
```

```
    */
   if ( dx_close( dxxxdev ) != 0 ) {
      perror( "close" );
   }

   /* Terminate the Program */
   exit( 0 );
}
```

■ Errors

None.

■ See Also

Global Tone Detection:

- "Global Tone Detection" (*Voice Features Guide for Windows NT*)

Building Tones:

- **dx_bldst( )**
- **dx_blddtcad( )**
- **dx_bldstcad( )**

Enabling Tone Detection:

- **dx_addtone( )**
- **dx_distone( )**
- **dx_enbtone( )**

R2MF Tones:

- **r2_creatfsig( )**
- **r2_playbsig( )**

Name:	int dx_blddtcad(tid,freq1,fq1dev,freq2,fq2dev,ontime, ontdev,offtime,offtdev,repcnt)	
Inputs:	unsigned int tid	• tone ID to assign.
	unsigned int freq1	• frequency 1 in Hz
	unsigned int fq1dev	• frequency 1 deviation in Hz
	unsigned int freq2	• frequency 2 in Hz
	unsigned int fq2dev	• frequency 2 deviation in Hz
	unsigned int ontime	• tone-on time in 10ms
	unsigned int ontdev	• tone-on time deviation in 10ms
	unsigned int offtime	• tone-off time in 10ms
	unsigned int offtdev	• tone-off time deviation in 10ms
	unsigned int repcnt	• number of repetitions if cadence
Returns:	0 if success	
	-1 if failure	
Includes:	srllib.h	
	dxxxlib.h	
Category:	Global Tone Detection	

■ Description

The **dx_blddtcad( )** function 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.

Issuing a **dx_blddtcad( )** defines a new tone but **dx_addtone( )** must be used to add the tone to the channel.

Use **dx_addtone( )** to enable detection of the user-defined tone on a channel.

Parameter	Description
tid:	specifies a unique identifier for the tone.
	NOTE: If you are using R2MF tone detection, reserve the use of tone IDs 101 to 115 for the R2MF tones. See **r2_creatfsig( )** for further information.
freq1:	specifies the first frequency (in Hz) for the tone

Parameter	Description
frq1dev:	specifies the allowable deviation for the first frequency (in Hz).
freq2:	specifies the second frequency (in Hz) for the tone
frq2dev:	specifies the allowable deviation for the second frequency (in Hz).
ontime:	specifies the length of time for which the cadence is "on" (in 10ms units)
ontdev:	specifies the allowable deviation for "on" time. (in 10ms units)
offtime:	specifies the length of time for which the cadence is "off" (in 10ms units)
offtdev:	specifies the allowable deviation for "off" time (in 10ms units).
repcnt:	specifies the number of repetitions for the cadence (i.e. the number of times that an on/off signal is repeated).

■ Cautions

1. Only 1 user-defined tone per process can be defined at any time. **dx_blddtcad( )** will replace the previous user-defined tone definition.
2. If you are using R2MF tone detection, reserve the use of tone ID's 101 to 115 for the R2MF tones. See **r2_creatfsig( )** for further information.

■ Example

```
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

#define TID_2   102

main()
{
   int  dxxxdev;

   /*
    * Open the Voice Channel Device and Enable a Handler
    */
   if ( ( dxxxdev = dx_open( "dxxxB1C1", NULL) ) == -1 ) {
```

```
      perror( "dxxxB1C1" );
      exit( 1 );
   }

   /*
    * Describe a Dual Tone Frequency Tone of 950-1050 Hz
    * and 475-525 Hz. On between 190-210 msecs and off
    * 990-1010 msecs and a cadence of 3.
    */
   if ( dx_blddtcad( TID_2, 1000, 50, 500, 25, 20, 1,
               100, 1, 3 ) == -1 ) {
      printf( "Unable to build a Dual Tone Cadence" );
      printf( " Template\n");
   }

   /*
    * Continue Processing
    *   .
    *   .
    */

   /*
    * Close the opened Voice Channel Device
    */
   if ( dx_close( dxxxdev ) != 0 ) {
      perror( "close" );
   }

   /* Terminate the Program */
   exit( 0 );
}
```

■ Errors

None.

■ See Also

Global Tone Detection:

- "Global Tone Detection" (*Voice Features Guide for Windows NT*)

Building Tones:

- **dx_bldst( )**
- **dx_blddt( )**
- **dx_bldstcad( )**

Enabling Tone Detection:

- **dx_addtone( )**
- **dx_distone( )**
- **dx_enbtone( )**

R2MF Tones:

- **r2_creatfsig( )**
- **r2_playbsig( )**

Name:	int dx_bldst(tid,freq,fqdev,mode)	
Inputs:	unsigned int tid	• tone ID to assign.
	unsigned int freq	• frequency in Hz
	unsigned int fqdev	• frequency deviation in Hz
	unsigned int mode	• leading or trailing edge
Returns:	0 if success	
	-1 if failure	
Includes:	srllib.h	
	dxxxlib.h	
Category:	Global Tone Detection	
Mode:		

■ Description

The **dx_bldst( )** function defines a simple single frequency tone. Subsequent calls to **dx_addtone( )** will use this tone, until another tone is defined.

Issuing a **dx_bldst( )** defines a new tone but **dx_addtone( )** must be used to add the tone to the channel.

Use **dx_addtone( )** to enable detection of the user-defined tone on a channel.

Parameter	Description
tid:	specifies a unique identifier for the tone.
	NOTE: If you are using R2MF tone detection, reserve the use of tone IDs 101 to 115 for the R2MF tones. See **r2_creatfsig( )** for further information.
freq:	specifies the frequency (in Hz) for the tone
frqdev:	specifies the allowable deviation for the frequency (in Hz).
mode:	specifies whether detection is on the leading or trailing edge of the tone. Set to one of the following: • TN_LEADING • TN_TRAILING

Cautions

1. Only 1 tone per application may be defined at any time. **dx_bldst( )** will replace the previous user-defined tone definition.
2. If you are using R2MF tone detection, reserve the use of tone IDs 101 to 115 for the R2MF tones. See **r2_creatfsig( )** for further information.

Example

```
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

#define TID_3   103

main()
{
   int  dxxxdev;

   /*
    * Open the Voice Channel Device and Enable a Handler
    */
   if ( ( dxxxdev = dx_open( "dxxxB1C1", NULL) ) == -1 ) {
      perror( "dxxxB1C1" );
      exit( 1 );
   }

   /*
    * Describe a Simple Single Tone Frequency Tone of
    * 950-1050 Hz using trailing edge detection.
    */
   if ( dx_bldst( TID_3, 1000, 50, TN_TRAILING ) == -1 ) {
      printf( "Unable to build a Single Tone Template\n" );
   }

   /*
    * Continue Processing
    *   .
    *   .
    *   .
    */

   /*
    * Close the opened Voice Channel Device
    */
   if ( dx_close( dxxxdev ) != 0 ) {
      perror( "close" );
   }

   /* Terminate the Program */
   exit( 0 );
}
```

■ Errors

None.

■ See Also

Global Tone Detection:

- "Global Tone Detection" (*Voice Features Guide for Windows NT*)

Building Tones:

- **dx_blddtcad( )**
- **dx_blddt( )**
- **dx_bldstcad( )**

Enabling Tone Detection:

- **dx_addtone( )**
- **dx_distone( )**
- **dx_enbtone( )**

R2MF Tones:

- **r2_creatfsig( )**
- **r2_playbsig( )**

Name: int dx_bldstcad(tid,freq,fqdev,ontime,ontdev,offtime, offtdev,repcnt)

Inputs:

unsigned int tid	• tone ID to assign.
unsigned int freq	• frequency in Hz
unsigned int fqdev	• frequency deviation in Hz
unsigned int ontime	• tone on time in 10ms
unsigned int ontdev	• "on" time deviation in 10ms
unsigned int offtime	• tone off time in 10ms
unsigned int offtdev	• "off" time deviation in 10ms
unsigned int repcnt	• repetitions if cadence

Returns: 0 if success
-1 if failure

Includes: srllib.h
dxxxlib.h

Category: Global Tone Detection

■ Description

The **dx_bldstcad( )** function 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.

Issuing a **dx_bldstcad( )** defines a new tone but **dx_addtone( )** must be used to add the tone to the channel.

Use **dx_addtone( )** to enable detection of the user-defined tone on a channel.

Parameter	Description
tid:	specifies a unique identifier for the tone.
	NOTE: If you are using R2MF tone detection, reserve the use of tone IDs 101 to 115 for the R2MF tones. See **r2_creatfsig( )** for further information.
freq:	specifies the frequency (in Hz) for the tone
frqdev:	specifies the allowable deviation for the frequency (in Hz).
ontime:	specifies the length of time for which the cadence is "on."

Parameter	Description
	(10 ms units)
ontdev:	specifies the allowable deviation for "on" time in 10 ms units.
offtime:	specifies the length of time for which the cadence is "off." (10 ms units)
offtdev:	specifies the allowable deviation for "off" time in 10 ms units.
repcnt:	specifies the number of repetitions for the cadence (i.e., the number of times that an on/off signal is repeated).

■ Example

```
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

#define TID_4   104

main()
{
   int  dxxxdev;

   /*
    * Open the Voice Channel Device and Enable a Handler
    */
   if ( ( dxxxdev = dx_open( "dxxxB1C1", NULL) ) == -1 ) {
      perror( "dxxxB1C1" );
      exit( 1 );
   }

   /*
    * Describe a Single Tone Frequency Tone of 950-1050 Hz.
    * On between 190-210 msecs and off 990-1010 msecs and
    * a cadence of 3.
    */
   if ( dx_bldstcad( TID_4, 1000, 50, 20, 1, 100, 1, 3 ) == -1 ) {
      printf( "Unable to build a Single Tone Cadence" );
      printf( " Template\n");
   }

   /*
    * Continue Processing
    *   .
    *   .
    *   .
    */
```

```
   /*
    * Close the opened Voice Channel Device
    */
   if ( dx_close( dxxxdev ) != 0 ) {
      perror( "close" );
   }

   /* Terminate the Program */
   exit( 0 );
}
```

■ Cautions

1. Only 1 tone per application may be defined at any time. **dx_bldstcad( )** will replace the previous user-defined tone definition.
2. If you are using R2MF tone detection, reserve the use of tone IDs 101 to 115 for the R2MF tones. See **r2_creatfsig( )** for further information.

■ Errors

None.

■ See Also

Global Tone Detection:

- "Global Tone Detection" (*Voice Features Guide for Windows NT*)

Building Tones:

- **dx_blddtcad( )**
- **dx_blddt( )**
- **dx_bldst( )**

Enabling Tone Detection:

- **dx_addtone( )**
- **dx_distone( )**
- **dx_enbtone( )**

R2MF Tones:

- **r2_creatfsig( )**

- **r2_playbsig( )**

Name: void dx_bldtngen(tngenp,freq1,freq2,ampl1,ampl2,duration)

Inputs:		
	TN_GEN *tngenp	• pointer to tone generation structure
	unsigned short freq1	• frequency of tone 1 in Hz
	unsigned short freq2	• frequency of tone 2 in Hz
	short ampl1	• amplitude of tone 1 in dB
	short ampl2	• amplitude of tone 2 in dB
	short duration	• duration of tone in 10 ms units

Returns: none

Includes: srllib.h
dxxxlib.h

Category: Global Tone Generation

■ Description

dx_bldtngen( ) is a convenience function that sets up tone generation template (*TN_GEN*) by assigning 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.

Parameter	Description
tngenp:	points to the TN_GEN data structure where the tone generation template is output.
freq1:	specifies the frequency of tone 1 in Hz. Valid range is 200 to 3000 Hz.
freq2:	specifies the frequency of tone 2 in Hz. Valid range is 200 to 3000 Hz. To define a single tone, set **freq1** to the desired frequency and set **freq2** to 0.
ampl1:	specifies the amplitude of tone 1 in dB. Valid range is 0 to -40 dB. Calling this function with **ampl1** set to R2_DEFAMPL will set the amplitude to -10 dB.
ampl2:	specifies the amplitude of tone 2 in dB. Valid range is 0 to -40 dB. Calling this function with **ampl2** set to R2_DEFAMPL will set the amplitude to -10 dB.
duration:	specifies the duration of the tone in 10 ms units. A value of -1 specifies infinite duration (the tone will only terminate upon an external terminating condition).

Generating a tone with a high frequency component (approximately 700 Hz or higher) will cause the amplitude of the tone to increase. The increase will be approximately 1 dB at 1000 Hz. Also, the amplitude of the tone will increase by 2 dB if an analog (loop start) device is used, such as the LSI/120 board or the analog device on a D/4xD, D/41E, D/41ESC, or D/160SC-LS board.

■ Cautions

None.

■ Example

```
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   TN_GEN          tngen;
   int      dxxxdev;

   /*
    * Open the Voice Channel Device and Enable a Handler
    */
   if ( ( dxxxdev = dx_open( "dxxxB1C1", NULL) ) == -1 ) {
      perror( "dxxxB1C1" );
      exit( 1 );
   }

   /*
    * Build a Tone Generation Template.
    * This template has Frequency1 = 1140,
    * Frequency2 = 1020, amplitute at -10dB for
    * both frequencies and duration of 100 * 10 msecs.
    */
   dx_bldtngen( &tngen, 1140, 1020, -10, -10, 100 );

   /*
    * Continue Processing
    *   .
    *   .
    *   .
    */

   /*
    * Close the opened Voice Channel Device
    */
   if ( dx_close( dxxxdev ) != 0 ) {
      perror( "close" );
   }
```

```
   /* Terminate the Program */
   exit( 0 );
}
```

■ Errors

None.

■ See Also

Generating Tones:

- *TN_GEN* (*Chapter 4. Voice Data Structures and Device Parameters*)
- **dx_playtone( )**
- "Global Tone Generation" (*Voice Features Guide for Windows NT*)

R2MF functions:

- **r2_creatfsig( )**
- **r2_playbsig( )**

Name:	int dx_chgdur(tonetype, ontime, ondev, offtime, offdev)	
Inputs:	int tonetype	• tone to modify
	int ontime	• on duration
	int ondev	• ontime deviation
	int offtime	• off duration
	int offdev	• offtime deviation
Returns:	0	• success
	-1	• tone does not have cadence values
	-2	• unknown tone type
Includes:	srllib.h dxxxlib.h	
Category:	PerfectCall Call Analysis	

■ Description

The Voice Driver comes with default definitions for each of the PerfectCall Call Analysis tones, which are identified by **tonetype**. The **dx_chgdur( )** function alters standard definition of duration component.

Changing a tone definition has no immediate effect on the behavior of an application. The **dx_initcallp( )** function takes the tone definitions and uses them to initialize a channel. Once a channel is initialized, subsequent changes to the tone definitions have no effect on that channel. For these changes to take effect, **dx_deltones** must be called and then followed by calling **dx_initcallp**.

Parameter	Description
tonetype:	specifies the identifier of the tone whose definition is to be modified. It may be one of the following: • 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: Fax or modem tone • TID_FAX2: Alternate fax or modem tone • TID_RINGBK1: Ringback
ontime:	is the length of time that the tone is on

Parameter	Description
	(10 ms units).
ondev:	is the maximum permissible deviation from **ontime** (10 ms units).
offtime:	is the length of time that the tone is off (10 ms units).
offdev:	is the maximum permissible deviation from **offtime** (10 ms units).

■ Example

```
#include <stdio.h>

#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   DX_CAP  cap_s;
   int    ddd, car;
   char     *chnam, *dialstrg;

   chnam    = "dxxxB1C1";
   dialstrg = "L1234";

   /*
    *  Open channel
    */
   if ((ddd = dx_open( chnam, NULL )) == -1 ) {
      /* handle error */
   }

/*
    *  Delete any previous tones
    */
   if ( dx_deltones(ddd) < 0 ) {
      /* handle error */
   }

/*
    *  Change Enhanced call progress default local dial tone
    */
   if (dx_chgfreq( TID_DIAL_LCL, 425, 150, 0, 0 ) < 0) {
      /* handle error */
   }

   /*
    *  Change Enhanced call progress default busy cadence
    */
```

```
   if (dx_chgdur( TID_BUSY1, 550, 400, 550, 400 ) < 0) {
      /* handle error */
   }

   if (dx_chgrepcnt( TID_BUSY1, 4 ) < 0) {
      /* handle error */
   }

   /*
    * Now enable Enhanced call progress with above changed settings.
    */
   if (dx_initcallp( ddd )) {
      /* handle error */
   }

   /*
    *  Set off Hook
    */
   if ((dx_sethook( ddd, DX_OFFHOOK, EV_SYNC )) == -1) {
      /* handle error */
   }

   /*
    *  Dial
    */

   if ((car = dx_dial( ddd, dialstrg,(DX_CAP *)&cap_s, DX_CALLP|EV_SYNC))==-1) {
      /* handle error */
   }

   switch( car ) {
   case CR_NODIALTONE:
      printf(" Unable to get dial tone\n");
      break;

   case CR_BUSY:
      printf(" %s engaged\n", dialstrg );
      break;

   case CR_CNCT:
      printf(" Successful connection to %s\n", dialstrg );
      break;

   default:
      break;
   }

   /*
    *  Set on Hook
    */
   if ((dx_sethook( ddd, DX_ONHOOK, EV_SYNC )) == -1) {
      /* handle error */
   }

   dx_close( ddd );
}
```

■ Cautions

This function changes only the definition of a signal. The new definition does not apply to a channel until **dx_deltones** is called and then **dx_initcallp( )** is called on that channel.

■ See Also

- **dx_chgfreq( )**
- **dx_chgrepcnt( )**
- **dx_deltones( )**
- **dx_initcallp( )**

Name:	int dx_chgfreq(tonetype, freq1, freq1dev, freq2, freq2dev)	
Inputs:	int tonetype	• tone to modify
	int freq1	• frequency of first tone
	int freq1dev	• frequency deviation for first tone
	int freq2	• frequency of second tone
	int freq2dev	• frequency deviation of second tone
Returns:	0	• success
	-1	• failure due to bad parameter(s) for tone type
	-2	• failure due to unknown tone type
Includes:	srllib.h dxxxlib.h	
Category:	PerfectCall Call Analysis	

■ Description

The **dx_chgfreq( )** function changes the standard definition for one of the PerfectCall Call Analysis tones, identified by **tonetype**, by modifying its frequency component.

The Voice Driver comes with default definitions for each of the PerfectCall Call Analysis tones; this function alters the frequency component of one of the definitions.

PerfectCall Call Analysis supports both single-frequency and dual-frequency tones. For dual-frequency tones, the frequency and tolerance of each component may be specified independently. For single-frequency tones, specifications for the second frequency are set to zero.

Changing a tone definition has no immediate effect on the behavior of an application. The **dx_initcallp( )** function takes the tone definitions and uses them to initialize a channel. Once a channel is initialized, subsequent changes to the tone definitions have no effect on that channel. For these changes to take effect, **dx_deltones** must be called and then followed by calling **dx_initcallp**.

Parameter	Description
tonetype:	specifies the identifier of the tone whose definition is to be modified. It may be one of the following: • 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: Fax or modem tone • TID_FAX2: Alternate fax or modem tone • TID_RINGBK1: Ringback
freq1:	is the frequency of the first tone (in Hz).
freq1dev:	is the maximum permissible deviation from **freq1** (in Hz).
freq2:	is the frequency of the second tone, if any (in Hz). If there is only one frequency, **freq2** is set to zero.
freq2dev:	is the maximum permissible deviation from **freq2** (in Hz).

■ Example

```
#include <stdio.h>

#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   DX_CAP  cap_s;
   int             ddd, car;
   char     *chnam, *dialstrg;

   chnam    = "dxxxB1C1";
   dialstrg = "L1234";

   /*
    *  Open channel
    */
    if ((ddd = dx_open( chnam, NULL )) == -1 ) {
      /* handle error */
   }

/*
    *  Delete any previous tones
    */
   if ( dx_deltones(ddd) < 0 ) {
```

```
      /* handle error */
   }

/*
    *  Change Enhanced call progress default local dial tone
    */
   if (dx_chgfreq( TID_DIAL_LCL, 425, 150, 0, 0 ) < 0) {
      /* handle error */
   }

   /*
    *  Change Enhanced call progress default busy cadence
    */
   if (dx_chgdur( TID_BUSY1, 550, 400, 550, 400 ) < 0) {
      /* handle error */
   }

   if (dx_chgrepcnt( TID_BUSY1, 4 ) < 0) {
      /* handle error */
   }

   /*
    * Now enable Enhanced call progress with above changed settings.
    */
   if (dx_initcallp( ddd )) {
      /* handle error */
   }

   /*
    *  Set off Hook
    */
   if ((dx_sethook( ddd, DX_OFFHOOK, EV_SYNC )) == -1) {
      /* handle error */
   }

   /*
    *  Dial
    */
   if ((car = dx_dial( ddd, dialstrg,(DX_CAP *)&cap_s, DX_CALLP|EV_SYNC))==-1) {
      /* handle error */
   }

   switch( car ) {
   case CR_NODIALTONE:
      printf(" Unable to get dial tone\n");
      break;

   case CR_BUSY:
      printf(" %s engaged\n", dialstrg );
      break;

   case CR_CNCT:
      printf(" Successful connection to %s\n", dialstrg );
      break;

   default:
      break;
   }
```

```
   /*
    *  Set on Hook
    */
   if ((dx_sethook( ddd, DX_ONHOOK, EV_SYNC )) == -1) {
      /* handle error */
   }

   dx_close( ddd );
}
```

■ Cautions

None.

■ See Also

- **dx_chgdur( )**
- **dx_chgrepcnt( )**
- **dx_deltones( )**
- **dx_initcallp( )**

Name:	int dx_chgrepcnt(tonetype, repcnt)	
Inputs:	int tonetype	• tone to modify
	int repcnt	• repetition count
Returns:	0	• success
	-1	• tone does not have a repetition value
	2	• unknown tone type
Includes:	srllib.h	
	dxxxlib.h	
Category:	PerfectCall Call Analysis	

■ Description

The **dx_chgrepcnt( )** function changes the standard definition for one of the PerfectCall Call Analysis tones, identified by **tonetype**, by modifying its repetition count component (the number of times that the signal must repeat before being recognized as valid).

The Voice Driver comes with default definitions for each of the PerfectCall Call Analysis tones; this function alters the repetition count component of one of the definitions.

Changing a tone definition has no immediate effect on the behavior of an application. The **dx_initcallp( )** function takes the tone definitions and uses them to initialize a channel. Once a channel is initialized, subsequent changes to the tone definitions have no effect on that channel. For these changes to take effect, **dx_deltones** must be called followed by calling **dx_initcallp( )**.

Parameter	Description
tonetype:	specifies the identifier of the tone whose definition is to be modified. It may be one of the following: • 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: Fax or modem tone • TID_FAX2: Alternate fax or modem tone

Parameter	Description
	• TID_RINGBK1: Ringback
repcnt:	is the number of times that the signal must repeat.

■ Example

```
#include <stdio.h>

#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   DX_CAP  cap_s;
   int    ddd, car;
   char     *chnam, *dialstrg;

   chnam    = "dxxxB1C1";
   dialstrg = "L1234";

   /*
    *  Open channel
    */
   if ((ddd = dx_open( chnam, NULL )) == -1 ) {
      /* handle error */
   }

/*
    *  Delete any previous tones
    */
   if ( dx_deltones(ddd) < 0 ) {
      /* handle error */
   }

/*
    *  Change Enhanced call progress default local dial tone
    */
   if (dx_chgfreq( TID_DIAL_LCL, 425, 150, 0, 0 ) < 0) {
      /* handle error */
   }

   /*
    *  Change Enhanced call progress default busy cadence
    */
   if (dx_chgdur( TID_BUSY1, 550, 400, 550, 400 ) < 0) {
      /* handle error */
   }

   if (dx_chgrepcnt( TID_BUSY1, 4 ) < 0) {
      /* handle error */
   }
```

```
   /*
    * Now enable Enhanced call progress with above changed settings.
    */
   if (dx_initcallp( ddd )) {
      /* handle error */
   }

   /*
    *  Set off Hook
    */
   if ((dx_sethook( ddd, DX_OFFHOOK, EV_SYNC )) == -1) {
      /* handle error */
   }

   /*
    *  Dial
    */
   if ((car = dx_dial( ddd, dialstrg,(DX_CAP *)&cap_s, DX_CALLP|EV_SYNC))==-1) {
      /* handle error */
   }

      switch( car ) {
   case CR_NODIALTONE:
      printf(" Unable to get dial tone\n");
      break;

   case CR_BUSY:
      printf(" %s engaged\n", dialstrg );
      break;

   case CR_CNCT:
      printf(" Successful connection to %s\n", dialstrg );
      break;

   default:
      break;
   }

   /*
    *  Set on Hook
    */
   if ((dx_sethook( ddd, DX_ONHOOK, EV_SYNC )) == -1) {
      /* handle error */
   }

   dx_close( ddd );
}
```

■ Cautions

This function changes only the definition of a tone. The new definition does not apply to a channel until **dx_initcallp( )** is called on that channel.

See Also

- **dx_chgdur( )**
- **dx_chgfreq( )**
- **dx_deltones( )**
- **dx_initcallp( )**

Name:	int dx_close(dev)	
Inputs:	int dev	• valid Dialogic channel or board device handle
Returns:	0 if successful -1 if error	
Includes:	srllib.h dxxxlib.h	
Category:	Device Management	

■ Description

The **dx_close( )** function closes Dialogic devices opened previously by using **dx_open( )**. It releases the handle and breaks any link the calling process has with the device through this handle. It will release the handle whether the device is busy or idle.

NOTE: **dx_close( )** disables the generation of all events. It does not affect the hookstate or any of the parameters that have been set for the device.

The function parameter is defined as follows:

Parameter	Description
dev:	specifies the valid Dialogic device handle obtained when a board or channel was opened using **dx_open( )**.

■ Cautions

Once a device is closed, a process can no longer perform any action on that device using that device handle. Other handles for that device that exist in the same process or other processes will still be valid. The only process affected by **dx_close( )** is the process that called the function.

NOTES:
1. The **dx_close( )** function doesn't affect any action occurring on a device, it only breaks the link between the calling process and the device by freeing the specified device handle. Other links through different device handles are still valid.
2. Never use the Windows NT **close( )** function to close a Voice device; unpredictable results will occur.

3. **dx_close( )** will discard any outstanding events on that handle.

Example

```
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
main()
{
   DX_CAP cap;
   int chdev;
    /* continue processing */
    if (dx_close (chdev) ==-1)
```

Errors

If this function returns -1 to indicate failure, check **errno** for one of the following reasons:

EINVAL	• Invalid Argument
EBADF	• Invalid file descriptor
EINTR	• A signal was caught

Name:	void dx_clrcap(capp)	
Inputs:	DX_CAP *capp	• pointer to Call Analysis Parameter Structure
Returns:	None	
Includes:	srllib.h dxxxlib.h	
Category:	Structure Clearance	

■ Description

The **dx_clrcap( )** function clears all the fields in a DX_CAP structure by setting them to zero. **dx_clrcap( )** is a VOID function that returns no value. It is provided as a convenient way of clearing a DX_CAP structure.

The function parameter is defined as follows:

Parameter	Description
capp:	points to the DX_CAP structure. See *Chapter 4. Voice Data Structures and Device Parameters* for information about the DX_CAP structure.

■ Cautions

The DX_CAP structure should be cleared and using **dx_clrcap( )** before the structure is used as an argument in a **dx_dial( )** function call. This will prevent parameters from being set unintentionally.

■ Example

```
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   DX_CAP cap;
   int chdev;

   /* open the channel using dx_open */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
     /* process error */
```

```
   }
   .
   .
   /* set call analysis parameters before doing call analysis */
   dx_clrcap(&cap);
   cap.ca_nbrdna = 5;  /* 5 rings before no answer */
   .
   .
   /* continue with call analysis */
   .
   .
}
```

Errors

None.

See Also

- **dx_dial( )**
- DX_CAP (*Chapter 4. Voice Data Structures and Device Parameters*)
- "Call Analysis" (*Voice Features Guide for Windows NT*)

Name:	int dx_clrdigbuf(chdev)	
Inputs:	int chdev	• valid Dialogic channel device handle
Returns:	0 if success -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	Configuration	

■ Description

The **dx_clrdigbuf( )** function causes the digits present in the firmware digit buffer of the channel specified by **chdev** to be flushed.

The function parameter is defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.

■ Cautions

None.

■ Example

```
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()

{
   int chdev;      /* channel descriptor */
   .
   .
   .
   /* Open Channel */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
     /* process error */
   }

   /* Clear digit buffer */
```

```
   if (dx_clrdigbuf(chdev) == -1) {
     /* process error*/
   }
   .
   .
}
```

See the function references for **dx_getdig( )**, **dx_play( )**, and **dx_rec( )** for more examples of how to use **dx_clrdigbuf( )**.

■ Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	• Invalid Parameter
EDX_SYSTEM	• Windows NT system error - check **errno**

Name:	int dx_clrsvcond(chdev)
Inputs:	int chdev • valid Dialogic channel device handle
Returns:	0 if success -1 if failure
Includes:	srllib.h dxxxlib.h
Category:	Speed and Volume

■ Description

The **dx_clrsvcond( )** function clears any speed or volume adjustment conditions that have been previously set with the **dx_setsvcond( )** function or the convenience functions **dx_addspddig( )** or **dx_addvoldig( )**.

Each time you want to reset a single adjustment condition, you must reset all adjustment conditions, by first clearing them using this function, and then resetting the conditions using **dx_setsvcond( )**, **dx_addspddig( )** or **dx_addvoldig( )**.

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.

■ Cautions

None.

■ Example

```
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   int  dxxxdev;

   /*
    * Open the Voice Channel Device and Enable a Handler
    */
   if ( ( dxxxdev = dx_open( "dxxxB1C1", NULL) ) == -1 ) {
```

```
      perror( "dxxxB1C1" );
      exit( 1 );
   }

   /*
    * Clear all Speed and Volume Conditions
    */
   if ( dx_clrsvcond( dxxxdev ) == -1 ) {
      printf( "Unable to Clear the Speed/Volume" );
      printf( " Conditions\n" );
      printf( "Lasterror = %d  Err Msg = %s\n",
         ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
      dx_close( dxxxdev );
      exit( 1 );
   }

   /*
    * Continue Processing
    *   .
    *   .
    *   .
    */

   /*
    * Close the opened Voice Channel Device
    */
   if ( dx_close( dxxxdev ) != 0 ) {
      perror( "close" );
   }

   /* Terminate the Program */
   exit( 0 );
}
```

■ Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	• Invalid Parameter
EDX_BADPROD	• Function not supported on this board
EDX_SYSTEM	• Windows NT system error - check **errno**

■ See Also

- **dx_setsvcond( )**
- **dx_addspddig( )**
- **dx_addvoldig( )**
- "Speed and Volume Modification Tables" (*Voice Features Guide for Windows NT*)

- *DX_SVCB* (*Chapter 4. Voice Data Structures and Device Parameters*)

Name:	int dx_clrtpt(tptp,size)	
Inputs:	DV_TPT *tptp	• pointer to termination parameter table structure
	int size	• number of entries to clear
Returns:	0 if success -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	Structure Clearance	

■ Description

The **dx_clrtpt( )** function clears all DV_TPT fields except **tp_type** and **tp_nextp** in the number of DV_TPT structures indicated in the **size** parameter. **dx_clrtpt( )** is provided as a convenient way of clearing a DV_TPT structure, if this is required before initializing it for a new set of terminating conditions.

NOTE: The DV_TPT is defined in *srllib.h* since it can be used by other non-Voice devices. Valid Voice values for the DV_TPT are listed in *Appendix* A, and the DV_TPT structure is described in detail in the *Standard Runtime Library Programmer's Guide.*

Prior to calling **dx_clrtpt( )**, you must set the **tp_type** field of DV_TPT as follows:

IO_CONT	• if the next DV_TPT is contiguous
IO_LINK	• if the next DV_TPT is linked
IO_EOT	• for the last DV_TPT

If **tp_type** is set to IO_LINK, you *MUST* set **tp_nextp** to point to the next DV_TPT in the chain. **dx_clrtpt( )** uses the information in **tp_type**, and in tp_nextp if IO_LINK is set, to access the next DV_TPT. By setting the **tp_type** and **tp_nextp** fields appropriately, **dx_clrtpt( )** can be used to clear a combination of contiguous and linked DV_TPT structures.

The function parameters are defined as follows:

Parameter	Description
tptp:	points to the first DV_TPT to be cleared. See *Appendix* A for information about the DV_TPT.
size:	indicates the number of DV_TPT structures to clear. If **size** is set to 0, the function will return a 0 to indicate success.

■ Cautions

dx_clrtpt( ) uses the information present in **tp_type** and **tp_nextp** (if IO_LINK is set) to access the next DV_TPT in the chain. The last DV_TPT in the chain must have its **tp_type** field set to IO_EOT. If the DV_TPTs have to be reinitialized with a new set of conditions, **dx_clrtpt( )** must be called only *after* the links have been set up, as illustrated below.

■ Example

```
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   DV_TPT tpt1[2];
   DV_TPT tpt2[2];

   /* Set up the links in the DV_TPTs */
   tpt1[0].tp_type = IO_CONT;
   tpt1[1].tp_type = IO_LINK;
   tpt1[1].tp_nextp = &tpt2[0];

   tpt2[0].tp_type = IO_CONT;
   tpt2[1].tp_type = IO_EOT;
   /* set up the other DV_TPT fields as required for termination */
   .
   .
   /* play a voice file, get digits, etc. */
   .
   .
   /* clear out the DV_TPT structures if required */
   dx_clrtpt(&tpt1[0],4);
   /* now set up the DV_TPT structures for the next play */
   .
   .
}
```

Errors

The function will fail and return -1 if IO_EOT is encountered in the **tp_type** field before the number of DV_TPTstructures specified in **size** have been cleared.

See Also

- DV_TPT (*Chapter 4. Voice Data Structures and Device Parameters*)

Name:	int dx_deltones(chdev)	
Inputs:	int chdev	• valid Dialogic channel device handle
Returns:	0	• Success
	-1	• Error return code
Includes:	srllib.h	
	dxxxlib.h	
Category:	Global Tone Detection	

■ Description

The **dx_deltones( )** function removes all user-defined tones previously added to a channel with **dx_addtone( )**. If no user-defined tones were previously enabled for this channel, this function has no effect.

NOTE: Calling this function deletes ALL user-defined tones defined by **dx_blddt( )**, **dx_bldst( )**, **dx_bldstcad( )**, or **dx_blddtcad( )**.

Parameter	Description
chdev:	specifies the valid Dialogic channel device handle obtained when the channel was opened using **dx_open( )**.

■ Cautions

None.

■ Example

```
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   int  dxxxdev;
   /*
    * Open the Voice Channel Device and Enable a Handler
    */
   if ( ( dxxxdev = dx_open( "dxxxB1C1", NULL) ) == -1 ) {
      perror( "dxxxB1C1" );
      exit( 1 );
   }
```

```
   /*
    * Delete all Tone Templates
    */
   if ( dx_deltones( dxxxdev ) == -1 ) {
      printf( "Unable to Delete all the Tone Templates\n" );
      printf( "Lasterror = %d  Err Msg = %s\n",
          ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
      dx_close( dxxxdev );
      exit( 1 );
   }

   /*
    * Continue Processing
    *   .
    *   .
    *   .
    */

   /*
    * Close the opened Voice Channel Device
    */
   if ( dx_close( dxxxdev ) != 0 ) {
      perror( "close" );
   }

   /* Terminate the Program */
   exit( 0 );
}
```

■ Errors

If the function returns -1 to indicate failure, call **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to return one of the following errors:

EDX_BADPARM	• Invalid parameter
EDX_BADPROD	• Function not supported on this board
EDX_SYSTEM	• Windows NT System error - check **errno**

■ See Also

Adding and Enabling User-defined Tones:

- **dx_addtone( )**
- **dx_enbtone( )**

Building Tones:

- **dx_blddt( )**
- **dx_bldst( )**
- **dx_bldstcad( )**

- **dx_blddtcad( )**

Name:	int dx_dial(chdev,dialstrp,capp,mode)	
Inputs:	int chdev	• valid Dialogic channel device handle
	char *dialstrp	• pointer to the ASCIIZ dial string
	DX_CAP *capp	• pointer to Call Analysis Parameter Structure
	unsigned short mode	• asynchronous/synchronous setting and Call Analysis flag
Returns:	0 to indicate successful initiation (Asynchronous) >=0 to indicate Call Analysis result if successful (Synchronous) -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	I/O	
Mode:	synchronous/asynchronous	

■ Description

The **dx_dial( )** function dials an ASCIIZ string on an open, idle channel and optionally enables Call Analysis to provide information about the call.

To determine the state of the channel during a dial and/or Call Analysis, use **ATDX_STATE( )**, which will return one of the following:

CS_DIAL	• dial state (with or without Call Analysis)
CS_CALL	• Call Analysis state

NOTE: **dx_dial( )** doesn't affect the hook state.

dx_dial( ) without Call Analysis enabled cannot be terminated using **dx_stopch( )**, unlike most I/O functions.

The function parameters are defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.

Parameter	Description
dialstrp:	points to the ASCII dial string. **dialstrp** must contain a null-terminated string of ASCII characters. Valid dialing and control characters are described Table 3.

Table 3 Valid Characters for Each Dialing Mode

Pulse Digit	Description	DTMF Digit	Description	MF Digit	Description
"0"-"9"		"0"-"9"		"0"-"9"	
		"*"		"*"	KP
		"#"		"#"	ST
		"a"		"a"	PST
		"b"		"b"	ST2
		"c"		"c"	ST3
		"d"		"L"	
","	pause	","	pause	"I"	
"&"	flash	"&"	flash	"X"	
"L"		"L"			
"I"		"I"			
"X"		"X"			
Change dial mode to:		**Change dial mode to:**		**Change dial mode to:**	
"P"	Pulse mode.	"P"	Pulse mode.	"P"	Pulse mode.
"T"	DTMF mode.	"T"	DTMF mode.	"T"	DTMF mode.
"M"	MF mode.	M"	MF mode.	"M"	MF mode.

The dialing mode is specified by a "T" (DTMF tone dialing), "P" (pulse dialing), or "M" (MF dialing) in the dial string. If "T", "P", or "M" is not specified in **dialstrp**, DTMF tone dialing is used.

NOTE: MF dialing is only available on systems with MF

capability such as the D/4xD board with MF support, or a D/21E, D/41E, D/41ESC, D/160SC-LS, D/240SC, D/240SC-T1, D/300SC-E1, D/320SC board.

The pause character "," and the flash character "&" are not available in MF dialing mode. To send these characters while sending a string of MF digits, switch to DTMF or pulse mode before sending "," or "&", then switch back to MF mode by sending an "M". The following string demonstrates this use:

```
M*1234T,M5678a
```

capp: points to the Call Analysis Parameter Structure, DX_CAP. This structure is described in (*Chapter 4. Voice Data Structures and Device Parameters*).

To use the default Call Analysis parameters, specify NULL in **capp** and DX_CALLP in **mode** .

The D/40 board does not have the Call Analysis feature. When using **dx_dial( )**, do not pass **capp** to D/40 channels; pass a NULL pointer and do not set **mode** to DX_CALLP.

mode specifies whether an ASCIIZ string should be dialed with or without Call Analysis enabled, and whether the function should run asynchronously or synchronously. **mode** is a bit mask that can be set to a combination of the following values:

DX_CALLP	• Enable Call Analysis.
EV_ASYNC	• Run **dx_dial( )** asynchronously.
EV_SYNC	• Run **dx_dial( )** synchronously. (default)

To run **dx_dial( )** without Call Analysis, specify only EV_ASYNC or EV_SYNC.

NOTE: If **dx_dial( )** is called on a channel that is onhook, the function will only dial digits. Call analysis will not occur

■ Asynchronous Operation

Set the **mode** field to EV_ASYNC, using a bitwise OR. When running asynchronously, the function will return 0 to indicate it has initiated successfully, and will generate one of the following termination events to indicate completion:

TDX_DIAL	• termination of dialing (without Call Analysis)
TDX_CALLP	• termination of dialing (with Call Analysis)

If asynchronous **dx_dial( )** terminates with a TDX_DIAL event, use **ATDX_TERMMSK( )** to determine the reason for termination. If **dx_dial( )** terminates with a TDX_CALLP event, use **ATDX_CPTERM( )** to determine the reason for termination. These Call Analysis termination reasons are listed under the description of Call Analysis, below.

Use the SRL Event Management functions to handle the termination event. See *Appendix* A for more information about the Event Management functions.

■ Synchronous Operation

By default, this function runs synchronously, and will return a 0 to indicate that it has completed successfully.

When synchronous dialing terminates, the function will return the Call Progress result (if Call Analysis is enabled) or 0 to indicate success (if Call Analysis isn't enabled).

■ Call Analysis

Call Analysis provides information about the call. It is enabled to run on the call after dialing completes by setting the **mode** field. The function can be set to run using default Call Analysis parameters, or by using the Call Analysis Parameter structure (DX_CAP).

Call Analysis results can be retrieved using **ATDX_CPTERM( ).**

If **dx_dial( )** is running synchronously, the Call Analysis results will also be returned by the function.

For more information about Call Analysis see the *Features Guide*.

Possible Call Analysis termination reasons are listed below:

CR_BUSY	• line was busy
CR_CEPT	• operator intercept
CR_CNCT	• call connected
CR_ERROR	• Call Analysis error
CR_FAXTONE	• fax machine or modem
CR_NOANS	• no answer
CR_NODIALTONE	• no dial tone
CR_NORB	• no ringback
CR_STOPD	• Call Analysis stopped due to **dx_stopch( )**

If Call Analysis is enabled, additional information about the call can be obtained using 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_CRTNID( )	• Returns tone identifier
ATDX_DTNFAIL( )	• Returns dial tone fail character
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
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_SHORTLOW( )	• Returns duration of shorter silence
ATDX_SIZEHI( )	• Returns duration of non-silence

■ Cautions

1. If you attempt to dial a channel in MF mode and do not have MF capabilities on that channel, DTMF tone dialing is used.

2. Issuing a **dx_stopch( )** on a channel that is dialing without Call Analysis enabled has no effect on the dial, and will return 0. The digits specified in the **dialstrp** parameter will still be dialed.
3. Issuing a **dx_stopch( )** on a channel that is dialing with Call Analysis enabled will cause the dialing to complete, but Call Analysis will not be executed. The digits specified in the dialstrp parameter will be dialed. Any Call Analysis information collected prior to the stop will be returned by Extended Attribute functions.
4. This function must be issued when the channel is idle.

■ Example 1: Call Analysis with user-specified parameters (Synchronous Mode)

```
/* Call Analysis with user-specified parameters and synchronous mode. */

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

main()
{
   int cares, chdev;
   DX_CAP capp;
   .
   .
   /* open the channel using dx_open( ).  Obtain channel device descriptor in
    * chdev
    */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1)  {
     /* process error */
   }

   /* take the phone off-hook */
   if ((dx_sethook(chdev,DX_OFFHOOK,EV_SYNC)) == -1) {
     /* process error */
   }

   /* Clear DX_CAP structure */
   dx_clrcap(&capp);

   /* Set the DX_CAP structure as needed for call analysis.
    * Allow 3 rings before no answer.
    */
   capp.ca_nbrdna = 3;

   /*  Perform the outbound dial with call analysis enabled. */
   if ((cares = dx_dial(chdev,"5551212",&capp,DX_CALLP|EV_SYNC)) == -1) {
     /* perform error routine */
   }
```

```
   switch (cares) {
     case CR_CNCT:     /* Call Connected, get some additional info */
      printf("\nDuration of short low - %ld ms",ATDX_SHORTLOW(chdev)*10);
      printf("\nDuration of long low  - %ld ms",ATDX_LONGLOW(chdev)*10);
      printf("\nDuration of answer    - %ld ms",ATDX_ANSRSIZ(chdev)*10);
      break;
     case CR_CEPT:     /* Operator Intercept detected */
      printf("\nFrequency detected - %ld Hz",ATDX_FRQHZ(chdev));
      printf("\n%% of Frequency out of bounds - %ld Hz",ATDX_FRQOUT(chdev));
      break;
     case CR_BUSY:
      .
      .
   }
   /* carry out the next state */
   .
   .
}
```

■ Example 2: Call Analysis with default parameters (Synchronous Mode)

```
/* Call Analysis with default parameters and synchronous mode. */

#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   int cares, chdev;
   DX_CAP capp;

   /* open the channel using dx_open( ).  Obtain channel device descriptor
    * in chdev
    */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
     /* process error */
   }

   /* take the phone off-hook */
   if ((dx_sethook(chdev,DX_OFFHOOK,EV_SYNC)) == -1) {
     /* process error */
   }

   /* Perform the outbound dial with call analysis enabled and capp set to
    * NULL
    */
   if ((cares = dx_dial(chdev,"5551212",(DX_CAP *)NULL,DX_CALLP|EV_SYNC)) ==
        -1) {
     /* perform error routine */
   }
   /* Analyze the call analysis results as in Example 1 */
   .
   .
}
```

Example 3: Call Analysis with default parameters (Asynchronous, Callback Mode)

```
/* Call Analysis with user-specified parameters and asynchronous, callback mode. */

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

#define MAXCHAN 24

int dial_handler();

DX_CAP capp;

main()
{
   int i, chdev[MAXCHAN];
   char *chnamep;
   int srlmode;

   /* Set SRL to run in polled mode. */
   srlmode = SR_POLLMODE;
   if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1) {
      /* process error */
   }

   for (i=0; i<MAXCHAN; i++) {

      /* Set chnamep to the channel name, e.g., dxxxB1C1, dxxxB1C2,... */

      /* Open the device using dx_open( ).  chdev[i] has channel device
       * descriptor.
       */
       if ((chdev[i] = dx_open(chnamep,NULL)) == -1)   {
          /* process error */
       }

      /* Using sr_enbhdlr(), set up handler function to handle call analysis
       * completion events on this channel.
       */
       if (sr_enbhdlr(chdev[i], TDX_CALLP, dial_handler) == -1) {
          /* process error */
       }

      /* Before issuing dx_dial(), place the phone off-hook. */

      /* Clear DX_CAP structure */
      dx_clrcap(&capp);

      /* Set the DX_CAP structure as needed for call analysis.
       * Allow 3 rings before no answer.
       */
      capp.ca_nbrdna = 3;
```

```
      /*  Perform the outbound dial with call analysis enabled. */
       if (dx_dial(chdev[i],"5551212",&capp,DX_CALLP|EV_ASYNC) == -1) {
          /* perform error routine */
       }

       /* Use sr_waitevt() to wait for the completion of call analysis.
        * On receiving the completion event,TDX_CALLP,control is transferred
        * to the handler function previously established using sr_enbhdlr().
        */
         .
         .

   }
}

int dial_handler()
{
   int chdev;

   chdev = sr_getevtdev();
   switch (ATDX_CPTERM(chdev)) {
     case CR_CNCT:  /* Call Connected, get some additional info */
      printf("\nDuration of short low - %ld ms",ATDX_SHORTLOW(chdev)*10);
      printf("\nDuration of long low  - %ld ms",ATDX_LONGLOW(chdev)*10);
      printf("\nDuration of answer    - %ld ms",ATDX_ANSRSIZ(chdev)*10);
      break;
     case CR_CEPT:     /* Operator Intercept detected */
      printf("\nFrequency detected - %ld Hz",ATDX_FRQHZ(chdev));
      printf("\n%% of Frequency out of bounds - %ld Hz",ATDX_FRQOUT(chdev));
      break;
     case CR_BUSY:
       .
      .
   }

   /* Kick off next function in the state machine model. */
   .
   .

   return 0;
}
```

■ Example 4: PerfectCall Call Analysis (Synchronous Mode)

```
#include <stdio.h>

#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   DX_CAP  cap_s;
   int            ddd, car;
   char     *chnam, *dialstrg;

   chnam    = "dxxxB1C1";
```

```
   dialstrg = "L1234";
   /*
    *  Open channel
    */
   if ((ddd = dx_open( chnam, NULL )) == -1 ) {
      /* handle error */
   }

/*
    *  Delete any previous tones
    */
   if ( dx_deltones(ddd) < 0 ) {
      /* handle error */
   }

/*
    *  Change Enhanced call progress default local dial tone
    */
   if (dx_chgfreq( TID_DIAL_LCL, 425, 150, 0, 0 ) < 0) {
      /* handle error */
   }

   /*
    *  Change Enhanced call progress default busy cadence
    */
   if (dx_chgdur( TID_BUSY1, 550, 400, 550, 400 ) < 0) {
      /* handle error */
   }

   if (dx_chgrepcnt( TID_BUSY1, 4 ) < 0) {
      /* handle error */
   }

   /*
    * Now enable Enhanced call progress with above changed settings.
    */
   if (dx_initcallp( ddd )) {
      /* handle error */
   }

   /*
    *  Set off Hook
    */
   if ((dx_sethook( ddd, DX_OFFHOOK, EV_SYNC )) == -1) {
      /* handle error */
   }

   /*
    *  Dial
    */
   if ((car = dx_dial( ddd, dialstrg,(DX_CAP *)&cap_s, DX_CALLP|EV_SYNC))==-1) {
      /* handle error */
   }

   switch( car ) {
   case CR_NODIALTONE:
      printf(" Unable to get dial tone\n");
      break;

   case CR_BUSY:
```

```
      printf(" %s engaged\n", dialstrg );
      break;

   case CR_CNCT:
      printf(" Successful connection to %s\n", dialstrg );
      break;

   default:
      break;
   }

   /*
    *  Set on Hook
    */
   if ((dx_sethook( ddd, DX_ONHOOK, EV_SYNC )) == -1) {
      /* handle error */
   }

   dx_close( ddd );
}
```

■ Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	• Invalid Parameter
EDX_BUSY	• Channel is busy
EDX_SYSTEM	• Windows NT system error - check **errno**

■ See Also

- **dx_stopch( )**

Retrieving termination reasons and events for dx_dial( ) with Call Analysis:

- Event Management functions (*Standard Runtime Library Programmer's Guide for Windows NT*)
- **ATDX_CPTERM( )**

Retrieving termination reasons for dx_dial( ) without Call Analysis:

- **ATDX_TERMMSK( )**

Call Analysis:

- DX_CAP (*Chapter 4. Voice Data Structures and Device Parameters*)
- "Call Analysis" (*Voice Features Guide for Windows NT*)
- **ATDX_ANSRSIZ( )**
- **ATDX_CPERROR**
- **ATDX_FRQDUR*( )**
- **ATDX_FRQHZ*( )**
- **ATDX_FRQOUT( )**
- **ATDX_LONGLOW( )**
- **ATDX_SHORTLOW( )**
- **ATDX_SIZEHI( )**

Name: int dx_distone(chdev,toneid,evt_mask)

Inputs:	int chdev	• channel device
	int toneid	• tone template identification
	int evt_mask	• event mask
Returns:	=0	• Success
	-1	• Error return code
Category:	Global Tone Detection	

■ Description

The **dx_distone( )** function disables detection of TONE ON and/or TONE OFF for a user-defined tone on a channel. Detection capability for user-defined tones is enabled on a channel by default when **dx_addtone( )** is called.

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.
toneid:	specifies the user-defined tone identifier for which detection is being disabled. To disable detection of all user-defined tones on the channel, set **toneid** to TONEALL.
evt_mask:	specifies whether to disable detection of the user-defined tone going on or going off. Set to one or both of the following using a bitwise-OR (|) operator. • DM_TONEON disable TONE ON detection • DM_TONEOFF disable TONE OFF detection **evt_mask** affects the enabled/disabled status of the tone template and will remain in effect until **dx_distone( )** or **dx_enbtone( )** is called again to reset it.

■ Example

```
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
```

```
#define TID_1   101

main()
{
   int  dxxxdev;

   /*
    * Open the Voice Channel Device and Enable a Handler
    */
   if ( ( dxxxdev = dx_open( "dxxxB1C1", NULL) ) == -1 ) {
      perror( "dxxxB1C1" );
      exit( 1 );
   }

   /*
    * Describe a Simple Dual Tone Frequency Tone of 950-
    * 1050 Hz and 475-525 Hz using leading edge detection.
    */
   if ( dx_bldt( TID_1, 1000, 50, 500, 25, TN_LEADING ) == -1 ) {
      printf( "Unable to build a Dual Tone Template\n" );
   }

   /*
    * Bind the Tone to the Channel
    */
   if ( dx_addtone( dxxxdev, NULL, 0 ) == -1 ) {
      printf( "Unable to Bind the Tone %d\n", TID_1 );
      printf( "Lasterror = %d  Err Msg = %s\n",
          ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
      dx_close( dxxxdev );
      exit( 1 );
   }

   /*
    * Disable Detection of ToneId TID_1
    */
   if ( dx_distone( dxxxdev, TID_1, DM_TONEON | DM_TONEOFF ) == -1 ) {
      printf( "Unable to Disable Detection of Tone %d\n", TID_1 );
      printf( "Lasterror = %d  Err Msg = %s\n",
          ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
      dx_close( dxxxdev );
      exit( 1 );
   }

   /*
    * Continue Processing
    *   .
    *   .
    *   .
    */

   /*
    * Close the opened Voice Channel Device
    */
   if ( dx_close( dxxxdev ) != 0 ) {
      perror( "close" );
   }

   /* Terminate the Program */
   exit( 0 );
```

```
}
```

Errors

If the function returns -1 to indicate failure, call **ATDX_LASTERR( )** and **ATDV_ERRMSGP( )** to return one of the following errors:

EDX_BADPARM	• Invalid parameter
EDX_BADPROD	• Function not supported on this board
EDX_SYSTEM	• Windows NT System error - check **errno**
EDX_TNMSGSTATUS	• Invalid message status setting
EDX_TONEID	• Bad tone ID

See Also

Global Tone Detection functions:

- **dx_addtone( )**
- **dx_blddt( ), dx_bldst( ), dx_blddtcad( ), dx_bldstcad( )**
- **dx_enbtone( )**
- "Global Tone Detection" (*Voice Features Guide for Windows NT*)

Event Retrieval:

- **dx_getevt( )**
- *DX_CST* data structure
- **sr_getevtdatap( )** (in the *Standard Runtime Library Programmer's Guide for Windows NT*)

Name:	int dx_enbtone(chdev,toneid,evt_mask)	
Inputs:	int chdev	• valid Dialogic channel device handle
	int toneid	• tone template identification
	int evt_mask	• event mask
Returns:	0:	• Success
	-1	• Error return code
Category:	Global Tone Detection	

■ Description

The **dx_enbtone( )** function enables detection of TONE ON and/or TONE OFF for a user-defined tone on a channel. Detection capability for tones is enabled on a channel by default when **dx_addtone( )** is called.

The description of **dx_addtone( )** (earlier in this chapter) explains how to synchronously and asynchronously retrieve CST tone on and tone off events.

Use this function to enable a tone that has been disabled using **dx_distone( )**.

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using dx_open( ).
toneid:	specifies the user-defined tone identifier for which detection is being enabled. To enable detection of all user-defined tones on the channel, set toneid to TONEALL.
evt_mask:	specifies whether to enable detection of the user-defined tone going on or going off. Set to one or both of the

Parameter	Description	
	following using a bitwise -OR (|) operator.	
	• DM_TONEON	disable TONE ON detection
	• DM_TONEOFF	disable TONE OFF detection
	evt_mask affects the enabled/disabled status of the tone template and will remain in effect until dx_enbtone( ) or dx_distone( ) is called again to reset it.	

■ Example

```
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

#define TID_1   101

main()
{
   int  dxxxdev;

   /*
    * Open the Voice Channel Device and Enable a Handler
    */
   if ( ( dxxxdev = dx_open( "dxxxB1C1", NULL) ) == -1 ) {
      perror( "dxxxB1C1" );
      exit( 1 );
   }

   /*
    * Describe a Simple Dual Tone Frequency Tone of 950-
    * 1050 Hz and 475-525 Hz using leading edge detection.
    */
   if ( dx_blddt( TID_1, 1000, 50, 500, 25, TN_LEADING ) == -1 ) {
      printf( "Unable to build a Dual Tone Template\n" );
   }

   /*
    * Bind the Tone to the Channel
    */
```

```
   if ( dx_addtone( dxxxdev, NULL, 0 ) == -1 ) {
      printf( "Unable to Bind the Tone %d\n", TID_1 );
      printf( "Lasterror = %d  Err Msg = %s\n",
        ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
      dx_close( dxxxdev );
      exit( 1 );
   }

/*
    * Enable Detection of ToneId TID_1
    */
   if ( dx_enbtone( dxxxdev, TID_1, DM_TONEON | DM_TONEOFF ) == -1 ) {
      printf( "Unable to Enable Detection of Tone %d\n", TID_1 );
      printf( "Lasterror = %d  Err Msg = %s\n",
          ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
      dx_close( dxxxdev );
      exit( 1 );
   }

   /*
    * Continue Processing
    *   .
    *   .
    *   .
    */

   /*
    * Close the opened Voice Channel Device
    */
   if ( dx_close( dxxxdev ) != 0 ) {
      perror( "close" );
   }

   /* Terminate the Program */
   exit( 0 );
}
```

■ Cautions

None.

■ Errors

If the function returns -1 to indicate failure, call **ATDX_LASTERR( )** and **ATDV_ERRMSGP( )** to return one of the following errors:

EDX_BADPARM	• Invalid parameter
EDX_BADPROD	• Function not supported on this board
EDX_SYSTEM	• Windows NT System error - check **errno**
EDX_TONEID	• Bad tone ID
EDX_TNMSGSTATUS	• Invalid message status setting

■ See Also

Global Tone Detection:

- **dx_addtone( )**
- **dx_blddt( ), dx_bldst( ), dx_blddtcad( ), dx_bldstcad( )**
- **dx_distone( )**
- "Global Tone Detection" (*Voice Features Guide for Windows NT*)

Event Retrieval:

- **dx_getevt( )**
- *DX_CST* data structure
- **sr_getevtdatap( )** (in the *Standard Runtime Library Programmer's Guide for Windows NT*)

Name:	int dx_fileclose(handle)	
Inputs:	int handle	• handle returned from **dx_fileopen( )**
Returns:	0 if success -1 if failure	
Category:	File Management	

■ Description

The **dx_fileclose( )** function closes the file associated with the handle returned by the dx_fileopen( ) function. See the **_close** function in the *Microsoft Visual C++ Run-Time Library Reference* for more information.

■ Cautions

Use **dx_fileclose( )** instead of **_close** to ensure the compatibility of applications with the libraries across various versions of Visual C++.

■ Example

```
/* Play a voice file. Terminate on receiving 4 digits or at end of file
 */
#include <fcntl.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
main()
{
   int chdev;
   DX_IOTT iott;
   DV_TPT  tpt;
   DV_DIGIT dig;
   .
   .
   /* Open the device using dx_open( ). Get channel device descriptor in
    * chdev.
    */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
     /* process error */
   }
   /* set up DX_IOTT */
   iott.io_type = IO_DEV|IO_EOT;
   iott.io_bufp = 0;
   iott.io_offset = 0;
   iott.io_length = -1;  /* play till end of file */
   if((iott.io_handle = dx_fileopen("prompt.vox",
         O_RDONLY|O_BINARY)) == -1)  {
     /* process error */
   }
```

```
   /* set up DV_TPT */
   dx_clrtpt(&tpt,1);
   tpt.tp_type   = IO_EOT;          /* only entry in the table */
   tpt.tp_termno = DX_MAXDTMF;      /* Maximum digits */
   tpt.tp_length = 4;               /* terminate on four digits */
   tpt.tp_flags  = TF_MAXDTMF;      /* Use the default flags */
   /* clear previously entered digits */
   if (dx_clrdigbuf(chdev) == -1)  {
     /* process error */
   }
   /* Now play the file */
   if (dx_play(chdev,&iott,&tpt,EV_SYNC) == -1) {
     /* process error */
   }
   /* get digit using dx_getdig( ) and continue processing. */
   .
   .
   if (dx_fileclose(iott.io_handle) == -1) {
     /* process error */
   }
}
```

■ Errors

If this function returns -1 to indicate failure, **errno** is set to **EBADF** to indicate an invalid file-handle parameter.

■ See Also

- **dx_fileopen( )**
- **dx_fileseek( )**
- **dx_fileread( )**
- **dx_filewrite( )**

Name:	int dx_fileopen(filep, flags, pmode)	
Inputs:	const char *filep	• filename
	int flags	• type of operations allowed
	int pmode	• permission mode
Returns:	file handle if success	
	-1 if failure	
Category:	File Management	

■ Description

The **dx_fileopen( )** function opens the file specified by filep and prepares the file for reading and writing, as specified by *flags*. See the **_open** function in the *Microsoft Visual C++ Run-Time Library Reference* for more information.

■ Cautions

Use **dx_fileopen( )** instead of **_open** to ensure the compatibility of applications with the libraries across various versions of Visual C++.

■ Example

```
/* Play a voice file. Terminate on receiving 4 digits or at end of file*/
#include <fcntl.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
main()
{
   int chdev;
   DX_IOTT iott;
   DV_TPT  tpt;
   DV_DIGIT dig;
   .
   .
   /* Open the device using dx_open( ). Get channel device descriptor in
    * chdev.
    */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
     /* process error */
   }
   /* set up DX_IOTT */
   iott.io_type = IO_DEV|IO_EOT;
   iott.io_bufp = 0;
   iott.io_offset = 0;
   iott.io_length = -1;  /* play till end of file */
   if((iott.io_handle = dx_fileopen("prompt.vox", O_RDONLY|O_BINARY)) == -1)  {
     /* process error */
   }
```

```
   /* set up DV_TPT */
   dx_clrtpt(&tpt,1);
   tpt.tp_type   = IO_EOT;          /* only entry in the table */
   tpt.tp_termno = DX_MAXDTMF;      /* Maximum digits */
   tpt.tp_length = 4;               /* terminate on four digits */
   tpt.tp_flags  = TF_MAXDTMF;      /* Use the default flags */
   /* clear previously entered digits */
   if (dx_clrdigbuf(chdev) == -1)  {
     /* process error */
   }
   /* Now play the file */
   if (dx_play(chdev,&iott,&tpt,EV_SYNC) == -1) {
     /* process error */
   }
   /* get digit using dx_getdig( ) and continue processing. */
   .
   .
   if (dx_fileclose(iott.io_handle) == -1) {
     /* process error */
   }
}
```

■ Errors

If this function returns -1 to indicate failure, **errno** is set to one of the following values:

EACCES	Tried to open read only file for writing, file's sharing mode does not allow specified operations, or given path is directory
EEXIST	_O_CREAT and _O_EXCL flags specified, but filename already exists
EINVAL	Invalid *flags* or *pmode* argument
EMFILE	No more file handles available (too many files open)
ENOENT	File or path not found

■ See Also

- **dx_fileclose( )**
- **dx_fileseek( )**
- **dx_fileread( )**
- **dx_filewrite( )**

Name:	int dx_fileread(handle, buffer, count)	
Inputs:	int handle	• handle returned from **dx_fileopen( )**
	void *buffer	• storage location for data
	unsigned int count	• maximum number of bytes
Returns:	number of bytes if success -1 if failure	
Category:	File Management	

■ Description

The **dx_fileread( )** function turns number of bytes read by application. The function will read the number of bytes from the file associated with the handle into the buffer. The number of bytes read may be less than the value of *count* if there are fewer than *count* bytes left in the file or if the file was opened in text mode. See the **_read** function in the *Microsoft Visual C++ Run-Time Library Reference* for more information.

■ Cautions

Use **dx_fileread( )** instead of **_read** to ensure the compatibility of applications with the libraries across various versions of Visual C++.

■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
int cd;                 /* channel descriptor */
DX_UIO myio;            /* user definable I/O structure */
/*
 * User defined I/O functions
 */
int my_read(fd,ptr,cnt)
int fd;
char * ptr;
unsigned cnt;
{
   printf("My read\n");
   return(dx_fileread(fd,ptr,cnt));
}
/*
 * my write function
 */
int my_write(fd,ptr,cnt)
```

```
int fd;
char * ptr;
unsigned cnt;
{
printf("My write \n");
   return(dx_filewrite(fd,ptr,cnt));
}
/*
 * my seek function
 */
long my_seek(fd,offset,whence)
int fd;
long offset;
int whence;
{
   printf("My seek\n");
   return(dx_fileseek(fd,offset,whence));
}
void main(argc,argv)
int argc;
char *argv[];
{
   .
   . /* Other initialization */
   .
   DX_UIO uioblk;
/* Initialize the UIO structure */
uioblk.u_read=my_read;
uioblk.u_write=my_write;
uioblk.u_seek=my_seek;
/* Install my I/O routines */
dx_setuio(devhandle,uioblk);
vodat_fd = dx_fileopen("JUNK.VOX",O_RDWR|O_BINARY);
/*This block uses standard I/O functions */
iott->io_type = IO_DEV|IO_CONT
iott->io_fhandle = vodat_fd;
iott->io_offset = 0;
iott->io_length = 20000;
/*This block uses my I/O functions */
iottp++;
iottp->io_type = IO_DEV|IO_UIO|IO_CONT
iottp->io_fhandle = vodat_fd;
iott->io_offset = 20001;
iott->io_length = 20000;
/*This block uses standard I/O functions */
iottp++
iott->io_type = IO_DEV|IO_CONT
iott->io_fhandle = vodat_fd;
iott->io_offset = 20002;
iott->io_length = 20000;
/*This block uses my I/O functions */
iott->io_type = IO_DEV|IO_UIO|IO_EOT
iott->io_fhandle = vodat_fd;
iott->io_offset = 10003;
iott->io_length = 20000;
devhandle = dx_open("dxxxB1C1", 0);
dx_sethook(devhandle, DX-ONHOOK,EV_SYNC)
dx_wtring(devhandle,1,DX_OFFHOOK,EV_SYNC);
dx_clrdigbuf;
   if(dx_rec(devhandle,iott,(DX_TPT*)NULL,RM_TONE|EV_SYNC) == -1) {
  perror("");
  exit(1);
}
dx_clrdigbuf(devhandle);
   if(dx_play(devhandle,iott,(DX_TPT*)EV_SYNC) == -1 {
```

```
perror("");
exit(1);
}
dx_close(devhandle);
```

Errors

If this function returns -1 to indicate failure, **errno** is set to **EBADF** which indicates an invalid file-handle parameter, a closed file, or a locked file.

See Also

- **dx_fileopen( )**
- **dx_fileclose( )**
- **dx_fileseek( )**
- **dx_filewrite( )**

Name:	long dx_fileseek(handle, offset, origin)	
Inputs:	int handle	• handle returned from **dx_fileopen( )**
	long offset	• number of bytes from the origin
	int origin	• initial position
Returns:	number of bytes read if success -1 if failure	
Category:	File Management	

■ Description

The **dx_fileseek( )** function moves file pointer associated with handle to a new location that is *offset* bytes from *origin*. The function returns the offset, in bytes, of the new position from the beginning of the file. See the **_lseek** function in the *Microsoft Visual C++ Run-Time Library Reference* for more information.

■ Cautions

Use **dx_fileseek( )** instead of **_lseek** to ensure the compatibility of applications with the libraries across various versions of Visual C++.

■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
int cd;                 /* channel descriptor */
DX_UIO myio;            /* user definable I/O structure */
/*
 * User defined I/O functions
 */
int my_read(fd,ptr,cnt)
int fd;
char * ptr;
unsigned cnt;
{
   printf("My read\n");
   return(dx_fileread(fd,ptr,cnt));
}
/*
 * my write function
 */
int my_write(fd,ptr,cnt)
int fd;
```

```
char * ptr;
unsigned cnt;
{
printf("My write \n");
   return(dx_filewrite(fd,ptr,cnt));
}
/*
 * my seek function
 */
long my_seek(fd,offset,whence)
int fd;
long offset;
int whence;
{
   printf("My seek\n");
   return(dx_fileseek(fd,offset,whence));
}
void main(argc,argv)
int argc;
char *argv[];
{
   .
   . /* Other initialization */
   .
   DX_UIO uioblk;
/* Initialize the UIO structure */
uioblk.u_read=my_read;
uioblk.u_write=my_write;
uioblk.u_seek=my_seek;
/* Install my I/O routines */
dx_setuio(devhandle,uioblk);
vodat_fd = dx_fileopen("JUNK.VOX",O_RDWR|O_BINARY);
/*This block uses standard I/O functions */
iott->io_type = IO_DEV|IO_CONT
iott->io_fhandle = vodat_fd;
iott->io_offset = 0;
iott->io_length = 20000;
/*This block uses my I/O functions */
iottp++;
iottp->io_type = IO_DEV|IO_UIO|IO_CONT
iottp->io_fhandle = vodat_fd;
iott->io_offset = 20001;
iott->io_length = 20000;
/*This block uses standard I/O functions */
iottp++
iott->io_type = IO_DEV|IO_CONT
iott->io_fhandle = vodat_fd;
iott->io_offset = 20002;
iott->io_length = 20000;
/*This block uses my I/O functions */
iott->io_type = IO_DEV|IO_UIO|IO_EOT
iott->io_fhandle = vodat_fd;
iott->io_offset = 10003;
iott->io_length = 20000;
devhandle = dx_open("dxxxB1C1", NULL);
dx_sethook(devhandle, DX-ONHOOK,EV_SYNC)
dx_wtring(devhandle,1,DX_OFFHOOK,EV_SYNC);
dx_clrdigbuf;
   if(dx_rec(devhandle,iott,(DX_TPT*)NULL,RM_TONE|EV_SYNC) == -1) {
  perror("");
  exit(1);
}
dx_clrdigbuf(devhandle);
   if(dx_play(devhandle,iott,(DX_TPT*)EV_SYNC) == -1 {
   perror("");
```

```
exit(1);
}
dx_close(devhandle);
```

Errors

If this function returns -1 to indicate failure, errno is set to the following values:

EBADF	• Invalid file-handle parameter, a closed file, or a locked file.
EINVAL	• Value for *origin* is invalid or the position specified by *offset* is before the beginning of the file

On devices incapable of seeking, the return value is undefined.

See Also

- **dx_fileopen( )**
- **dx_fileclose( )**
- **dx_fileread( )**
- **dx_filewrite( )**

Name:	int dx_filewrite(handle, buffer, count)	
Inputs:	int handle	• handle returned from **dx_fileopen( )**
	void *buffer	• data to be written
	unsigned int count	• number of bytes
Returns:	number of bytes if success -1 if failure	
Category:	File Management	

■ Description

The **dx_filewrite( )** function writes count byes from buffer into file associated with handle. The write operation begins at the current position of the file pointer (if any) associated with the given file. If the file was opened for appending, the operation begins at the current end of the file. After the write operation, the file pointer is increased by the number of bytes actually written. See the **_write** function in the *Microsoft Visual C++ Run-Time Library Reference* for more information.

■ Cautions

Use **dx_filewrite( )** instead of **_write** to ensure the compatibility of applications with the libraries across various versions of Visual C++.

■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
int cd;                 /* channel descriptor */
DX_UIO myio;            /* user definable I/O structure */
/*
 * User defined I/O functions
 */
int my_read(fd,ptr,cnt)
int fd;
char * ptr;
unsigned cnt;
{
   printf("My read\n");
   return(dx_fileread(fd,ptr,cnt));
}
/*
 * my write function
```

```
 */
int my_write(fd,ptr,cnt)
int fd;
char * ptr;
unsigned cnt;
{
printf("My write \n");
   return(dx_filewrite(fd,ptr,cnt));
}
/*
 * my seek function
 */
long my_seek(fd,offset,whence)
int fd;
long offset;
int whence;
{
   printf("My seek\n");
   return(dx_fileseek(fd,offset,whence));
}
void main(argc,argv)
int argc;
char *argv[];
{
   .
   . /* Other initialization */
   .
   DX_UIO uioblk;
/* Initialize the UIO structure */
uioblk.u_read=my_read;
uioblk.u_write=my_write;
uioblk.u_seek=my_seek;
/* Install my I/O routines */
dx_setuio(devhandle,uioblk);
vodat_fd = dx_fileopen("JUNK.VOX",O_RDWR|O_BINARY);
/*This block uses standard I/O functions */
iott->io_type = IO_DEV|IO_CONT
iott->io_fhandle = vodat_fd;
iott->io_offset = 0;
iott->io_length = 20000;
/*This block uses my I/O functions */
iottp++;
iottp->io_type = IO_DEV|IO_UIO|IO_CONT
iottp->io_fhandle = vodat_fd;
iott->io_offset = 20001;
iott->io_length = 20000;
/*This block uses standard I/O functions */
iottp++
iott->io_type = IO_DEV|IO_CONT
iott->io_fhandle = vodat_fd;
iott->io_offset = 20002;
iott->io_length = 20000;
/*This block uses my I/O functions */
iott->io_type = IO_DEV|IO_UIO|IO_EOT
iott->io_fhandle = vodat_fd;
iott->io_offset = 10003;
iott->io_length = 20000;
devhandle = dx_open("dxxxB1C1", NULL);
dx_sethook(devhandle, DX-ONHOOK,EV_SYNC)
dx_wtring(devhandle,1,DX_OFFHOOK,EV_SYNC);
dx_clrdigbuf;
   if(dx_rec(devhandle,iott,(DX_TPT*)NULL,RM_TONE|EV_SYNC) == -1) {
  perror("");
  exit(1);
}
```

```
dx_clrdigbuf(devhandle);
   if(dx_play(devhandle,iott,(DX_TPT*)EV_SYNC) == -1 {
   perror("");
   exit(1);
   }
   dx_close(devhandle);
```

Errors

If this function returns -1 to indicate failure, errno is set to the following values:

EBADF	• File handle is invalid or the file is not opened for writing
ENOSPC	• Not enough space left on the device for the operation

See Also

- **dx_fileopen( )**
- **dx_fileclose( )**
- **dx_fileseek( )**
- **dx_fileread( )**

Name: int dx_getcursv(chdev,curvolp,curspeedp)

Inputs:
- int chdev — valid Dialogic channel device handle
- int * curvolp — pointer to current absolute volume setting
- int * curspeedp — pointer to current absolute speed setting

Returns: 0 if success
-1 if failure

Includes: srllib.h
dxxxlib.h

Category: Speed and Volume

■ Description

The **dx_getcursv( )** function returns the specified channel's current speed and volume adjustments active on a channel. For example, use **dx_getcursv( )** to determine the speed and volume level set interactively by a listener using DTMF digits during a play. (DTMF digits are set as play adjustment conditions using the **dx_setsvcond( )** function, or by one of the convenience functions **dx_addspddig( )** and **dx_addvoldig( )**)

Parameter	Description
chdev:	specifies the valid channel device handle obtained by a call to **dx_open( )**.
curvolp:	points to an integer that represents the current absolute volume setting for the channel. This value will lie between -30dB and +10dB.
curspeedp:	points to an integer that represents the current absolute speed setting for the channel. This value will be between -50% and +50%.

■ Cautions

None.

■ Example

```
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

/*
 * Global Variables
 */

main()
{
   int  dxxxdev;
   int  curspeed, curvolume;

   /*
    * Open the Voice Channel Device and Enable a Handler
    */
   if ( ( dxxxdev = dx_open( "dxxxB1C1", NULL) ) == -1 ) {
      perror( "dxxxB1C1" );
      exit( 1 );
   }

   /*
    * Get the Current Volume and Speed Settings
    */
   if ( dx_getcursv( dxxxdev, &curvolume, &curspeed ) == -1 ) {
      printf( "Unable to Get the Current Speed and" );
      printf( " Volume Settings\n");
      printf( "Lasterror = %d  Err Msg = %s\n",
         ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
      dx_close( dxxxdev );
      exit( 1 );
   } else {
      printf( "Volume = %d   Speed = %d\n", curvolume, curspeed );
   }

   /*
    * Continue Processing
    *   .
    *   .
    *   .
    */

     /*
    * Close the opened Voice Channel Device
    */
   if ( dx_close( dxxxdev ) != 0 ) {
      perror( "close" );
   }

   /* Terminate the Program */
   exit( 0 );
}
```

■ Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	• Invalid Parameter
EDX_BADPROD	• Function not supported on this board
EDX_SYSTEM	• Windows NT system error - check **errno**

■ See Also

Related to Speed and Volume:

- **dx_adjsv( )**
- **dx_addspddig( )**
- **dx_addvoldig( )**
- **dx_setsvmt( )**
- **dx_getsvmt( )**
- **dx_setsvcond( )**
- **dx_clrsvcond( )**
- "Speed and Volume Modification Tables" (*Voice Features Guide for Windows NT*)
- *DX_SVMT* structure (*Chapter 4. Voice Data Structures and Device Parameters*)

Name:	int dx_getdig(chdev,tptp,digitp,mode)	
Inputs:	int chdev	• valid Dialogic channel device handle
	DV_TPT *tptp	• pointer to Termination Parameter Table Structure
	DV_DIGIT *digitp	• pointer to User Digit Buffer Structure
	unsigned short mode	• asynchronous/synchronous setting
Returns:	0 to indicate successful initiation (asynchronous) number of digits (+1 for NULL) if successful (synchronous) -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	I/O	
Mode:	synchronous/asynchronous	

■ Description

The **dx_getdig( )** function initiates the collection of digits from an open channel's digit buffer. Upon termination of the function, the collected digits are written in ASCIIZ format into the local buffer, which is arranged as a *DV_DIGIT* structure.

The type of digits collected depends on the digit detection mode set by the **dx_setdigtyp( )** function (for standard Voice board digits) or by the **dx_addtone( )** function (for user-defined digits).

See the function descriptions for **dx_setdigtyp( )**, **dx_addtone( )** in this chapter and the description of the *DV_DIGIT* structure in *Chapter 4. Voice Data Structures and Device Parameters* for more information.

Set termination conditions using the DV_TPT structure. This structure is pointed to by the **tptp** parameter described below.

■ Asynchronous Operation

To run this function asynchronously set the **mode** field to EV_ASYNC. When running asynchronously, this function will return 0 to indicate it has initiated successfully, and will generate a termination event (see below) to indicate

completion. Use the SRL Event Management functions to handle the termination event. See *Appendix* A for more information about the Event Management functions.

Termination of asynchronous digit collection is indicated by a TDX_GETDIG event. After **dx_getdig( )** terminates, use the **ATDX_TERMMSK( )** function to determine the reason for termination.

■ Synchronous Operation

By default, this function runs synchronously. Termination of synchronous digit collection is indicated by a return value greater than 0 that represents the number of digits received (+1 for NULL). Use **ATDX_TERMMSK( )** to determine the reason for termination.

The function parameters are defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.
tptp:	points to the Termination Parameter Table Structure, DV_TPT, which specifies termination conditions for this function. See *Appendix* A for more information about the DV_TPT. Termination conditions are listed below: DX_DIGTYPE • User-defined digits DX_MAXDTMF • Maximum number of digits received DX_MAXSIL • Maximum silence DX_MAXNOSIL • Maximum non-silence DX_LCOFF • Loop current off DX_IDDTIME • Inter-digit delay DX_MAXTIME • Function time DX_DIGMASK • Digit mask termination DX_PMOFF • Pattern match silence off DX_PMON • Pattern match silence on DX_TONE • Tone-off or Tone-on detection
digitp:	points to the User's Digit Buffer Structure,, where collected

Parameter	Description
	digits and their types are stored in arrays. The digit types in *DV_DIGIT* can be one of the following: DG_DTMF • DTMF digit DG_LPD • Loop Pulse digit DG_MF • MF digit DG_USER1 • User-defined digit DG_USER2 • User-defined digit DG_USER3 • User-defined digit DG_USER4 • User-defined digit DG_USER5 • User-defined digit See *Chapter 4. Voice Data Structures and Device Parameters* for information about the DV_DIGIT structure. See **dx_addtone( )** for information about creating user-defined digits.
mode:	specifies whether to run **dx_getdig( )** asynchronously or synchronously. Specify one of the following: EV_ASYNC: Run **dx_getdig( )** asynchronously. EV_SYNC: Run **dx_getdig( )** synchronously (default).

The channel's digit buffer contains up to 31 digits, collected on a First-In First-Out (FIFO) basis. Since the digits remain in the channel's digit buffer until they are overwritten or cleared using **dx_clrdigbuf( )**, the digits in the channel's buffer may have been received prior to this function call. DG_MAXDIGS is the define for the maximum number of digits that can be returned by a single call to **dx_getdig( )**.

NOTE: By default, after the 31st digit, all subsequent digits will be discarded. You can use the **dx_setdigbuf( )** function with the mode parameter set to DX_DIGCYCLIC, which will cause all incoming digits to overwrite the oldest digit in the buffer. See the **dx_setdigbuf( )** function.

■ Cautions

1. Some MF digits use approximately the same frequencies as DTMF digits (see *Appendix* C). Because there is a frequency overlap, if you have the incorrect kind of detection enabled, MF digits may be mistaken for DTMF digits, and vice versa. To ensure that digits are correctly detected,

only one kind of detection should be enabled at any time. To set MF digit detection, use the **dx_setdigtyp( )** function.

2. A digit that is set to adjust play-speed or play-volume (using **dx_setsvcond( )**) will not be passed to **dx_getdig( )**, and will not be used as a terminating condition. If a digit is defined to adjust play and to terminate play, then the play adjustment will take priority.
3. When operating asynchronously, ensure that the digit buffer stays in scope for the duration of the function.
4. The channel must be idle, or the function will return an EDX_BUSY error.
5. Speed and volume control are supported on the D/21D, D/21E, D/41D, D/41E, D/41ESC, D/81A, D/121B, D/160SC-LS, D/240SC, D/240SC-T1, D/300SC-E1 and D/320SC boards only. Do not use the Speed and Volume control functions to control speed on the D/120, D/121, or D/121A boards.
6. If the function is operating synchronously and there are no digits in the buffer, the return value from this function will be 1, which indicates the NULL terminator.

■ Example 1: Using dx_getdig( ) in synchronous mode

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
main()
{
   DV_TPT tpt[3];
   DV_DIGIT digp;
   int chdev, numdigs, cnt;
   /* open the channel with dx_open( ). Obtain channel device descriptor
    * in chdev
    */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
     /* process error */
   }
   /* initiate the call */
   .
   .
   /* Set up the DV_TPT and get the digits */
   dx_clrtpt(tpt,3);
   tpt[0].tp_type   = IO_CONT;
   tpt[0].tp_termno = DX_MAXDTMF;       /* Maximum number of digits */
   tpt[0].tp_length = 4;                /* terminate on 4 digits */
   tpt[0].tp_flags  = TF_MAXDTMF;       /* terminate if already in buf. */
   tpt[1].tp_type   = IO_CONT;
   tpt[1].tp_termno = DX_LCOFF;         /* LC off termination */
   tpt[1].tp_length = 3;                /* Use 30 ms (10 ms resolution
```

```
                                        * timer) */
   tpt[1].tp_flags  = TF_LCOFF|TF_10MS; /* level triggered, clear history,
                                        * 10 ms resolution */
   tpt[2].tp_type   = IO_EOT;
   tpt[2].tp_termno = DX_MAXTIME;       /* Function Time */
   tpt[2].tp_length = 100;              /* 10 seconds (100 ms resolution
                                        * timer) */
   tpt[2].tp_flags  = TF_MAXTIME;       /* Edge-triggered */
   /* clear previously entered digits */
   if (dx_clrdigbuf(chdev) == -1) {
     /* process error */
   }
if ((numdigs = dx_getdig(chdev,tpt, &digp, EV_SYNC)) == -1) {
     /* process error */
   }
   for (cnt=0; cnt < numdigs; cnt++) {
      printf("\nDigit received = %c, digit type = %d",
                  digp.dg_value[cnt], digp.dg_type[cnt]);
   }
   /* go to next state */
   .
   .
}
```

■ Example 2: Using dx_getdig( ) in asynchronous mode

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
#define MAXCHAN 24
int digit_handler();
DV_TPT stpt[3];
DV_DIGIT digp[256];
main()
{
   int i, chdev[MAXCHAN];
   char *chnamep;
   int srlmode;
   /* Set SRL to run in polled mode. */
   srlmode = SR_POLLMODE;
   if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1) {
      /* process error */
   }
   for (i=0; i<MAXCHAN; i++) {
      /* Set chnamep to the channel name - e.g., dxxxB1C1 */
      /* open the channel with dx_open( ). Obtain channel device
       * descriptor in chdev[i]
       */
if ((chdev[i] = dx_open(chnamep,NULL)) == -1) {
          /* process error */
       }
      /* Using sr_enbhdlr(), set up handler function to handle dx_getdig()
       * completion events on this channel.
       */
       if (sr_enbhdlr(chdev[i], TDX_GETDIG, digit_handler) == -1) {
           /* process error */
       }
      /* initiate the call */
      .
      .
      /* Set up the DV_TPT and get the digits */
```

```
      dx_clrtpt(tpt,3);
      tpt[0].tp_type   = IO_CONT;
      tpt[0].tp_termno = DX_MAXDTMF;       /* Maximum number of digits */
      tpt[0].tp_length = 4;                /* terminate on 4 digits */
      tpt[0].tp_flags  = TF_MAXDTMF;       /* terminate if already in buf*/
      tpt[1].tp_type   = IO_CONT;
      tpt[1].tp_termno = DX_LCOFF;         /* LC off termination */
      tpt[1].tp_length = 3;                /* Use 30 ms (10 ms resolution
                                            * timer) */
      tpt[1].tp_flags  = TF_LCOFF|TF_10MS; /* level triggered, clear
                                            * history, 10 ms resolution */
      tpt[2].tp_type   = IO_EOT;
      tpt[2].tp_termno = DX_MAXTIME;        /* Function Time */
      tpt[2].tp_length = 100;               /* 10 seconds (100 ms resolution
                                            * timer) */
      tpt[2].tp_flags  = TF_MAXTIME;        /* Edge triggered */
     /* clear previously entered digits */
      if (dx_clrdigbuf(chdev[i]) == -1) {
         /* process error */
      }
      if (dx_getdig(chdev[i], tpt, &digp[chdev[i]], EV_ASYNC) == -1) {
         /* process error */
      }
   }
 /* Use sr_waitevt() to wait for the completion of dx_getdig().
  * On receiving the completion event, TDX_GETDIG, control is transferred
  * to the handler function previously established using sr_enbhdlr().
  */
      .
        .
}

int digit_handler()
{
   int chfd;
   int cnt, numdigs;
   chfd = sr_getevtdev();
   numdigs = strlen(digp[chfd].dg_value);
   for(cnt=0; cnt < numdigs; cnt++) {
      printf("\nDigit received = %c, digit type = %d",
                        digp[chfd].dg_value[cnt], digp[chfd].dg_type[cnt]);
   }

   /* Kick off next function in the state machine model. */
   .
   .
   return 0;
}
```

■ Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	• Invalid Parameter
EDX_BADTPT	• Invalid DV_TPT entry
EDX_BUSY	• Channel busy
EDX_SYSTEM	• Windows NT system error - check **errno**

See Also

Setting User-Defined Digits:

- **dx_addtone( )**
- **dx_setdigtyp( )**

Collecting Digits:

- *DV_DIGIT*
- **dx_sethook( )**

Name:	int dx_getevt(chdev,eblkp,timeout)	
Inputs:	int chdev	• valid Dialogic channel device handle
	DX_EBLK *eblkp	• Pointer to Event Block Structure
	int timeout	• Timeout value in seconds
Returns:	0 if success -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	Call Status Transition Event	

■ Description

The **dx_getevt( )** function is used to synchronously monitor channels for possible call status transition events in conjunction with **dx_setevtmsk( )**. **dx_getevt( )** blocks and returns control to the program after one of the events set by **dx_setevtmsk( )** occurs on the channel specified in the **chdev** parameter. The *DX_EBLK* structure contains the event that ended the blocking.

The function parameters are defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.
eblkp:	points to the Event Block Structure DX_EBLK, which will contain the event that ended the blocking.
timeout:	specifies the maximum amount of time in seconds to wait for an event to occur. timeout can have one of the following values: # of seconds: maximum length of time **dx_getevt( )** will wait for an event. When the time specified has elapsed, the function will terminate and return an error. -1: **dx_getevt( )** will block until an event occurs; it will not time out. 0: The function will return -1 immediately if no event is present.

Parameter	Description

NOTE: When the time specified in **timeout** expires, **dx_getevt( )** will terminate and return an error. The Standard Attribute function **ATDV_LASTERR( )** can be used to determine the cause of the error, which in this case is EDX_TIMEOUT.

■ Cautions

We recommend enabling only one process per channel. The event that **dx_getevt( )** is waiting for may change if another process sets a different event for that channel. See **dx_setevtmsk( )** for more details.

■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
main()
{
   int chdev;          /* channel descriptor */
   int timeout;        /* timeout for function */
   DX_EBLK eblk;       /* Event Block Structure */
   .
   .
   .
   /* Open Channel */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
     /* process error */
   }
   /* Set RINGS or WINK as events to wait on */
   if (dx_setevtmsk(chdev,DM_RINGS|DM_WINK) == -1) {
     /* process error */
   }
   /* Set timeout to 5 seconds */
   timeout = 5;
   if (dx_getevt(chdev,&eblk,timeout) == -1){
      /* process error */
      if (ATDV_LASTERR(chdev) == EDX_TIMEOUT) {    /* check if timed out */
         printf("Timed out waiting for event.\n");
      }
      else {
         /* further error processing */
         .
         .
      }
   }
   switch (eblk.ev_event) {
   case DE_RINGS:
      printf("Ring event occurred.\n");
      break;
   case DE_WINK:
```

```
      printf("Wink event occurred.\n");
      break;
   }
   .
   .
}
```

Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	• Invalid Parameter
EDX_SYSTEM	• Windows NT system error - check **errno**
EDX_TIMEOUT	• Timeout time limit is reached

See Also

- **dx_setevtmsk( )**
- *DX_EBLK* (*Chapter 4. Voice Data Structures and Device Parameters*)

Name:	int dx_getparm(dev,parm,valuep)	
Inputs:	int dev	• valid Dialogic channel or board device handle
	unsigned long parm	• parameter type to get value of
	void *valuep	• pointer to variable for returning parameter value
Returns:	0 if success -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	Configuration	

■ Description

The **dx_getparm( )** function obtains the current parameter settings for an open device. **dx_getparm( )** can only obtain the value of one parameter at a time. The channel must be idle (i.e., no I/O function running) when calling **dx_getparm( )**.

The function parameters are defined as follows:

Parameter	Description
dev:	specifies the valid Dialogic device handle obtained when a board or channel was opened using **dx_open( )**.
parm	specifies the define for the parameter type whose value is to be returned in the variable pointed to by **valuep**. Board and channel parameter defines, defaults and descriptions are listed in *Section 5.2.* Clearing Voice Structures
valuep:	points to the variable where the value of the parameter specified in **parm** should be returned.

NOTE: You must use a void* cast on the returned parameter value, as demonstrated in the example that follows.

valuep should point to a variable large enough to hold the value of the parameter. Refer to *dxxxlib.h* for parameters sizes. The size of a parameter is encoded in the define for the parameter. The defines for parameter sizes are PM_SHORT, PM_BYTE, PM_INT, PM_LONG,

Parameter	Description
	PM_FLSTR (fixed length string), and PM_VLSTR (variable length string). See *dxxxlib.h* for DXBD_ and DXCH_ defines. Most parameters are of type short.

■ Cautions

We highly recommend that you clear the variable the parameter value is returned to prior to calling **dx_getparm( )**, as illustrated in the example below. The variable whose address is passed to should be of a size sufficient to hold the value of the parameter. See the description section of this function for more information.

■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   int bddev;
   unsigned short parmval;

   /* open the board using dx_open( ). Obtain board device descriptor in
    * bddev
    */
   if ((bddev = dx_open("dxxxB1",NULL)) == -1) {
     /* process error */
   }

   parmval = 0;    /* CLEAR parmval */

   /* get the number of channels on the board. DXBD_CHNUM is of type
    * unsigned short as specified by the PM_SHORT define in the definition
    * for DXBD_CHNUM in dxxxlib.h. The size of the variable parmval is
    * sufficient to hold the value of DXBD_CHNUM.
    */
   if (dx_getparm(bddev, DXBD_CHNUM, (void *)&parmval) == -1) {
     /* process error */
   }

   printf("\nNumber of channels on board = %d",parmval);
   .
   .
}
```

Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	• Invalid Parameter
EDX_SYSTEM	• Windows NT system error - check **errno**
EDX_BUSY	• Channel is busy (when channel device handle is specified) or first channel is busy (when board device handle is specified)

See Also

- **dx_setparm( )**

Name:	int dx_getsvmt(chdev,tabletype,svmtp)	
Inputs:	int chdev	• valid Dialogic channel device handle
	unsigned short tabletype	• table to retrieve (speed or volume)
	DX_SVMT * svmtp	• pointer to DX_SVMT structure
Returns:	0 if success -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	Speed and Volume	

■ Description

The **dx_getsvmt( )** function returns contents of Speed or Volume Modification Table to the *DX_SVMT* structure.

For a full description of the Speed and Volume Modification Tables see the *Voice Features Guide for Windows NT* see a description in section *4.1.6.* DX_SVMT - *speed/volume modification table structure* of the DX_SVMT structure.

Parameter	Description	
chdev:	specifies the valid channel device handle obtained by a call to **dx_open( )**.	
tabletype:	specifies whether to retrieve the Speed or the Volume Modification Table.	
	SV_SPEEDTBL	Retrieve the Speed Modification Table values
	SV_VOLUMETBL	Retrieve the Volume Modification Table values
svmtp:	points to the *DX_SVMT* structure that contains the Speed/Volume Modification Table entries.	

■ Cautions

None.

■ Example

```
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

/*
 * Global Variables
 */

main()
{
   DX_SVMT         svmt;
   int      dxxxdev, index;

   /*
    * Open the Voice Channel Device and Enable a Handler
    */
   if ( ( dxxxdev = dx_open( "dxxxB1C1", NULL) ) == -1 ) {
      perror( "dxxxB1C1" );
      exit( 1 );
   }

   /*
    * Get the Current Volume Modification Table
    */
   memset( &svmt, 0, sizeof( DX_SVMT ) );
   if (dx_getsvmt( dxxxdev, SV_VOLUMETBL, &svmt ) == -1 ){
      printf( "Unable to Get the Current Volume" );
      printf( " Modification Table\n" );
      printf( "Lasterror = %d  Err Msg = %s\n",
         ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
      dx_close( dxxxdev );
      exit( 1 );
   } else {
      printf( "Volume Modification Table is:\n" );
      for ( index = 0; index < 10; index++ ) {
         printf( "decrease[ %d ] = %d\n", index,
                svmt.decrease[ index ] );
      }

      printf( "origin = %d\n", svmt.origin );

      for ( index = 0; index < 10; index++ ) {
         printf( "increase[ %d ] = %d\n", index,
                svmt.increase[ index ] );
      }
   }

   /*
    * Continue Processing
    *   .
    *   .
    *   .
    */

   /*
```

```
    * Close the opened Voice Channel Device
    */
   if ( dx_close( dxxxdev ) != 0 ) {
      perror( "close" );
   }

   /* Terminate the Program */
   exit( 0 );
}
```

■ Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	• Invalid Parameter
EDX_BADPROD	• Function not supported on this board
EDX_SPDVOL	• Must Specify either SV_SPEEDTBL or SV_VOLUMETBL
EDX_SYSTEM	• Windows NT system error - check **errno**

■ See Also

- **dx_addspddig( )**
- **dx_addvoldig( )**
- **dx_adjsv( )**
- **dx_clrsvcond( )**
- **dx_getcursv( )**
- **dx_setsvcond( )**
- **dx_setsvmt( )**
- "Speed and Volume Modification Tables" (*Voice Features Guide for Windows NT*)
- *DX_SVMT* (*Chapter 4. Voice Data Structures and Device Parameters*)

Name:	int dx_initcallp(chdev)	
Inputs:	int chdev	• valid Dialogic channel device handle
	0	• success
	-1	• failure
Returns:	srllib.h	
	dxxxlib.h	
Category:	PerfectCall Call Analysis	

■ Description

The **dx_initcallp( )** function initializes and activates PerfectCall Call Analysis on the channel identified by **chdev**. In addition, this function adds all tones used in Call Analysis to the channel's Global Tone Detection (GTD) templates.

To use PerfectCall Call Analysis, **dx_initcallp( )** must be called prior to using **dx_dial( )** on the specified channel. If **dx_dial( )** is called before initializing the channel with **dx_initcallp( )**, then Call Analysis will operate in Basic mode only for that channel.

PerfectCall Call Analysis allows the application to detect three different types of dial tone, two busy signals, ringback, and two fax or modem tones on the channel. It is also capable of distinguishing between a live voice and an answering machine when a call is connected. Parameters for these capabilities are downloaded to the channel when **dx_initcallp( )** is called.

The Voice Driver comes equipped with useful default definitions for each of the signals mentioned above. The application can change these definitions through the **dx_chgdur( )**, **dx_chgfreq( )**, and **dx_chgrepcnt( )** functions. The **dx_initcallp( )** function takes whatever definitions are currently in force and uses these definitions to initialize the specified channel.

Once a channel is initialized with the current tone definitions, these definitions cannot be changed for that channel without deleting all tones (**dx_deltones( )**) and re-initializing with another call to **dx_initcallp( )**. **dx_deltones** also disables PerfectCall Call Analysis. Note, however, that **dx_deltones( )** will erase all user-defined tones from the channel (including any Global Tone Detection information), and not just the PerfectCall Call Analysis tones.

Parameter	Description
chdev:	specifies the channel device handle.

■ Example

```
#include <stdio.h>

#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   DX_CAP  cap_s;
   int      ddd, car;
   char   *chnam, *dialstrg;

   chnam    = "dxxxB1C1";
   dialstrg = "L1234";

   /*
    *  Open channel
    */
   if ((ddd = dx_open( chnam, NULL )) == -1 ) {
      /* handle error */
   }

/*
    *  Delete any previous tones
    */
   if ( dx_deltones(ddd) < 0 ) {
      /* handle error */
   }

/*
    *  Change Enhanced call progress default local dial tone
    */
   if (dx_chgfreq( TID_DIAL_LCL, 425, 150, 0, 0 ) < 0) {
      /* handle error */
   }

   /*
    *  Change Enhanced call progress default busy cadence
    */
   if (dx_chgdur( TID_BUSY1, 550, 400, 550, 400 ) < 0) {
      /* handle error */
   }

   if (dx_chgrepcnt( TID_BUSY1, 4 ) < 0) {
      /* handle error */
   }

   /*
    * Now enable Enhanced call progress with above changed settings.
    */
```

```
   if (dx_initcallp( ddd )) {
      /* handle error */
   }

   /*
    *  Set off Hook
    */
   if ((dx_sethook( ddd, DX_OFFHOOK, EV_SYNC )) == -1) {
      /* handle error */
   }

   /*
    *  Dial
    */
   if ((car = dx_dial( ddd, dialstrg,(DX_CAP *)&cap_s, DX_CALLP|EV_SYNC))==-1) {
      /* handle error */
   }

   switch( car ) {
   case CR_NODIALTONE:
      printf(" Unable to get dial tone\n");
      break;

   case CR_BUSY:
      printf(" %s engaged\n", dialstrg );
      break;

   case CR_CNCT:
      printf(" Successful connection to %s\n", dialstrg );
      break;

   default:
      break;
   }

   /*
    *  Set on Hook
    */
   if ((dx_sethook( ddd, DX_ONHOOK, EV_SYNC )) == -1) {
      /* handle error */
   }

   dx_close( ddd );
}
```

■ Cautions

The channel must be idle.

■ See Also

- **dx_chgdur( )**
- **dx_chgfreq( )**
- **dx_chgrepcnt( )**

- **dx_deltones( )**

Name: int dx_open(namep,oflags)
Inputs: char *namep • pointer to device name to open
int oflags • Reserved for future use
Returns: >0 to indicate valid Dialogic device handle if successful
-1 if failure
Includes: srllib.h
dxxxlib.h
Category: Device Management

■ Description

The **dx_open( )** function opens a Voice device and returns a unique Dialogic device handle to identify the device. All subsequent references to the opened device must be made using the handle until the device is closed. A device can be opened more than once by any number of processes.

NOTE: The device handle returned by this function is **Dialogic defined**. It is not a standard Windows NT file descriptor. Any attempts to use Windows NT operating system commands such as **read( )**, **write( )**, or **ioctl( )** will produce unexpected results.

In applications that spawn child processes off a parent process, the device handle is not inheritable by the child process. Make sure devices are opened in the child process.

The function parameters are defined as follows:

Parameter	Description
namep:	points to an ASCIIZ string that contains the name of the valid Dialogic device. These valid devices can be either boards or channels.
oflags:	is reserved for future use. This parameter should be set to NULL.

■ Cautions

Do not use the Windows NT **open( )** function to open a Voice device. Unpredictable results will occur.

Example

```
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
main()
{
   int chdev;      /* channel descriptor */
   .
   .
   .
   /* Open Channel */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
     /* process error */
   }
   .
   .
}
```

Errors

If this function returns -1 to indicate failure, check errno for one of the following reasons:

EINVAL	• Invalid Argument
EBADF	• Invalid file descriptor
EINTR	• A signal was caught
EIO	• Error during a Windows NT STREAMS open

See Also

- **dx_close( )**

Name:	int dx_play(chdev,iottp,tptp,mode)	
Inputs:	int chdev	• valid Dialogic channel device handle
	DX_IOTT *iottp	• pointer to I/O Transfer Table Structure
	DV_TPT *tptp	• pointer to Termination Parameter Table Structure
	unsigned short mode	• asynchronous/synchronous playing mode bit mask for this play session
Returns:	0 if success -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	I/O	
Mode:	synchronous/asynchronous	

■ Description

The **dx_play( )** function plays recorded voice data or transfers Analog Display Services Interface (ADSI) data on a specified channel. The voice data may come from any combination of data files, memory, or custom devices.

The order of play and the location of the voice data is specified in the *DX_IOTT* structure pointed to by **iottp**. The DX_IOTT structure is described in *Chapter 4. Voice Data Structures and Device Parameters*.

NOTE: For a single file synchronous play, **dx_playf( )** is more convenient because you do not have to set up a DX_IOTT structure. See the **dx_playf( )** function description for more information.

■ Asynchronous Operation

To run this function asynchronously set the **mode** field to EV_ASYNC. When running asynchronously, this function will return 0 to indicate it has initiated successfully, and will generate a termination event (see below) to indicate completion.

Termination conditions for play are set using the DV_TPT structure. Play continues until all data specified in *DX_IOTT* has been played, or until one of the conditions specified in DV_TPT is satisfied.

When **dx_play( )** terminates, the current channel's status information, including the reason for termination, can be accessed using Extended Attribute functions.

Termination of asynchronous play is indicated by a TDX_PLAY event.

After **dx_play( )** terminates, use the **ATDX_TERMMSK( )** function to determine the reason for termination.

Use the SRL Event Management functions to handle the termination event. See *Appendix* A for more information about the Event Management functions.

NOTE: The DX_IOTT structure must remain in scope for the duration of the function if running asynchronously.

■ Synchronous Operation

By default, this function runs synchronously, and will return a 0 to indicate that it has completed successfully.

Termination conditions for play are set using the DV_TPT structure. Play continues until all data specified in *DX_IOTT* has been played, or until one of the conditions specified in DV_TPT is satisfied.

Termination of synchronous play is indicated by a return value of 0. After **dx_play( )** terminates, use the **ATDX_TERMMSK( )** function to determine the reason for termination.

■ Analog Display Services Interface (ADSI) Protocol

The Analog Display Services Interface (ADSI) protocol is used to transmit data to a display-based telephone that is connected to an analog loop start line. An ADSI alert tone is used to verify that Dialogic hardware is connected to an ADSI telephone and to alert the telephone that ADSI data will be transferred.

NOTE: Check with your telephone manufacturer to verify that your telephone is a true ADSI-compliant device.

Each time a new call is initiated on a channel, send the alert tone to alert the telephone that ADSI data will be transferred.

The ADSI alert tone can be sent and acknowledged, and ADSI data can be transferred using the **dx_setparm( )** and **dx_play( )** or **dx_playf( )** functions. This is accomplished by setting the voice channel parameter DXCH_DTINITSET to DM_A in the **dx_setparm( )** function and executing the **dx_play( )** or **dx_playf( )** function with the PM_ADSIALERT define ORed in the **mode** parameter.

If the acknowledgment digit is not received from the telephone within 500 ms following the end of the alert tone, the function will return a 0 but the termination mask returned by **ATDX_TERMMSK( )** will be TM_MAXTIME to indicate an ADSI protocol error.

NOTE: The function will return a -1 if a failure is due to a general play error.

If the handshaking and transmission are successful, the function terminates normally with a TM_EOD (End of data reached on playback) termination mask returned by **ATDX_TERMMSK( )** to indicate that the operation is complete.

To transfer ADSI data without an alert tone, use the **dx_clrdigbuf( )** or **dx_getdig( )** function to ensure that there are no pending digits. Transfer ADSI data using the **dx_play( )** or **dx_playf( )** function with the PM_ADSI define ORed in the **mode** parameter.

If the transmission is successful, the function terminates normally with a TM_EOD (End of data reached on playback) termination mask returned by **ATDX_TERMMSK( )** to indicate that the operation is complete.

The application is responsible for determining whether the message count acknowledgement matches the number of messages that were transmitted and for retransmitting any messages. Use the **dx_getdig( )** function with DV_TPT **tp_termno** set to DX_DIGTYPE to receive the DTMF string 'adx' where 'x' is the message count acknowledgement digit (1 - 5).

NOTE: The ADSI data must conform to interface requirements described in Bellcore Technical Reference TR-NWT-000030, *Voiceband Data Transmission Interface Generic Requirements*.

For information about message requirements (how the data should be displayed on the Customer Premise Equipment), see Bellcore Technical Reference TR-NWT-001273, *Generic Requirements for an SPCS to*

Customer Premises Equipment Data Interface for Analog Display Services.

Each technical reference can be obtained from Bellcore by calling 1-800-521-CORE.

Example code for defining and playing an alert tone, receiving acknowledgement of the alert tone, and transferring ADSI data is shown in Example 3.

The **dx_play( )** function parameters are defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.
iottp:	points to the I/O Transfer Table Structure, *DX_IOTT*, which sets the order in which and media from which the voice data will be played. See*5.1. Always Check Return Code in Voice Programming* for information about the DX_IOTT structure.
tptp:	points to the Termination Parameter Table Structure, DV_TPT, which specifies termination conditions for playing. Valid DV_TPT terminating conditions for **dx_play( )** are listed below:

DX_DIGTYPE	• User-defined digit occurred
DX_MAXDTMF	• Maximum number of digits received
DX_MAXSIL	• Maximum silence
DX_MAXNOSIL	• Maximum non-silence
DX_LCOFF	• Loop current off
DX_IDDTIME	• Inter-digit delay
DX_MAXTIME	• Function time
DX_DIGMASK	• Digit mask termination
DX_PMOFF	• Pattern match silence off
DX_PMON	• Pattern match silence on
DX_TONE	• Tone-off or Tone-on detection

See *Appendix* A, which describes the Standard Runtime

Parameter	Description
	Library, for information about this structure. **NOTE:** In addition to DV_TPT terminations, the function can fail due to maximum byte count, **dx_stopch( )**, or end of file. See **ATDX_TERMMSK( )** for a full list of termination reasons.
mode:	defines the play mode and asynchronous/synchronous mode. One or more of the play mode parameters listed below may be selected in the bit mask for play mode combinations (see Table 4). Choose one only: EV_ASYNC: Run **dx_play( )** asynchronously. EV_SYNC: Run **dx_play( )** synchronously (default). Choose one or more: MD_ADPCM: Play using Adaptive Differential Pulse Code Modulation encoding algorithm (4 bits per sample). Playing with ADPCM is the default setting. MD_PCM: Play using Pulse Code Modulation encoding algorithm (8 bits per sample). PM_ALAW Play using A-Law. PM_TONE: Transmit a tone before initiating play. If this mode is not selected, no tone will be transmitted. No tone transmitted is the default setting. PM_SR6: Play using 6KHz sampling rate (6,000 samples per second). PM_SR8: Play using 8KHz sampling rate (8,000 samples per second).

Parameter	Description	
	PM_ADSIALERT:	Play using the ADSI protocol with an alert tone preceding play. If ADSI protocol mode is selected, it is not necessary to select any other play mode parameters. PM_ADSIALERT should be ORed with the EV_SYNC or EV_ASYNC parameter in the **mode** parameter.
	PM_ADSI:	Play using the ADSI protocol without an alert tone preceding play. If ADSI protocol mode is selected, it is not necessary to select any other play mode parameters. If ADSI data will be transferred, PM_ADSI should be ORed with the EV_SYNC or EV_ASYNC parameter in the **mode** parameter.

NOTES:

1. The rate specified in the last play function will apply to the next play function, unless the rate was changed in the parameter DXCH_PLAYDRATE using **dx_setparm( )**.
2. Specifying PM_SR6 or PM_SR8 using **dx_play( )** changes the setting of the parameter DXCH_PLAYDRATE. DXCH_PLAYDRATE can also be set and queried using **dx_setparm( )** and **dx_getparm( )**. The default setting for DXCH_PLAYDRATE is 6KHz.
3. Make sure data is played using the same encoding algorithm and sampling rate used when the data was recorded.
4. MD_PCM *can* be used on D/12x or D/81A board.
5. The D/21E, D/41E, D/41ESC, D/160SC-LS, D/240SC, D/240SC-T1, D/300SC-E1 and D/320SC boards enable the user to select either A-Law or mu-Law encoding of data. The default on the board is set to mu-Law and returns to mu-Law after each play. The A-Law parameters must be passed each time the play function is called. Enable A-Law playback by OR'ing the new play mode,

Parameter	Description
	PM_ALAW.

Table 4 shows play mode selections when transmitting or not transmitting a tone before initiating play. The first column of the table lists the two play features (tone or no tone), and the first row lists each type of encoding algorithm (ADPCM or PCM) and data-storage rate for each algorithm/sampling rate combination in parenthesis (24 Kbps, 32 Kbps, 48 Kbps, or 64 Kbps).

Select the desired play feature in the first column of the table and look across that row until the column containing the desired encoding algorithm and data-storage rate is reached. The play modes that must be entered in the mode bit mask are provided where the feature row and encoding algorithm/data-storage rate column intersect. Parameters listed in { } are default settings and do not have to be specified.

NOTE: If PM_ADSI play mode is selected (not shown in Table 4), the ADSI protocol will be used to transfer ADSI data and it is not necessary to select any other play mode parameters. PM_ADSI should be ORed with the EV_SYNC or EV_ASYNC parameter in the mode parameter.

Table 4. Play Mode Selections

Feature(s)	ADPCM (24 Kbps)	ADPCM (32 Kbps)	PCM (48 Kbps)	PCM (64 Kbps)
• Tone	PM_TONE PM_SR6 {MD_ADPCM }	PM_TONE PM_SR8 {MD_ADPCM }	PM_TONE PM_ALAW* PM_SR6 MD_PCM	PM_TONE PM_ALAW* PM_SR8 MD_PCM
• No Tone	PM_SR6 {MD_ADPCM }	PM_SR8 {MD_ADPCM }	PM_SR6 MD_PCM	PM_SR8 MD_PCM

{ } = Default modes.
* = Select if file was encoded using A-Law (supported by D/41ESC, D/160SC-LS, D/240SC, D/240SC-T1, D/300SC-E1 and D/320SC boards only).

NOTE: **dx_play( )** will run synchronously if you do not specify EV_ASYNC, or if

you specify EV_SYNC (default).

■ Cautions

Whenever **dx_play( )** is called, its speed and volume is based on the most recent adjustment made using **dx_adjsv( )** or **dx_setsvcond( )**.

■ Example 1: Using dx_play( ) in synchronous mode.

```
/* Play a voice file. Terminate on receiving 4 digits or at end of file*/
#include <fcntl.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
main()
{
   int chdev;
   DX_IOTT iott;
   DV_TPT  tpt;
   DV_DIGIT dig;
   .
   .
   /* Open the device using dx_open( ). Get channel device descriptor in
    * chdev.
    */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
     /* process error */
   }
   /* set up DX_IOTT */
   iott.io_type = IO_DEV|IO_EOT;
   iott.io_bufp = 0;
   iott.io_offset = 0;
   iott.io_length = -1;  /* play till end of file */
   if((iott.io_fhandle = dx_fileopen("prompt.vox", O_RDONLY|O_BINARY))
         == -1)  {
     /* process error */
   }
   /* set up DV_TPT */
   dx_clrtpt(&tpt,1);
   tpt.tp_type   = IO_EOT;          /* only entry in the table */
   tpt.tp_termno = DX_MAXDTMF;      /* Maximum digits */
   tpt.tp_length = 4;               /* terminate on four digits */
   tpt.tp_flags  = TF_MAXDTMF;      /* Use the default flags */
   /* clear previously entered digits */
   if (dx_clrdigbuf(chdev) == -1)  {
     /* process error */
   }
/* Now play the file */
   if (dx_play(chdev,&iott,&tpt,EV_SYNC) == -1) {
     /* process error */
   }
   /* get digit using dx_getdig( ) and continue processing. */
   .
   .
}
```

■ Example 2: Using dx_play( ) in asynchronous mode.

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
#define MAXCHAN 24
int play_handler();
DX_IOTT prompt[MAXCHAN];
DV_TPT tpt;
DV_DIGIT dig;
main()
{
   int chdev[MAXCHAN], index, index1;
   char *chname;
   int i, srlmode, voxfd;
   /* Set SRL to run in polled mode. */
   srlmode = SR_POLLMODE;
   if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1) {
      /* process error */
   }
   /* initialize all the DX_IOTT structures for each individual prompt */
   .
   .
   /* Open the vox file to play; the file descriptor will be used
    * by all channels.
    */
   if ((voxfd = dx_fileopen("prompt.vox", O_RDONLY|O_BINARY)) == -1) {
      /* process error */
   }
/* For each channel, open the device using dx_open(), set up a DX_IOTT
    * structure for each channel, and issue dx_play() in asynchronous mode. */
   for (i=0; i<MAXCHAN; i++) {
      /* Set chname to the channel name, e.g., dxxxB1C1, dxxxB1C2,... */
      /* Open the device using dx_open( ).  chdev[i] has channel device
      * descriptor.
      */
      if ((chdev[i] = dx_open(chname,NULL)) == -1)   {
         /* process error */
      }
      /* Use sr_enbhdlr() to set up handler function to handle play
      * completion events on this channel.
      */
      if (sr_enbhdlr(chdev[i], TDX_PLAY, play_handler) == -1) {
         /* process error */
      }
      /*
       * Set the DV_TPT structures up for MAXDTMF. Play until one digit is
       * pressed or the file is played
       */
      dx_clrtpt(&tpt,1);
      tpt.tp_type   = IO_EOT;            /* only entry in the table */
      tpt.tp_termno = DX_MAXDTMF;        /* Maximum digits */
      tpt.tp_length = 1;                 /* terminate on the first digit */
      tpt.tp_flags  = TF_MAXDTMF;        /* Use the default flags */
      prompt[i].io_type = IO_DEV|IO_EOT; /* play from file */
      prompt[i].io_bufp = 0;
      prompt[i].offset = 0;
      prompt[i].io_length = -1;          /* play till end of file */
      prompt[i].io_nextp = NULL;
      prompt[i].io_fhandle = voxfd;
      /* play the data */
      if (dx_play(chdev[i],&prompt[i],&tpt,EV_ASYNC) == -1) {
         /* process error */
```

```
      }
   }
/* Use sr_waitevt to wait for the completion of dx_play().
    * On receiving the completion event, TDX_PLAY, control is transferred
    * to the handler function previously established using sr_enbhdlr().
    */
   .
   .
}
int play_handler()
{
   long term;
   /* Use ATDX_TERMMSK() to get the reason for termination. */
   term = ATDX_TERMMSK(sr_getevtdev());
   if (term & TM_MAXDTMF) {
      printf("play terminated on receiving DTMF digit(s)\n");
   } else if (term & TM_EOD) {
      printf("play terminated on reaching end of data\n");
   } else {
      printf("Unknown termination reason: %x\n", term);
   }
   /* Kick off next function in the state machine model. */
   .
   .
   return 0;
}
```

■ Example 3: Defining and playing an alert tone, receiving acknowledgement of the alert tone, and using dx_play( ) to transfer ADSI data.

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
int parm;
DV_TPT tpt[2];
DV_DIGIT digit;
TN_GEN tngen;
DX_IOTT iott;
main(argc,argv)
   int argc;
   char* argv[];
{
   int chfd;
   char channame[12];
   parm = SR_POLLMODE;
   sr_setparm[SRL_DEVICE, SR_MODEID, &parm);
   /*
    * Open the channel using the command line arguments as input
    */
   sprintf(channame, "%sC%s", argv[1],argv[2]);
if (( chfd = dx_open(channame, NULL)) == -1) {
    printf("Board open failed on device %s\n",channame);
    exit(1);
}
    printf("Devices open and waiting .....\n");
    /*
  * Take the phone off-hook to talk to the ADSI phone
  * This assumes we are connected through a Skutch Box.
  */
 if (dx_sethook( chfd, DX_OFFHOOK, EV_SYNC) == -1)  {
    printf("sethook failed\n");
```

```
    while (1) {
       sleep(5);
       dx_clrdigbuf( chfd );
    printf("Digit buffer cleared ..\n);
   /*
    * Generate the alert tone
    */
iott.io_type =IO_DEV|IO_EOT;
    iott.io_fhandle = dx_fileopen("message.asc",O_RDONLY);
iott.io_length = -1;
    parm = DM_D
    if (dx_setparm (chfd, DXCH_DTINITSET, (void *)parm) ==-1){
        printf ("dx_setparm on DTINITSET failed\n");
        exit(1);
    {
    if (dx_play(chfd,&iott,(DV_TPT *)NULL, PM_ADSIALERT|EV_SYNC) ==-1) {
       printf("dx_play on the ADSI file failed\n");
       exit(1);
    }
 }

 dx_close(chfd);
 exit(0);
}
```

■ Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	• Invalid Parameter
EDX_BADIOTT	• Invalid *DX_IOTT* entry
EDX_BADTPT	• Invalid DX_TPT entry
EDX_BUSY	• Busy executing I/O function
EDX_SYSTEM	• Windows NT system error - check **errno**

■ See Also

Related Functions:

- **dx_playf( )**
- **dx_rec( )**
- **dx_recf( )**
- **dx_setparm( ), dx_getparm( )**

Setting Speed and Volume:

- **dx_adjsv( )**
- **dx_setsvcond( )**

Setting Order and Location for Voice Data:

- *DX_IOTT* (*Chapter 4. Voice Data Structures and Device Parameters*)

Retrieving and Handling Play Termination Events:

- Event Management functions (*Standard Runtime Library Programmer's Guide for Windows NT* and *Appendix* A of this guide)
- **ATDX_TERMMSK( )**
- DV_TPT (*Appendix* A)

Name:	int dx_playf(chdev,fnamep,tptp,mode)	
Inputs:	int chdev	• valid Dialogic channel device handle
	char *fnamep	• pointer to name of file to play
	DV_TPT *tptp	• pointer to Termination Parameter Table Structure
	unsigned short mode	• playing mode bit mask for this play session
Returns:	0 if success -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	Convenience	

■ Description

dx_playf( ) is a convenience function that synchronously plays voice data or transfers ADSI data (using the ADSI protocol) from a single file.

dx_playf( ) operates the same as synchronous **dx_play( )** if the *DX_IOTT* structure specified a single file entry. **dx_playf( )** is provided as a convenient way to play back data or transfer ADSI data from a single file without having to specify a DX_IOTT structure for only one file. The **dx_playf( )** function opens and closes the file specified by **fnamep** while the **dx_play( )** function uses a DX_IOTT structure that requires the application to open and close the file.

Parameter	Description
fnamep:	points to the file from which voice data will be played.

For information about other function arguments and transferring ADSI data, see **dx_play( )**.

■ Source Code

```
/*****************************************************************************
 *        NAME: int dx_playf(devd,filep,tptp,mode)
 * DESCRIPTION: This function opens and plays a
 *              named file.
 *      INPUTS: devd - channel descriptor
 *              tptp - pointer to the termination control block
 *              filep - pointer to file name
```

```
 *     OUTPUTS: Data is played.
 *     RETURNS: 0 - success -1 - failure
 *       CALLS: open() dx_play() close()
 *    CAUTIONS: none.
 ****************************************************************************/
int dx_playf(devd,filep,tptp,mode)
   int     devd;
   char    *filep;
   DV_TPT  *tptp;
   USHORT  mode;
{
   DX_IOTT iott;
   int     rval;


   /*
    * If Async then return Error
    * Reason: IOTT's must be in scope for the duration of the play
    */
   if ( mode & EV_ASYNC ) {
      return( -1 );
   }


   /* Open the File */
   if ((iott.io_fhandle = dx_fileopen(filep,O_RDONLY)|O-BINARY) == -1) {
      return -1;
   }


   /* Use dx_play() to do the Play */
   iott.io_type = IO_EOT | IO_DEV;
   iott.io_offset = (unsigned long)0;
   iott.io_length = -1;


   rval = dx_play(devd,&iott,tptp,mode);


   if (dx_fileclose(iott.io_fhandle) == -1) {
      return -1;
   }


   return rval;
}
```

■ Example

```
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>


main()
{
   int chdev;
   DV_TPT tpt[2];


   /* Open the channel using dx_open( ). Get channel device descriptor in
    * chdev.
    */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
     /* process error */
   }
```

```
   /*
    * Set up the DV_TPT structures for MAXDTMF. Play until one digit is
    *  pressed or the file has completed play
    */
   dx_clrtpt(tpt,1);
   tpt[0].tp_type   = IO_EOT;      /* only entry in the table */
   tpt[0].tp_termno = DX_MAXDTMF;  /* Maximum digits */
   tpt[0].tp_length = 1;           /* terminate on the first digit */
   tpt[0].tp_flags  = TF_MAXDTMF;  /* Use the default flags */

   if (dx_playf(chdev,"weather.vox",tpt,EV_SYNC) == -1) {
     /* process error */
   }
   .
   .
}
```

■ Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	• Invalid Parameter
EDX_BADIOTT	• Invalid *DX_IOTT* entry
EDX_BADTPT	• Invalid DX_TPT entry
EDX_BUSY	• Busy executing I/O function
EDX_SYSTEM	• Windows NT system error - check **errno**

■ See Also

Related Functions:

- **dx_rec( )**
- **dx_recf( )**
- **dx_setparm( ), dx_getparm( )**

Setting Speed and Volume:

- **dx_adjsv( )**
- **dx_setsvcond( )**

Setting and Handling Play Termination:

- **ATDX_TERMMSK( )**
- DV_TPT (*Appendix* A)

Name:	short dx_playiottdata(chdev, iottp, tptp, xpbp, mode)	
Inputs:	int chdev	• valid Dialogic channel device handle
	DX_IOTT *iottp	• pointer to I/O transfer table
	DV_TPT *tptp	• pointer to termination parameter block
	DX_XPB *xpbp	• pointer to I/O transfer parameter block
	unsigned short mode	• play mode
Returns:	0 if success -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	I/O function	
Mode:	synchronous or asynchronous	

■ Description

The **dx_playiottdata( )** function plays back recorded voice data from multiple sources on a channel. The file format for the files to be played is specified in the **wFileFormat** field of the *DX_XPB*. Other fields in the DX_XPB describe the data format. For files that include data format information (i.e. WAVE files), these other fields are ignored.

Parameter	Description	
chdev	channel device descriptor.	
iottp	the voice data may come from any combination of data files, memory, or custom devices. The order of playback and the location of the voice data is specified in an array of *DX_IOTT* structures pointed to by **iottp**	
tptp	pointer to Termination parameter table	
xpbp	pointer to I/O transfer parameter block	
mode	specifies the record mode:	
	PM_TONE	play 200 ms audible tone
	EV_SYNCH	synchronous mode
	EV_ASYNCH	asynchronous mode

■ Cautions

1. All files specified in the *DX_IOTT* table must be of the same file format type and match the file format indicated in *DX_XPB*.
2. All files specified in the DX_IOTT table must contain data of the type described in DX_XPB.
3. When playing or recording VOX files, the data format is specified in DX_XPB rather than through the mode argument of this function.
4. When set to play WAVE files, all other fields in the DX_XPB are ignored.
5. When set to play WAVE files, this function will fail if an unsupported data format is attempted to be played. The supported data forms are:

- 6, 8, and 11KHz linear 8-bit PCM (WAVE_FORMAT_PCM)
- 6, 8, and 11KHz mu-law 8-bit PCM (WAVE_FORMAT_MULAW)
- 6, 8, and 11KHz a-law 8-bit PCM (WAVE_FORMAT_ALAW)
- 6 and 8KHz 4-bit Oki ADPCM (WAVE_FORMAT_DIALOGIC_OKI_ADPCM)

■ Example

```
#include "srllib.h"
#include “dxxxlib.h”

int chdev;              /* channel descriptor */
int fd;         /* file descriptor for file to be played */
DX_IOTT iott;   /* I/O transfer table */
DV_TPT tpt;     /* termination parameter table */
DX_XPB xpb;     /* I/O transfer parameter block */
.
.
.
/* Open channel */
if ((chdev = dx_open("dxxxB1C1",0)) == -1) {
      printf("Cannot open channel\n");
      printf("errno = %d\n",errno);
      exit(1);
}
/* Set to terminate play on 1 digit */
tpt.tp_type   = IO_EOT;
tpt.tp_termno = DX_MAXDTMF;
tpt.tp_length = 1;
tpt.tp_flags  = TF_MAXDTMF;
/* Open VOX file to play */
if ((fd = dx_fileopen("HELLO.VOX",O_RDONLY|O_BINARY)) == -1) {
      printf("File open error\n");
      exit(2);
}
/* Set up DX_IOTT */
```

```
iott.io_fhandle = fd;
iott.io_bufp    = 0;
iott.io_offset  = 0;
iott.io_length  = -1;
iott.io_typ = IO_DEV | IO_EOT;
/*
 * Specify VOX file format for ADPCM at 8KHz
 */
xpb.wFileFormat = FILE_FORMAT_VOX;
xpb.wDataFormat = DATA_FORMAT_DIALOGIC_ADPCM;
xpb.nSamplesPerSec = DRT_8KHZ;
xpb.nBitsPerSample = 4;

/* Wait forever for phone to ring and go offhook */
if (dx_wtring(chdev,1,DX_OFFHOOK,-1) == -1) {
      printf("Error waiting for ring - %s\n", ATDV_LASTERR(chdev));
      exit(3);
}
/* Start playback */
if (dx_playiottdata(chdev,&iott,&tpt,&xpb,EV_SYNC)==-1) {
      printf("Error playing file - %s\n", ATDV_ERRMSGP(chdev));
      exit(4);
}
```

■ Errors

In asynchronous mode, function returns immediately and a TDX_PLAY event is queued upon completion. Check **ATDX_LASTTERM( )** for the termination reason. If a failure occurs, then a TDX_ERROR event will be queued. Use **ATDV_LASTERR( )** to determine the reason for error.

In synchronous mode, if this function returns -1 to indicate failure, one of the following reasons will be contained by **ATDV_LASTERR( )** :

Equate	Returned When
EDX_BUSY	Channel is busy
EDX_XPBPARM	Invalid *DX_XPB* setting
EDX_BADIOTT	Invalid *DX_IOTT* setting
EDX_SYSTEM	System I/O errors
EDX_BADWAVFILE	Invalid WAV file
EDX_SH_BADCMD	Unsupported command or WAV file format

■ See Also

- **dx_playwav( )**
- **dx_playvox( )**

Name:	int dx_playtone(chdev,tngenp,tptp,mode)	
Inputs:	int chdev	• valid Dialogic channel device handle
	TN_GEN *tngenp	• pointer to the TN_GEN structure
	DV_TPT*tptp	• pointer to the DV_TPT structure
	int mode	• asynchronous/synchronous
Returns:	0 if success -1 if failurewhat error	
Includes:	srllib.h dxxxlib.h	
Category:	Global Tone Generation	
Mode:	asynchronous/synchronous	

■ Description

The **dx_playtone( )** function plays tone defined by TN_GEN template, which defines the frequency amplitude and duration of a single or dual frequency tone to be played.

NOTE: The dx_playtone( ) function is necessary for supporting the R2MF protocol in an application. See r2_playbsig( ) for information.

■ Asynchronous Operation

To run this function asynchronously set the **mode** field to EV_ASYNC. When running asynchronously, this function will return 0 to indicate it has initiated successfully, and will generate a termination event (see below) to indicate completion.

Set termination conditions using the DV_TPT structure. This structure is pointed to by the **tptp** parameter described below.

Termination of this function is indicated by a TDX_PLAYTONE event.

Use the SRL Event Management functions to handle the termination event. See *Appendix* A for more information about the Event Management functions.

After **dx_playtone( )** terminates, use the **ATDX_TERMMSK( )** function to determine the reason for termination.

■ Synchronous Operation

By default, this function runs synchronously, and will return a 0 to indicate that it has completed successfully.

Set termination conditions using the DV_TPT structure. This structure is pointed to by the **tptp** parameter described below.

Termination of synchronous play is indicated by a return value of 0.

After **dx_playtone( )** terminates, use the **ATDX_TERMMSK( )** function to determine the reason for termination.

Parameter	Description	
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.	
tngenp:	points to the *TN_GEN* template structure, which defines the frequency, amplitude and duration of a single or dual frequency tone. See *Chapter* 4. Voice Data Structures and Device Parameters for a full description of this template. **dx_bldtngen( )** can be used to set up the structure.	
tptp:	points to the DV_TPT data structure, which specifies one of the following terminating conditions for this function:	
	DX_DIGTYPE	• Digit termination for user-defined tone
	DX_MAXDTMF	• Maximum number of digits received
	DX_MAXSIL	• Maximum silence
	DX_MAXNOSIL	• Maximum non-silence
	DX_LCOFF	• Loop current off
	DX_IDDTIME	• Inter-digit delay
	DX_MAXTIME	• Function time
	DX_DIGMASK	• Digit mask termination
	DX_PMOFF	• Pattern match silence off
	DX_PMON	• Pattern match silence on
	DX_TONE	• Tone-off or Tone-on detection

Parameter	Description
	See *Appendix* A, which describes the Standard Runtime Library, for information about this structure.
mode:	specifies whether to run this function asynchronously or synchronously. Set to one of the following:
	EV_ASYNC: Run **dx_playtone( )** asynchronously.
	EV_SYNC: Run **dx_playtone( )** synchronously (default).

■ Cautions

1. The channel must be idle when calling this function.
2. If the tone generation template contains an invalid **tg_dflag**, or the specified amplitude or frequency is outside the valid range, **dx_playtone( )** will generate a TDX_ERROR event if asynchronous, or -1 if synchronous.

■ Example

```
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

#define TID_1   101

main()
{
   TN_GEN   tngen;
   DV_TPT     tpt[ 5 ];
   int        dxxxdev;

   /*
    * Open the Voice Channel Device and Enable a Handler
    */
   if ( ( dxxxdev = dx_open( "dxxxB1C1", NULL) ) == -1 ) {
      perror( "dxxxB1C1" );
      exit( 1 );
   }

   /*
```

```
 * Describe a Simple Dual Tone Frequency Tone of 950-
 * 1050 Hz and 475-525 Hz using leading edge detection.
 */
if ( dx_bldt( TID_1, 1000, 50, 500, 25, TN_LEADING ) == -1 ) {
   printf( "Unable to build a Dual Tone Template\n" );
}

/*
 * Bind the Tone to the Channel
 */
if ( dx_addtone( dxxxdev, NULL, 0 ) == -1 ) {
   printf( "Unable to Bind the Tone %d\n", TID_1 );
   printf( "Lasterror = %d  Err Msg = %s\n",
       ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
   dx_close( dxxxdev );
   exit( 1 );
}

/*
 * Enable Detection of ToneId TID_1
 */
if ( dx_enbtone( dxxxdev, TID_1, DM_TONEON | DM_TONEOFF ) == -1 ) {
   printf( "Unable to Enable Detection of Tone %d\n", TID_1 );
   printf( "Lasterror = %d  Err Msg = %s\n",
       ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
   dx_close( dxxxdev );
   exit( 1 );
}

/*
 * Build a Tone Generation Template.
 * This template has Frequency1 = 1140,
 * Frequency2 = 1020, amplitute at -10dB for
 * both frequencies and duration of 100 * 10 msecs.
 */
dx_bldtngen( &tngen, 1140, 1020, -10, -10, 100 );

/*
 * Set up the Terminating Conditions
 */
tpt[0].tp_type = IO_CONT;
tpt[0].tp_termno = DX_TONE;
tpt[0].tp_length = TID_1;
tpt[0].tp_flags = TF_TONE;
tpt[0].tp_data = DX_TONEON;

tpt[1].tp_type = IO_CONT;
tpt[1].tp_termno = DX_TONE;
tpt[1].tp_length = TID_1;
tpt[1].tp_flags = TF_TONE;
tpt[1].tp_data = DX_TONEOFF;

tpt[2].tp_type = IO_EOT;
tpt[2].tp_termno = DX_MAXTIME;
tpt[2].tp_length = 6000;
tpt[2].tp_flags = TF_MAXTIME;

if (dx_playtone( dxxxdev, &tngen, tpt, EV_SYNC ) == -1 ){
   printf( "Unable to Play the Tone\n" );
   printf( "Lasterror = %d  Err Msg = %s\n",
       ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
```

```
      dx_close( dxxxdev );
      exit( 1 );
   }

   /*
    * Continue Processing
    *   .
    *   .
    *   .
    */

   /*
    * Close the opened Voice Channel Device
    */
   if ( dx_close( dxxxdev ) != 0 ) {
      perror( "close" );
   }

   /* Terminate the Program */
   exit( 0 );
}
```

■ Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	• Invalid parameter
EDX_BADPROD	• Function not supported on this board
EDX_BADTPT	• Invalid DV_TPT entry
EDX_BUSY	• Busy executing I/O function
EDX_AMPLGEN	• Invalid amplitude value in *TN_GEN* structure
EDX_FREQGEN	• Invalid frequency component in TN_GEN structure
EDX_FLAGGEN	• Invalid tn_dflag field in TN_GEN structure
EDX_SYSTEM	• Windows NT system error - check **errno**

■ See Also

Related to Tone Generation:

- **dx_bldtngen( )**
- *TN_GEN* (*Chapter 4. Voice Data Structures and Device Parameters*)
- "Global Tone Generation" (*Voice Features Guide for Windows NT*)

R2MF functions:

- **r2_creatfsig( )**
- **r2_playbsig( )**

Handling and Retrieving **dx_playtone( )** Termination Events:

- Event Management functions (*Standard Runtime Library Programmer's Guide for Windows NT* and *Appendix* A)
- DV_TPT (*Appendix* A)
- **ATDX_TERMMSK( )**

Name:	SHORT dx_playvox(chdev, filenamep, tptp, xpbp, mode)	
Inputs:	int chdev	• valid Dialogic channel device handle
	char *filenamep	• pointer to name of file to play
	DV_TPT *tptp	• pointer to termination parameter block
	DX_XPB *xpbp	• pointer to I/O transfer parameter block
	unsigned short mode	• play mode
Returns:	0 if successful -1 if failure	
Includes:	dxxxlib.h	
Category:	Convenience function	
Mode:	synchronous	

■ Description

The **dx_playvox( )** convenience function plays voice data stored in a single VOX file. If **xpbp** is set to NULL, it will interpret the data as 6KHz linear ADPCM.

Parameter	Description
chdev	Channel device descriptor
tcbp	Pointer to termination parameter table
filenamep	Pointer to name of file to play
xpbp	Pointer to I/O transfer parameter block (See the *DX_XPB* data structure)
mode	specifies the play mode: PM_TONE play 200 ms audible tone EV_SYNC synchronous operation (must be specified) **NOTE:** Both PM_TONE and EV_SYNC can be specified by ORing the two values.

■ Cautions

When playing or recording VOX files, the data format is specified in *DX_XPB* rather than through the **dl_stprm( )** function.

■ Example

```
#include "srllib.h"
#include "dxxxlib.h"

int chdev;        /* channel descriptor */
DV_TPT tpt;       /* termination parameter table */.
.
.

/* Open channel */
if ((chdev = dx_open("dxxxB1C1",0)) == -1) {
      printf("Cannot open channel\n");
      printf("errno = %d\n",errno);
      exit(1);
}
/* Set to terminate play on 1 digit */
tpt.tp_type   = IO_EOT;
tpt.tp_termno = DX_MAXDTMF;
tpt.tp_length = 1;
tpt.tp_flags  = TF_MAXDTMF;
/* Wait forever for phone to ring and go offhook */
if (dx_wtring(chdev,1,DX_OFFHOOK,-1) == -1) {
      printf("Error waiting for ring - %s\n",       ATDV_LASTERR(chdev));
      exit(3);
}

/* Start 6KHz ADPCM playback */
if (dx_playvox(chdev,&tpt,"HELLO.VOX",NULL,0) == -1) {
      printf("Error playing file - %s\n",       ATDV_ERRMSGP(chdev));
      exit(4);
}
```

■ Errors

If this function returns -1 to indicate failure, one of the following reasons will be contained by **ATDV_LASTERR( )**:

Equate	Returned When
EDX_BUSY	Channel is busy
EDX_XPBPARM	Invalid *DX_XPB* setting
EDX_BADIOTT	Invalid *DX_IOTT* setting
EDX_SYSTEM	System I/O errors
EDX_BADWAVFILE	Invalid WAV file

Equate	Returned When
EDX_SH_BADCMD	Unsupported command or WAV file format

See Also

- **dx_playiottdata( )**
- **dx_playwav( )**

Name: SHORT dx_playwav(chdev, filenamep, tptp, mode)

Inputs:	int chdev	• valid Dialogic channel device handle
	char *filenamep	• pointer to name of file to play
	DV_TPT*tptp	• pointer to termination parameter block
	unsigned short mode	• play mode
Returns:	0 if successful -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	Convenience function	
Mode:	synchronous	

■ Description

The **dx_playwav( )** convenience function plays voice data stored in a single WAVE file. This function calls **dx_playiottdata( )**.

The function does not specify a *DX_XPB* stucture because the WAVE file contains the necessary format information.

Parameter	Description
chdev	Channel device descriptor
tcbp	Pointer to termination parameter table
filenamep	Pointer to name of file to play
mode	specifies the play mode: PM_TONE play 200 ms audible tone EV_SYNC synchronous operation (must be specified) **NOTE:** Both PM_TONE and EV_SYNC can be specified by ORing the two values.

■ Cautions

This function fails when an unsupported data waveform attempts to play. The supported waveforms are:

- 6, 8, and 11KHz linear 8-bit PCM (WAVE_FORMAT_PCM)
- 6, 8, and 11KHz mu-law 8-bit PCM (WAVE_FORMAT_MULAW)
- 6, 8, and 11KHz a-law 8-bit PCM (WAVE_FORMAT_ALAW)
- 6 and 8KHz 4-bit Oki ADPCM (WAVE_FORMAT_DIALOGIC_OKI_ADPCM)

■ Example

```
#include "srllib.h"
#include "dxxxlib.h"

int chdev;        /* channel descriptor */
DV_TPT tpt;        /* termination parameter table */
.
.
.

/* Open channel */
if ((chdev = dx_open("dxxxB1C1",0)) == -1) {
      printf("Cannot open channel\n");
      printf("errno = %d\n",errno);
      exit(1);
}
/* Set to terminate play on 1 digit */
tpt.tp_type   = IO_EOT;
tpt.tp_termno = DX_MAXDTMF;
tpt.tp_length = 1;
tpt.tp_flags  = TF_MAXDTMF;
/* Wait forever for phone to ring and go offhook */
if (dx_wtring(chdev,1,DX_OFFHOOK,-1) == -1) {
      printf("Error waiting for ring - %s\n",     ATDV_LASTERR(chdev));
      exit(3);
}
/* Start playback */
if (dx_playwav(chdev,&tpt,"HELLO.WAV",EV_SYNC) == -1) {
      printf("Error playing file - %s\n",        ATDV_ERRMSGP(chdev));
      exit(4);
}
```

■ Errors

If this function returns -1 to indicate failure, one of the following reasons will be contained by **ATDV_LASTERR( )**:

Equate	Returned When
EDX_BUSY	Channel is busy
EDX_XPBPARM	Invalid *DX_XPB* setting
EDX_BADIOTT	Invalid *DX_IOTT* setting
EDX_SYSTEM	System I/O errors
EDX_BADWAVFILE	Invalid WAV file
EDX_SH_BADCMD	Unsupported command or WAV file format

■ See Also

- **dx_playiottdata( )**
- **dx_playvox( )**

Name:	int dx_rec(chdev,iottp,tptp,mode)	
Inputs:	int chdev	• valid Dialogic channel device handle
	DX_IOTT *iottp	• pointer to I/O Descriptor Table
	DV_TPT *tptp	• pointer to Termination Parameter Table Structure
	unsigned short mode	• asynchronous/synchronous setting and recording mode bit mask for this record session
Returns:	0 if successful -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	I/O	
Mode:	synchronous/asynchronous	

■ Description

The **dx_rec( )** function records voice data from a single channel. The data may be recorded to a combination of data files, memory, or custom devices.

The order in which voice data is recorded is specified in the *DX_IOTT* structure. The DX_IOTT structure must remain in scope for the duration of the function if running asynchronously.

After **dx_rec( )** is called, recording continues until **dx_stopch( )** is called, the data requirements specified in the DX_IOTT are fulfilled, or until one of the conditions for termination in the DV_TPT is satisfied. When **dx_rec( )** terminates, the current channel's status information, including the reason for termination, can be accessed using Extended Attribute functions.

NOTE: For a single file synchronous record, **dx_recf( )** is more convenient because you do not have to set up a DX_IOTT structure. See the function description of **dx_recf( )** for information.

■ Asynchronous Operation

To run this function asynchronously set the **mode** field to EV_ASYNC. When running asynchronously, this function will return 0 to indicate it has initiated

successfully, and will generate a termination event (see below) to indicate completion.

Set termination conditions using the DV_TPT structure. This structure is pointed to by the **tptp** parameter described below.

Termination of asynchronous recording is indicated by a TDX_RECORD event.

Use the SRL Event Management functions to handle the termination event. See *Appendix* A for more information about the Event Management functions.

After **dx_rec( )** terminates, use the **ATDX_TERMMSK( )** function to determine the reason for termination.

■ Synchronous Operation

By default, this function runs synchronously, and will return a 0 to indicate that it has completed successfully.

Set termination conditions using the DV_TPT structure. This structure is pointed to by the **tptp** parameter described below. After **dx_rec( )** terminates, use the **ATDX_TERMMSK( )** function to determine the reason for termination.

The function parameters are defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.
iottp:	points to the I/O Transfer Table Structure, *DX_IOTT*, which specifies the order in which and media onto which the voice data will be recorded. This structure is defined in *Chapter 4. Voice Data Structures and Device Parameters* and must remain in scope for the duration of the function if using asynchronously.
tptp:	points to the Termination Parameter Table Structure, DV_TPT, which specifies termination conditions for recording. Valid termination conditions for this function are listed below:
	DX_DIGTYPE • Digit termination for user defined tone

Parameter	Description	
	DX_MAXDTMF	• Maximum number of digits received
	DX_MAXSIL	• Maximum silence
	DX_MAXNOSIL	• Maximum non-silence
	DX_LCOFF	• Loop current off
	DX_IDDTIME	• Inter-digit delay
	DX_MAXTIME	• Function time
	DX_DIGMASK	• Digit mask termination
	DX_PMOFF	• Pattern match silence off
	DX_PMON	• Pattern match silence on
	DX_TONE	• Tone-off or Tone-on detection

See *Appendix* A, which describes the Standard Runtime Library, for information about this structure.

NOTE: In addition to DV_TPT terminations, the function can fail due to maximum byte count, **dx_stopch( )**, or end of file. See **ATDX_TERMMSK( )** for a full list of termination reasons.

mode: defines the recording mode. One or more of the values listed below may be selected in the bit mask (see Table 5 for record mode combinations).

Choose one only:

EV_ASYNC: Run **dx_rec( )** asynchronously.

EV_SYNC: Run **dx_rec( )** synchronously (default).

Choose one or more:

MD_ADPCM: Record using Adaptive Differential Pulse Code Modulation encoding algorithm (4 bits per sample). Recording with ADPCM is the default setting.

Parameter	Description	
	MD_PCM:	Record using Pulse Code Modulation encoding algorithm (8 bits per sample).
	MD_GAIN:	Record with Automatic Gain Control (AGC). Recording with AGC is the default setting.
	MD_NOGAIN:	Record without AGC.
	RM_ALAW:	Record using A-Law.
	RM_TONE:	Transmit a tone before initiating record. If this mode is not selected, no tone will be transmitted (the default setting).
	RM_SR6:	Record using 6KHz sampling rate (6,000 samples per second). This is the default setting.
	RM_SR8:	Record using 8KHz sampling rate (8,000 samples per second).

NOTES:

1. The rate specified in the last record function will apply to the next record function, unless the rate was changed in the parameter DXCH_RECRDRATE using **dx_setparm( )**.
2. Specifying RM_SR6 or RM_SR8 in mode changes the setting of the parameter DXCH_RECRDRATE. DXCH_RECRDRATE can also be set and queried using **dx_setparm( )** and **dx_getparm( )**. The default setting for DXCH_RECRDRATE is 6KHz.
3. If both MD_ADPCM and MD_PCM are set, MD_PCM will take precedence. If both MD_GAIN and MD_NOGAIN are set, MD_NOGAIN will take precedence. If both RM_TONE and NULL are set, RM_TONE takes precedence. If both RM_SR6 and RM_SR8 are set, RM_SR6 will take precedence.
4. MD_PCM and MD_NOGAIN can be used on D/12x or D/81A boards.

Parameter **Description**

5. When playing pre-recorded data, make sure it is played using the same encoding algorithm and sampling rate used when the data was recorded.

6. **dx_rec( )** will run synchronously if you do not specify EV_ASYNC, or if you specify EV_SYNC (default).

7. The D/21E, D/41E, D/41ESC, D/160SC-LS, D/240SC, D/240SC-T1, D/300SC-E1 and D/320SC boards enable the user to select either A-Law or mu-Law encoding of data. The default on the board is set to mu-Law and returns to mu-Law after each record. The A-Law parameters must be passed each time the record function is called. Enable A-Law record by OR'ing the new record, RM_ALAW.

Table 5 shows recording mode selections. The first column of the table lists all possible combinations of record features, and the first row lists each type of encoding algorithm (ADPCM or PCM) and the data-storage rate for each algorithm/sampling rate combination in parenthesis (24 Kbps, 32 Kbps, 48 Kbps, or 64 Kbps).

Select the desired record feature in the first column of the table and move across that row until the column containing the desired encoding algorithm and data-storage rate is reached. The record modes that must be entered in **dx_rec( )** are provided where the features row, and encoding algorithm/data-storage rate column intersect. Parameters listed in { } are default settings and do not have to be specified.

Table 5. Record Mode Selections

Feature	ADPCM (24 Kbps)	ADPCM (32 Kbps)	PCM (48 Kbps)	PCM (64 Kbps)
• AGC • No Tone	RM_SR6 {MD_ADPCM} {MD_GAIN}	RM_SR8 {MD_ADPCM} {MD_GAIN}	RM_SR6 RM_ALAW* MD_PCM {MD_GAIN}	RM_SR8 RM_ALAW* MD_PCM {MD_GAIN}
• No AGC • No Tone	MD_NOGAIN RM_SR6 {MD_ADPCM}	MD_NOGAIN RM_SR8 {MD_ADPCM}	MD_NOGAIN RM_SR6 MD_PCM	MD_NOGAIN RM_SR8 MD_PCM
• AGC • Tone	RM_TONE RM_SR6 {MD_ADPCM} {MD_GAIN}	RM_TONE RM_SR8 {MD_ADPCM} {MD_GAIN}	RM_TONE RM_ALAW* RM_SR6 MD_PCM {MD_GAIN}	RM_TONE RM_ALAW* RM_SR8 MD_PCM {MD_GAIN}
• No AGC • Tone	MD_NOGAIN RM_TONE RM_SR6 {MD_ADPCM}	MD_NOGAIN RM_TONE RM_SR8 {MD_ADPCM}	MD_NOGAIN MD_PCM RM_SR6 RM_TONE RM_ALAW*	MD_NOGAIN MD_PCM RM_SR8 RM_TONE RM_ALAW*

{ } = Default modes.
* = Select if A-Law encoding is required (supported on D/41ESC, D/160SC-LS, D/240SC, D/240SC-T1, D/300SC-E1 and D/320SC boards only).

NOTE: **dx_rec( )** will run synchronously if you do not specify EV_ASYNC, or if you specify EV_SYNC (default).

■ Cautions

None.

■ Example 1: Using dx_rec( ) in synchronous mode

```
#include <fcntl.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
#define MAXLEN 10000
main()
{
   DV_TPT tpt;
   DX_IOTT iott[2];
   int chdev;
   char basebufp[MAXLEN];
   /*
    * open the channel using dx_open( )
    */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
     /* process error */
   }
   /*
    * Set up the DV_TPT structures for MAXDTMF
    */
   dx_clrtpt(&tpt,1);
   tpt.tp_type   = IO_EOT;          /* last entry in the table */
   tpt.tp_termno = DX_MAXDTMF;      /* Maximum digits */
   tpt.tp_length = 1;               /* terminate on the first digit */
   tpt.tp_flags  = TF_MAXDTMF;      /* Use the default flags */
   /*
    * Set up the DX_IOTT. The application records the voice data to memory
    * allocated by the user.
    */
   iott[0].io_type = IO_MEM|IO_CONT;   /* Record to memory */
   iott[0].io_bufp = basebufp;         /* Set up pointer to buffer */
   iott[0].io_offset = 0;              /* Start at beginning of buffer */
   iott[0].io_length = MAXLEN;         /* Record 10,000 bytes of voice data */
   iott[1].io_type = IO_DEV|IO_EOT;    /* Record to file, last DX_IOTT
                                        * entry */
   iott[1].io_bufp = 0;                /* Set up pointer to buffer */
   iott[1].io_offset = 0;              /* Start at beginning of buffer */
   iott[1].io_length = MAXLEN;         /* Record 10,000 bytes of voice
                                        * data */
   if((iott[1].io_fhandle = dx_fileopen("file.vox",
     O_RDWR|O_CREAT|O_TRUNC|O_BINARY,0666)) == -1)  {
     /* process error */
   }
   /* clear previously entered digits */
   if (dx_clrdigbuf(chdev) == -1) {
     /* process error */
   }
   if (dx_rec(chdev,&iott[0],&tpt,RM_TONE|EV_SYNC) == -1) {
     /* process error */
   }
   /* Analyze the data recorded */
   .
   .
}
```

■ Example 2: Using dx_rec( ) in asynchronous mode

```
#include <stdio.h>
#include <fcntl.h>
```

```
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
#define MAXLEN 10000
#define MAXCHAN 24
int record_handler();
DV_TPT tpt;
DX_IOTT iott[MAXCHAN];
int chdev[MAXCHAN];
char basebufp[MAXCHAN][MAXLEN];
main()
{
   int i, srlmode;
   char *chname;
   /* Set SRL to run in polled mode. */
   srlmode = SR_POLLMODE;
   if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1) {
      /* process error */
   }
/* Start asynchronous dx_rec() on all the channels. */
   for (i=0; i<MAXCHAN; i++) {
     /* Set chname to the channel name, e.g., dxxxB1C1, dxxxB1C2,... */
     /*
      * open the channel using dx_open( )
      */
      if ((chdev[i] = dx_open(chname,NULL)) == -1) {
        /* process error */
      }
     /* Using sr_enbhdlr(), set up handler function to handle record
      * completion events on this channel.
      */
      if (sr_enbhdlr(chdev[i], TDX_RECORD, record_handler) == -1) {
         /* process error */
      }
     /*
      * Set up the DV_TPT structures for MAXDTMF
      */
      dx_clrtpt(&tpt,1);
      tpt.tp_type   = IO_EOT;          /* last entry in the table */
      tpt.tp_termno = DX_MAXDTMF;      /* Maximum digits */
      tpt.tp_length = 1;               /* terminate on the first digit */
      tpt.tp_flags  = TF_MAXDTMF;      /* Use the default flags */
     /*
      * Set up the DX_IOTT. The application records the voice data to memory
      * allocated by the user.
      */
      iott[i].io_type = IO_MEM|IO_EOT;  /* Record to memory, last DX_IOTT
                                         * entry */
      iott[i].io_bufp = basebufp[i];    /* Set up pointer to buffer */
      iott[i].io_offset = 0;            /* Start at beginning of buffer */
      iott[i].io_length = MAXLEN;       /* Record 10,000 bytes voice data */
      /* clear previously entered digits */
      if (dx_clrdigbuf(chdev) == -1) {
        /* process error */
      }

      /* Start asynchronous dx_rec() on the channel */
      if (dx_rec(chdev[i],&iott[i],&tpt,RM_TONE|EV_ASYNC) == -1) {
        /* process error */
      }
   }
/* Use sr_waitevt to wait for the completion of dx_rec().
    * On receiving the completion event, TDX_RECORD, control is transferred
    * to a handler function previously established using sr_enbhdlr().
    */
```

```
   .
      .
}

int record_handler()
{
  long term;
  /* Use ATDX_TERMMSK() to get the reason for termination. */
  term = ATDX_TERMMSK(sr_getevtdev());
  if (term & TM_MAXDTMF) {
     printf("record terminated on receiving DTMF digit(s)\n");
  } else if (term & TM_NORMTERM) {
     printf("normal termination of dx_rec()\n");
  } else {
     printf("Unknown termination reason: %x\n", term);
  }
  /* Kick off next function in the state machine model. */
   .
   .
   return 0;
}
```

■ Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADDEV	Invalid Device Descriptor
EDX_BADPARM	Invalid Parameter
EDX_BADIOTT	Invalid *DX_IOTT* entry
EDX_BADTPT	Invalid DX_TPT entry
EDX_BUSY	Busy executing I/O function
EDX_SYSTEM	Windows NT system error - check errno

■ See Also

Related Functions:

- **dx_recf( )**
- **dx_play( )**
- **dx_playf( )**
- **dx_setparm( ), dx_getparm( )**

Setting Order and Location for Voice Data:

- *DX_IOTT* (*Chapter 4. Voice Data Structures and Device Parameters*)

Retrieving and Handling Record Termination Events:

- Event Management functions (*Standard Runtime Library Programmer's Guide for Windows NT* and *Appendix* A of this guide)
- **ATDX_TERMMSK( )**
- DV_TPT (*Appendix* A)

Name:	int dx_recf(chdev,fnamep,tptp,mode)	
Inputs:	int chdev	• valid Dialogic channel device handle
	char *fnamep	• pointer to file to record to
	DV_TPT *tptp	• pointer to Termination Parameter Table Structure
	unsigned short mode	• recording mode bit mask for this record session
Returns:	0 if success -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	Convenience	
Mode:	synchronous/Asynchronous	

■ Description

The **dx_recf( )** function permits voice data to be recorded from a channel to a single file. **dx_recf( )** performs the same as synchronous **dx_rec( )** does with a *DX_IOTT* structure that specified a single file. **dx_recf( )** is provided as a convenient method for recording to one file without having to specify a DX_IOTT structure. **dx_recf( )** opens and closes the file pointed to by fnamep while **dx_rec( )** uses a DX_IOTT structure that requires the application to open the file.

Parameter	Description
fnamep:	points to the file from to which voice data will be recorded.

For information about other function arguments and other function information, see **dx_rec( )**.

■ Source Code

```
/*****************************************************************************
 *        NAME: int dx_recf(devd,filep,tptp,mode)
 * DESCRIPTION: Record data to a file
 *      INPUTS: devd - channel descriptor
 *              tptp - TPT pointer
 *              filep - ASCIIZ string for name of file to read into
 *              mode - tone initiation flag
 *     OUTPUTS: Data stored in file, status in CSB pointed to by csbp
```

```
 *     RETURNS: 0 or -1 on error
 *       CALLS: open() dx_rec() close()
 *    CAUTIONS: none.
*****************************************************************************
*/
int dx_recf(devd,filep,tptp,mode)
   int     devd;
   char    *filep;
   DV_TPT  *tptp;
   USHORT  mode;
{
   int     rval;
   DX_IOTT iott;
   /*
    * If Async then return Error
    * Reason: IOTT's must be in scope for the duration of the record
    */
   if ( mode & EV_ASYNC ) {
      return( -1 );
   }

   /* Open the File */
   if ((iott.io_fhandle = dx_fileopen(filep,(O_WRONLY|O_CREAT|O_TRUNC),0666)) == -
       1) {
      return -1;
   }

   /* Use dx_rec() to do the record */
   iott.io_type = IO_EOT | IO_DEV;
   iott.io_offset = (long)0;
   iott.io_length = -1;

   rval = dx_rec(devd,&iott,tptp,mode);

   if (dx_fileclose(iott.io_fhandle) == -1) {
      return -1;
   }

   return rval;
}
```

■ Example

```
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   int chdev;
   long termtype;
   DV_TPT tpt[2];

   /* Open the channel using dx_open( ). Get channel device descriptor in
    * chdev
    */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
      /* process error */
```

```
   }

   /* Set the DV_TPT structures up for MAXDTMF and MAXSIL */
   dx_clrtpt(tpt,2);
   tpt[0].tp_type   = IO_CONT;
   tpt[0].tp_termno = DX_MAXDTMF;          /* Maximum digits */
   tpt[0].tp_length = 1;                   /* terminate on the first digit */
   tpt[0].tp_flags  = TF_MAXDTMF;          /* Use the default flags */


   /*
    * If the initial silence period before the first non-silence period
    * exceeds 4 seconds then terminate. If a silence period after the
    * first non-silence period exceeds 2 seconds then terminate.
    */
   tpt[1].tp_type   = IO_EOT;               /* last entry in the table */
   tpt[1].tp_termno = DX_MAXSIL;            /* Maximum silence */
   tpt[1].tp_length = 20;                   /* terminate on 2 seconds of
                                             * continuous silence */
   tpt[1].tp_flags  = TF_MAXSIL|TF_SETINIT; /* Use the default flags and
                                             * initial silence flag */
   tpt[1].tp_data   = 40;                   /* Allow 4 seconds of initial
                                             * silence */
   if (dx_recf(chdev,"weather.vox",tpt,RM_TONE) == -1) {
      /* process error */
   }
   termtype = ATDX_TERMMSK(chdev);  /* investigate termination reason */
   if (termtype & TM_MAXDTMF)  {
      /* process DTMF termination */
   }
   . . .
}
```

■ Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	Invalid Parameter
EDX_BADIOTT	Invalid *DX_IOTT* entry
EDX_BADTPT	Invalid DX_TPT entry
EDX_BUSY	Busy executing I/O function
EDX_SYSTEM	Windows NT system error - check errno

■ See Also

Related Functions:

- **dx_rec( )**
- **dx_play( )**

- **dx_playf( )**
- **dx_setparm( ), dx_getparm( )**

Setting and Handling Record Termination:

- **ATDX_TERMMSK( )**
- DV_TPT (*Appendix* A)

Name:	short dx_reciottdata(chdev, iottp, tptp, xpbp, mode)	
Inputs:	int chdev	• valid Dialogic channel device handle
	DX_IOTT *iottp	• pointer to I/O Transfer Table
	DV_TPT *tptp	• pointer to Termination Parameter Table Stucture
	DX_XPB *xpbp	• pointer to I/O Transfer Parameter block
	unsigned short mode	• play mode
Returns:	0 if success -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	I/O function	
Mode:	synchronous or asynchronous	

■ Description

The **dx_reciottdata( )** function records voice data to multiple destinations, a combination of data files, memory, or custom devices.

Parameter	Description
chdev	channel device descriptor.
iottp	Pointer to *DX_IOTT* table that specifies the order and media onto which the voice data will be recorded.
tptp	pointer to Termination Parameter Table structure
xpbp	pointer to I/O transfer parameter block
mode	specifies the record mode: PM_TONE play 200 ms audible tone EV_SYNCH synchronous mode EV_ASYNCH asynchronous mode

■ Cautions

1. All files specified in the *DX_IOTT* table will be of the file format described in *DX_XPB*.
2. All files recorded to will have the data encoding and rate as described in DX_XPB.
3. When playing or recording VOX files, the data format is specified in DX_XPB rather than through the **dl_stprm( )** function.

■ Example

```
#include "srllib.h"
#include “dxxxlib.h”

int chdev;        /* channel descriptor */
int fd;           /* file descriptor for file to be played */
DX_IOTT iott;     /* I/O transfer table */
DV_TPT tpt;       /* termination parameter table */
DX_XPB xpb;       /* I/O transfer parameter block */

.
.

/* Open channel */
if ((chdev = dx_open("dxxxB1C1",0)) == -1) {
      printf("Cannot open channel\n");
      printf("errno = %d\n",errno);
      exit(1);
}
/* Set to terminate play on 1 digit */
tpt.tp_type   = IO_EOT;
tpt.tp_termno = DX_MAXDTMF;
tpt.tp_length = 1;
tpt.tp_flags  = TF_MAXDTMF;
/* Open file */
if ((fd = dx_fileopen("MESSAGE.VOX",O_RDWR|O_BINARY)) == -1) {
      printf("File open error\n");
      exit(2);
}
/* Set up DX_IOTT */
iott.io_fhandle = fd;
iott.io_bufp    = 0;
iott.io_offset  = 0;
iott.io_length  = -1;
iott.io_typ = IO_DEV | IO_EOT;

/*
 * Specify VOX file format for PCM at 8KHz.
 */
xpb.wFileFormat = FILE_FORMAT_VOX;
xpb.wDataFormat = DATA_FORMAT_PCM;
xpb.nSamplesPerSec = DRT_8KHZ;
xpb.nBitsPerSample = 8;

/* Wait forever for phone to ring and go offhook */
if (dx_wtring(chdev,1,DX_OFFHOOK,-1) == -1) {
      printf("Error waiting for ring - %s\n",  ATDV_LASTERR(chdev));
```

```
      exit(3);
}
/* Play intro message */
if (dx_playwav(chdev,&tpt,"HELLO.WAV",EV_SYNC) == -1) {
      printf("Error playing file - %s\n",      ATDV_ERRMSGP(chdev));
      exit(4);
}

/* Start recording */
if (dx_reciottdata(chdev,&iott,&tpt,&xpb,PM_TONE|EV_SYNC) == -1) {
      printf("Error recording file - %s\n",    ATDV_ERRMSGP(chdev));
      exit(4);
}
```

■ Errors

In asynchronous mode, function returns immediately and a TDX_RECORD event is queued upon completion. Check **ATDX_LASTTERM( )** for the termination reason. If a failure occurs, then a TDX_ERROR event will be queued. Use **ATDV_LASTERR( )** to determine the reason for error.

In synchronous mode, if this function returns -1 to indicate failure, one of the following reasons will be contained by **ATDV_LASTERR( )** :

Equate	Returned When
EDX_BUSY	Channel is busy
EDX_XPBPARM	Invalid *DX_XPB* setting
EDX_BADIOTT	Invalid *DX_IOTT* setting
EDX_SYSTEM	System I/O errors
EDX_BADWAVFILE	Invalid WAV file
EDX_SH_BADCMD	Unsupported command or WAV file format

■ See Also

- **dx_recwav( )**
- **dx_recvox( )**

Name:	SHORT dx_recvox(chdev, filenamep, tptp, xpbp, mode)	
Inputs:	int chdev	• valid Dialogic channel device handle
	char *filenamep	• pointer to name of file to record to
	DV_TPT*tptp	• pointer to Termination Parameter Table
	DX_XPB *xpbp	• pointer to I/O Transfer Parameter Block
	unsigned short mode	• play mode
Returns:	0 if successful -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	Convenience function	
Mode:	synchronous	

■ Description

The **dx_recvox( )** convenience function records voice data to a single VOX file. If **xpbp** is set to NULL, it will interpret the data as 6KHz linear ADPCM.

Parameter	Description
chdev	Channel device descriptor
tcbp	Pointer to Termination Parameter Table
filenamep	Pointer to name of file to record to
xpbp	Pointer to I/O Transfer Parameter Block (See the *DX_XPB* data structure)
mode	specifies the play mode: PM_TONE play 200 ms audible tone EV_SYNC synchronous operation (must be specified)

NOTE: Both PM_TONE and EV_SYNC can be specified by ORing the two values.

■ Cautions

When playing or recording VOX files, the data format is specified in *DX_XPB* rather than through the mode parameter of **dx_recvox( )**.

■ Example

```
#include "srllib.h"
#include "dxxxlib.h"

int chdev;                  /* channel descriptor */
DV_TPT tpt;                 /* termination parameter table */
DX_XPB xpb;                 /* I/O transfer parameter block */
.
.
.
/* Open channel */
if ((chdev = dx_open("dxxxB1C1",0)) == -1) {
      printf("Cannot open channel\n");
      printf("errno = %d\n",errno);
      exit(1);
}
/* Set to terminate play on 1 digit */
tpt.tp_type   = IO_EOT;
tpt.tp_termno = DX_MAXDTMF;
tpt.tp_length = 1;
tpt.tp_flags  = TF_MAXDTMF;
/* Wait forever for phone to ring and go offhook */
if (dx_wtring(chdev,1,DX_OFFHOOK,-1) == -1) {
      printf("Error waiting for ring - %s\n", ATDV_LASTERR(chdev));
      exit(3);
}
/* Start playback */
if (dx_playwav(chdev,&tpt,"HELLO.WAV",EV_SYNC) == -1) {
      printf("Error playing file - %s\n", ATDV_ERRMSGP(chdev));
      exit(4);
}
/* clear digit buffer */
dx_clrdigbuf(chdev);
/* Start 6KHz ADPCM recording */
if (dx_recvox(chdev,"MESSAGE.VOX", &tpt, NULL,PM_TONE|EV_SYNC) == -1) {
      printf("Error recording file - %s\n", ATDV_ERRMSGP(chdev));
      exit(4);
}
```

■ Errors

If this function returns -1 to indicate failure, one of the following reasons will be contained by **ATDV_LASTERR( )** :

Equate	Returned When
EDX_BUSY	Channel is busy

Equate	Returned When
EDX_XPBPARM	Invalid *DX_XPB* setting
EDX_BADIOTT	Invalid *DX_IOTT* setting
EDX_SYSTEM	System I/O errors
EDX_SH_BADCMD	Unsupported command or VOX file format

■ See Also

- **dx_reciottdata( )**
- **dx_recwav( )**

Name: SHORT dx_recwav(chdev, filenamep, tptp, xpbp, mode)

Inputs:	int chdev	• valid Dialogic channel device handle
	char *filenamep	• pointer to name of file to record to
	DV_TPT *tptp	• pointer to termination parameter block
	DX_XPB *xpbp	• pointer to I/O Transfer Block
	unsigned short mode	• record mode
Returns:	0 if successful -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	Convenience function	
Mode:	synchronous	

■ Description

The **dx_recwav( )** convenience function records voice data to a single WAVE file. If **xpbp** is set to NULL, the function will record in 11 KHz linear 8-bit PCM. This function calls **dx_reciottdata( )**.

Parameter	Description
chdev	channel device descriptor
tcbp	pointer to termination parameter table
filenamep	pointer to name of file to play
xpbp	pointer to I/O Transfer Parameter Block
mode	specifies the play mode: PM_TONE play 200 ms audible tone EV_SYNC synchronous operation (must be specified)

NOTE: Both PM_TONE and EV_SYNC can be specified by ORing the two values.

■ Cautions

None.

■ Example

```
#include "srllib.h"
#include “dxxxlib.h”

int chdev;              /* channel descriptor */
DV_TPT tpt;             /* termination parameter table */
DX_XPB xpb;             /* I/O transfer parameter block */
.
.
.
/* Open channel */
if ((chdev = dx_open("dxxxB1C1",0)) == -1) {
      printf("Cannot open channel\n");
      printf("errno = %d\n",errno);
      exit(1);
}
/* Set to terminate play on 1 digit */
tpt.tp_type   = IO_EOT;
tpt.tp_termno = DX_MAXDTMF;
tpt.tp_length = 1;
tpt.tp_flags  = TF_MAXDTMF;
/* Wait forever for phone to ring and go offhook */
if (dx_wtring(chdev,1,DX_OFFHOOK,-1) == -1) {
      printf("Error waiting for ring - %s\n", ATDV_LASTERR(chdev));
      exit(3);
}
/* Start playback */
if (dx_playwav(chdev,&tpt,"HELLO.WAV",EV_SYNC) == -1) {
      printf("Error playing file - %s\n", ATDV_ERRMSGP(chdev));
      exit(4);
}
/* clear digit buffer */
dx_clrdigbuf(chdev);
/* Start 11KHz PCM recording */
if (dx_recwav(chdev,"MESSAGE.WAV", &tpt,   (DX_XPB *)NULL,PM_TONE|EV_SYNC) == -1) {
      printf("Error recording file - %s\n", ATDV_ERRMSGP(chdev));
      exit(4);


}
```

■ Errors

If this function returns -1 to indicate failure, one of the following reasons will be contained by **ATDV_LASTERR( )** :

Equate	Returned When
EDX_BUSY	Channel is busy

Equate	Returned When
EDX_XPBPARM	Invalid *DX_XPB* setting
EDX_BADIOTT	Invalid *DX_IOTT* setting
EDX_SYSTEM	System I/O errors
EDX_BADWAVFILE	Invalid WAV file
EDX_SH_BADCMD	Unsupported command or WAV file format

■ See Also

- **dx_reciottdata( )**
- **dx_recvox( )**

Name:	int dx_setdigbuf(chdev,mode)	
Inputs:	int chdev	• valid Dialogic channel device handle
	int mode	• digit buffering mode
Returns:	0 if successful -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	I/O	
Mode:	synchronous	

■ Description

The **dx_setdigbuf( )** function sets the digit buffering mode that will be used by the Voice Driver. Once the digit buffer is full (31 digits), the application may select whether subsequent digits will be ignored or will overwrite the oldest digits in the queue.

The function parameters are defined as follows:

Parameter	Description	
chdev:	specifies the valid Dialogic channel device handle obtained by a call to **dx_open( )**.	
mode:	specifies the type of digit buffering that will be used. Mode can be:	
	• DX_DIGTRUNC	Incoming digits will be ignored if the digit buffer is full (default).
	• DX_DIGCYCLIC	Incoming digits will overwrite the oldest digits in the buffer if the buffer is full.

■ Cautions

None.

Example

```
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

int chfd;

int init_digbuf()
{

    /* open the device using dx_open, chfd has the device handle */

    /*
     * Set up digit buffering to be Cyclic. When digit
     * queue overflows oldest digit will be overwritten
     */
    if (dx_setdigbuf(chfd, DX_DIGCYCLIC) == -1) {
       printf("Error during setdigbuf %s\n", ATDV_ERRMGSP(chfd));
       return(1);
    }
    return(0);
}
```

Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	• Invalid Parameter
EDX_SYSTEM	• Windows NT system error - check **errno**
EDX_TIMEOUT	• Timeout limit is reached

Name:	int dx_setdigtyp(chdev,dmask)	
Inputs:	int chdev	• valid Dialogic channel device handle
	unsigned short dmask	• type of digit the channel will detect
Returns:	0 if successful -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	Configuration	

■ Description

The **dx_setdigtyp( )** function controls the types of digits the Voice channel detects.

NOTE: This function only applies to the standard Voice board digits (i.e., DTMF, MF, LPD). To set user-defined digits, use the **dx_addtone( )** function.

dx_setdigtyp( ) does not clear the previously detected digits in the digit buffer.

The function parameters are defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.
dmask:	sets the type of digits the channel will detect. More than one type of digit detection can be enabled in single function call, as shown in the function example. **dmask** can have one, or a combination of several, of the following values: DM_DTMF: enable DTMF digits; detection (default setting) DM_LPD: enable loop pulse detection DM_APD: enable audio pulse digits detection DM_MF: enable MF digit detection DM_DPD: enable dial pulse digit (DPD) detection

Parameter	Description
	DM_DPD2: enable zero train DPD detection To disable digit detection, set **dmask** to NULL.

NOTES: 1. MF detection can only be enabled on systems with MF capability, such as D/4xD boards with MF support.

2. The digit detection type specified in **dmask** will remain valid after the channel has been closed and reopened.

dx_setdigtyp( ) disables any digit detection enabled in a previous call to **dx_setdigtyp( )**.

■ Cautions

1. Some MF digits use approximately the same frequencies as DTMF digits (see *Appendix* C). Because there is a frequency overlap, if you have the incorrect kind of detection enabled, MF digits may be mistaken for DTMF digits, and vice versa. To ensure that digits are correctly detected, DTMF and MF detection should not be enabled at the same time.

■ Example

```
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>


main()
{
   int chdev;
   .
   .
   /* Open Voice channel */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
     /* process error */
   }

   /* Set channel to detect DTMF and loop pulse digits */
   if (dx_setdigtyp(chdev, DM_DTMF|DM_LPD) == -1) {
     /* error routine */
   }
   .
   .
}
```

■ Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	• Invalid Parameter
EDX_SYSTEM	• Windows NT system error - check **errno**

■ See Also

Specifying user-defined digits:

- **dx_addtone( )**

Name:	int dx_setevtmsk(chdev,mask)	
Inputs:	int chdev	• valid Dialogic channel device handle
	unsigned int mask	• event mask of events to wait for
Returns:	0 if successful -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	Call Status Transition Event	

■ Description

The **dx_setevtmsk( )** function enables detection of Call Status Transition (CST) event or group of events. This function can be used by synchronous or asynchronous applications waiting for a CST event.

NOTE: This function enables detection for all CST events except user-defined tone detection. See **dx_addtone( )** and **dx_enbtone( )** for information.

The function parameters are defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.
mask:	specifies the event(s) **dx_getevt( )** will wait for. mask can have one or more of the following values: DM_LCOFF • wait for loop current to be off DM_LCON • wait for loop current to be on DM_RINGS • wait for rings DM_RNGOFF • wait for ring to drop (hang-up) DM_SILOF • wait for non-silence DM_SILON • wait for silence DM_WINK • wait for wink to occur on an E&M line DM_DIGITS • turns off digit reporting and digits flag set by DM_DIGITS DM_LCREV • wait for flow of current to reverse (D/41ESC and D/160SC-LS boards

Parameter	Description
	only)
	When the event mask is set with DM_DIGITS, a digits flag is set that causes individual digit events to queue until this flag is turned off by the DM_DIGOFF equate. Setting the event mask for DM_DIGITS and then subsequently resetting the event mask without DM_DIGITS does not disable the queueing of digit events. Digit events will remain in the queue until collected by **dx_getevt( )**. This queue is not affected by **dx_getdig( )** calls. The digits flag is set by:

```
/* Set event mask to collect digits */
if (dx_setevtmsk(chdev, DM_DIGITS) == -1) {
```

To turn off the digits flag and stop queueing digits:

```
dx_setevtmsk(DM_DIGOFF);
dx_cirdigbuf(chdev);  /*Clear out queue*/
```

To poll for multiple events, perform an OR operation on the bit masks of the events you want to wait for. The first enabled CST event to occur will be returned. On the D/21E, D/41E, or D/160SC-LS board, when the DM-LCREV bit is OR'ed, a new event message DE_LCREV is queued when the flow of current over the line is reversed.

For configurations using the DID/120 Direct Inward Dialing board, you may need to check whether the caller has hung-up after the DID/120 board received the DNIS digits. DID systems often get the digits for a call back without ever going off hook. You can check for hang up as follows:

In asynchronous mode: look for a ring off event using **dx_setevtmsk(DM_RNGOFF)** and **dx_getevt( )** to get the event. After the event is received, issue a **dx_stopch( )** to stop the current multitasking function.

Alternatively, you can poll using **ATDX_LINEST( )** to determine when ring is not present (returns RLS_RING)

Parameter	Description
	and then issue a **dx_stopch( )** to stop the current multitasking function.

The following table outlines the synchronous or asynchronous handling of CST events:

	Synchronous	Asynchronous
1.	Call **dx_setevtmsk( )** to enable CST event(s)	Call **dx_setevtmsk( )** to enable CSTevent(s)
2.	Call **dx_getevt( )** to wait for CST event(s). Events are returned to the *DX_EBLK* structure.	Use SRL to asynchronously wait for for TDX_CST event(s).
3.		Use **sr_getevtdatap( )** to retrieve *DX_CST* structure.

NOTE: If DM_WINK is not specified in the **mask** parameter, and DM_RINGS is specified, a wink will be interpreted as an incoming call depending on the setting of the DXBD_R_ON parameter.

■ Cautions

1. Events are preserved between **dx_getevt( )** function calls. The event that was set remains the same, until another call to **dx_setevtmsk( )** or **dx_wtring( )** changes it. See **dx_wtring( )** for more information on how it changes the event mask.
2. If you call this function on a busy device, and specify DM_DIGITS as the **mask** argument, the function will fail.

■ Example 1: Using dx_setevtmsk( ) to wait for ring events - synchronous processing

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
```

```
main()
{
   int chdev;
   DX_EBLK eblk;
   .
   .
   /* open a channel with chdev as descriptor */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
     /* process error */
   }
   .
   .
   /* Set event mask to receive ring events */
   if (dx_setevtmsk(chdev, DM_RINGS) == -1) {
     /* error setting event */
   }
   .
   .
   /* check for ring event, timeout set to 20 seconds */
   if (dx_getevt(chdev,&eblk,20) == -1) {
     /* error timeout */
   }

   if(eblk.ev_event==DE_RINGS) {
      printf("Ring event occurred\n");
   }
   .
   .
}
```

■ Example 2: Using dx_setevtmsk( ) to handle call status transition events - asynchronous processing

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

#define MAXCHAN 24

int cst_handler();

main()
{
   int chdev[MAXCHAN];
   char *chname;
   int i, srlmode;

   /* Set SRL to run in polled mode. */
   srlmode = SR_POLLMODE;
   if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1) {
      /* process error */
   }

   for (i=0; i<MAXCHAN; i++) {

      /* Set chname to the channel name, e.g., dxxxB1C1, dxxxB1C2,... */
```

```
      /* Open the device using dx_open( ).  chdev[i] has channel device
       * descriptor.
       */
       if ((chdev[i] = dx_open(chname,NULL)) == -1)   {
          /* process error */
       }

      /* Use dx_setevtmsk() to enable call status transition events
       * on this channel.
       */
       if (dx_setevtmsk(chdev[i],
           DM_LCOFF|DM_LCON|DM_RINGS|DM_SILOFF|DM_SILON|DM_WINK) == -1) {
          /* process error */
       }

      /* Using sr_enbhdlr(), set up handler function to handle call status
       * transition events on this channel.
       */
       if (sr_enbhdlr(chdev[i], TDX_CST, cst_handler) == -1) {
          /* process error */
       }
      /* Use sr_waitevt to wait for call status transition event.
       * On receiving the transition event, TDX_CST, control is transferred
       * to the handler function previously established using sr_enbhdlr().
       */
                   .
                   .
   }
}

int cst_handler()
{
   DX_CST *cstp;

  /* sr_getevtdatap() points to the event that caused the call status
   * transition.
   */
   cstp = (DX_CST *)sr_getevtdatap();

   switch (cstp->cst_event) {
     case DE_RINGS:
      printf("Ring event occurred on channel %s\n",
              ATDX_NAMEP(sr_getevtdev()));
      break;
     case DE_WINK:
      printf("Wink event occurred on channel %s\n",
              ATDX_NAMEP(sr_getevtdev()));
      break;
     case DE_LCON:
      printf("Loop current ON event occurred on channel %s\n",
              ATDX_NAMEP(sr_getevtdev()));
      break;
     case DE_LCOFF:
               .
               .
   }

   /* Kick off next function in the state machine model. */
   .
   .
```

```
   return 0;
}
```

■ Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	• Invalid Parameter
EDX_SYSTEM	• Windows NT system error - check **errno**

■ See Also

CST Event Handling and Retrieval:

- **dx_getevt( )** - synchronous operation
- **sr_getevtdatap( )** - asynchronous operation (*Standard Runtime Library Programmer's Guide for Windows NT*)
- *DX_CST* data structure

Enabling User-Defined Tone Detection:

- **dx_addtone( )**

Name: void dx_setgtdamp(gtd minampl1, gtd maxampl1, gtd minampl2, gtd maxampl2)

Inputs:

short int gtd_minampl1	• Minimum amplitude of the first frequency
short int gtd_maxampl1	• Maximum amplitude of the first frequency
short int gtd_minampl2	• Minimum amplitude of the second frequency
short int gtd_maxampl2	• Maximum amplitude of the second frequency

Returns: void

Includes: srllib.h
dxxxlib.h

Category: GTD Function

■ Description

The **dx_setgtdamp( )** function sets up the amplitudes to be used by the general tone detection. This function must be called before calling **dx_bld...( )** functions and **dx_addtone( )**. Once called, the values set will take effect for all **dx_bld...( )** function calls.

If this function is not called, then the MINERG firmware parameters that were downloaded remain at the following settings: -42dBm for minimum amplitude and 0dBm for maximum amplitude.

Default Value	Description
GT_MIN_DEF	Default value in dB for minimum GTD amplitude that can be entered for **gtd_minampl*** parameters.
GT_MAX_DEF	Default value in dB for maximum GTD amplitude that can be entered for **gtd_maxampl*** parameters.

Parameter	Description
gtd minampl1:	specifies the minimum amplitude in dB of tone 1.

Parameter	Description
gtd maxampl1:	specifies the maximum amplitude in dB of tone 1.
gtd minampl2:	specifies the minimum amplitude in dB of tone 2.
gtd maxampl2:	specifies the maximum amplitude in dB of tone 2.

■ Cautions

If this function is called, then the amplitudes set will take effect for all tones added afterwards. To reset the amplitudes back to the defaults, then call this function with the defines GT_MIN_DEF and GT_MAX_DEF for minimum and maximum defaults.

■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
#include "voxlib.h"     /* Dialogic voice library header file */

#define TID 1;          /* Tone ID */

.
.
.
/*
 * Set amplitude for GTD;
 *     freq1 -30dBm to 0 dBm
 *     freq2 -30dBm to 0 dBm
 */
dx_setgtdamp(-30,0,-30,0);

/*
 * Build temporary simple dual tone frequency tone of
 * 950-1050 Hz and 475-525 Hz. using trailing edge detection, and
 * -30dBm to 0dBm.
if (dx_blddt(TID1, 1000, 50, 500, 25, TN LEADING) ==-1) {
  printf("Error building temporary tone: %d\n",dx_errno);
  exit(3);
}

.
.
.
```

Errors

None.

Name:	int dx_sethook(chdev,hookstate,mode)	
Inputs:	int chdev	• valid Dialogic channel device handle
	int hookstate	• hook state (on-hook or off-hook)
	unsigned short mode	• asynchronous/synchronous
Returns:	0 if successful -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	Configuration	
Mode:	asynchronous/synchronous	

■ Description

The **dx_sethook( )** function provides control of the hookswitch status of the specified channel. A hookswitch state may be either on-hook or off-hook.

NOTE: Do not call this function for a digital T-1 SCbus configuration that includes a D/240SC, D/240SC-T1, DTI/241SC, or DTI/301SC board. Transparent signaling for SCbus digital interface devices is not supported in System Release 4.1SC.

Parameter	Description
dx_sethook( )	clears loop current and silence history from the channel's buffers.

■ Asynchronous Operation

To run **dx_sethook( )** asynchronously, set the **mode** field to EV_ASYNC. The function will return 0 to indicate it has initiated successfully, and will generate a termination event to indicate completion. Use the SRL Event Management functions to handle the termination event.

If running asynchronously, termination is indicated by a TDX_SETHOOK event. The **cst_event** field in the data structure will specify one of the following:

- DX_ONHOOK if the hookstate has been set to on-hook
- DX_OFFHOOK if the hookstate has been set to off-hook

(Use the Event Management function **sr_getevtdatap( )** to return a pointer to the *DX_CST* structure). See *Appendix* A for more information about the Event Management functions.

ATDX_HOOKST( ) will also return the type of hookstate event.

■ Synchronous Operation

By default, this function runs synchronously.

If running synchronously (default) **dx_sethook( )** will return 0 when complete.

The function parameters are defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.
hookstate:	forces the **hookstate** of the specified channel to on-hook or off-hook. The following values can be specified: DX_ONHOOK: set to on-hook state DX_OFFHOOK: set to off-hook state
mode:	specifies whether to run **dx_sethook( )** asynchronously or synchronously. Specify one of the following: EV_ASYNC: Run **dx_sethook( )** asynchronously. EV_SYNC: Run **dx_sethook( )** synchronously (default).

■ Cautions

None.

■ Example 1: Using dx_sethook( ) in synchronous mode

```
#include <srllib.h>
#include <dxxxlib.h>
```

```
#include <windows.h>

main()
{
   int chdev;
   /* open a channel with chdev as descriptor */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
     /* process error */
   }

   /* put the channel on-hook */
   if (dx_sethook(chdev,DX_ONHOOK,EV_SYNC) == -1) {
     /* error setting hook state */
   }
   .
   .
   /* take the channel off-hook */
   if (dx_sethook(chdev,DX_OFFHOOK,EV_SYNC) == -1) {
     /* error setting hook state */
   }
   .
   .
}
```

■ Example 2: Using dx_sethook( ) in asynchronous mode

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

#define MAXCHAN 24

int sethook_hdlr();

main()
{
   int i, chdev[MAXCHAN];
   char *chnamep;
   int srlmode;

   /* Set SRL to run in polled mode. */
   srlmode = SR_POLLMODE;
   if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1) {
      /* process error */
   }

   for (i=0; i<MAXCHAN; i++) {

      /* Set chnamep to the channel name - e.g, dxxxB1C1, dxxxB1C2,... */

      /* open a channel with chdev[i] as descriptor */
      if ((chdev[i] = dx_open(chnamep,NULL)) == -1) {
         /* process error */
      }
```

```
      /* Using sr_enbhdlr(), set up handler function to handle sethook
       * events on this channel.
       */
      if (sr_enbhdlr(chdev[i], TDX_SETHOOK, sethook_hdlr) == -1) {
         /* process error */
      }

      /* put the channel on-hook */
      if (dx_sethook(chdev[i],DX_ONHOOK,EV_ASYNC) == -1) {
         /* error setting hook state */
      }
   }

   /* Use sr_waitevt() to wait for the completion of dx_sethook().
    * On receiving the completion event, TDX_SETHOOK, control is transferred
    * to the handler function previously established using sr_enbhdlr().
    */
   .
   .
}

int sethook_hdlr()
{
   DX_CST *cstp;

   /* sr_getevtdatap() points to the call status transition
    * event structure, which contains the hook state of the
    * device.
    */
   cstp = (DX_CST *)sr_getevtdatap();

   switch (cstp->cst_event) {
   case DX_ONHOOK:
      printf("Channel %s is ON hook\n", ATDX_NAMEP(sr_getevtdev()));
      break;
   case DX_OFFHOOK:
      printf("Channel %s is OFF hook\n", ATDX_NAMEP(sr_getevtdev()));
      break;
   default:
      /* process error */
      break;
   }

   /* Kick off next function in the state machine model. */
   .
   .

   return 0;
}
```

■ Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	• Invalid Parameter

EDX_SYSTEM	• Windows NT system error - check **errno**

See Also

- *DX_CST* structure
- **sr_getevtdatap( )** (*Standard Runtime Library Programmer's Guide for Windows NT*, and *Appendix* A)
- **ATDX_HOOKST( )**
- DV_TPT (*Appendix* A)

Name:	int dx_setparm(dev,parm,valuep)	
Inputs:	int dev	• valid Dialogic channel or board device handle
	unsigned long parm	• parameter type to set
	void *valuep	• pointer to parameter value
Returns:	0 if successful -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	Configuration	

■ Description

The **dx_setparm( )** function allows you to set the physical parameters of a channel or board device, such as off-hook delay, length of a pause, and flash character. Parameters can be set only one at a time. The possible values of **parm** are defined in (*Chapter 4. Voice Data Structures and Device Parameters*).

The channel must be idle (i.e., no I/O function running) when calling **dx_setparm( )**. Board and channel resources have different parameters that can be set. Setting board parameters affects all the channels on the board. Setting channel parameters only affects the specified channel.

To set board parameters the following requirements must be met:

- the board must be open
- all channels on the board must be closed

The function parameters are defined as follows:

Parameter	Description
dev:	specifies the valid channel or board device handle obtained when the channel or board was opened using **dx_open( )**.
parm:	specifies the channel or board parameter to set. Board and channel parameter defines, defaults and descriptions are listed in *Section 5.2.* Clearing Voice Structures

NOTE: The parameters set in **parm** will remain valid after the device has

Parameter	Description
	been closed and reopened.
valuep:	points to the variable that specifies the channel or board parameter to set.

NOTE: You must use a void* cast on the address of the parameter being sent to the driver in **valuep** as shown in the example.

■ Cautions

1. A constant cannot be used in place of **valuep**. The value of the parameter to be set must be placed in a variable and the address of the variable cast as void * must be passed to the function.
2. When setting channel parameters, the channel must be open and in the idle state.
3. When setting board parameters, all channels on that board must be idle.

■ Example

```
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   int bddev, parmval;
   /* Open the board using dx_open( ). Get board device descriptor in
    * bddev.
    */
   if ((bddev = dx_open("dxxxB1",NULL)) == -1) {
      /* process error */
   }
   /* Set the inter-ring delay to 6 seconds (default = 8) */
   parmval = 6;
   if (dx_setparm(bddev, DXBD_R_IRD, (void *)&parmval) == -1) {
      /* process error */
   }
   /* now wait for an incoming ring */
   . . .
}
```

■ Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	• Invalid Parameter
EDX_SYSTEM	• Windows NT system error - check **errno**

See Also

- **dx_getparm( )**

Name:	int dx_setsvcond(chdev, numblk, svcbp)	
Inputs:	int chdev	• valid channel device handle
	unsigned short numblk	• number of *DX_SVCB* blocks
	DX_SVCB * svcbp	• pointer to array of DX_SVCB structures
Returns:	0 if success -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	Speed and Volume	

■ Description

The **dx_setsvcond( )** function sets adjustments and adjustment conditions for all subsequent plays on the specified channel (until changed or cancelled). An *adjustment* is a modification to play-speed or play-volume that takes place due to an *adjustment condition* such as start of play, or the occurrence of an incoming digit during play. This function uses the specified channel's speed or volume Speed/Volume Modification Table. Detailed information about these tables are contained in the *Voice Features Guide for Windows NT*.

NOTE: Calls to **dx_setsvcond( )** are cumulative. If adjustment blocks have been set previously, calling this function adds more adjustment blocks to the list. To replace existing adjustment blocks, clear the current set of blocks using **dx_clrsvcond( )** before issuing a **dx_setsvcond( )**.

The following adjustments and adjustment conditions are defined in the Speed/Volume Adjustment Condition Blocks (*DX_SVCB*):

- which Speed/Volume Modification Table to use (speed or volume)
- adjustment type (increase/decrease, absolute value, toggle)
- adjustment conditions (incoming digit, beginning of play)
- level/edge sensitivity for incoming digits

See 5.2. Clearing Voice Structures for a full description of the *DX_SVCB* structure. Up to 20 DX_SVCB blocks can be specified in the form of an array.

NOTES: **1.** This function is similar to **dx_adjsv( )**. Use **dx_adjsv( )** to explicitly adjust the play immediately and use **dx_setsvcond( )** to adjust the

play in response to specified conditions. See the description of **dx_adjsv( )** for more information.

2. Whenever the play is started its speed and volume is based on the most recent modification.

Parameter	Description
chdev:	specifies the valid channel device handle obtained by a call to **dx_open( )**.
numblk:	specifies the number of *DX_SVCB* blocks in the array. Set to a value between 1 and 20.
svcbp:	points to an array of DX_SVCB structures.

■ Cautions

1. Condition blocks can only be added to the array (up to a maximum of 20). To reset or remove any condition, you should clear the whole array, and reset all conditions if required. (e.g., If DTMF digit "1" has already been set to increase play-speed by one step, a second call that attempts to redefine "1" to the origin, will have no affect. The digit will retain its original setting).
2. The digit that causes the play adjustment will not be passed to the digit buffer, so it cannot be retrieved using **dx_getdig( )** or **ATDX_BUFDIGS( )**.
3. 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.
4. Speed and volume control is supported on the D/21D, D/21E, D/41D, D/41E, D/41ESC, D/81A, D/121B, D/160SC-LS, D/240SC, D/240SC-T1, D/300SC-E1 and D/320SC boards only. Do not use the Speed and Volume control functions to control speed on the D/120, D/121, or D/121A boards.

■ Example

```
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

/*
 * Global Variables
 */
```

```
DX_SVCB svcb[ 10 ] = {
   /*  BitMask  AjustmentSize  AsciiDigit  DigitType */
   { SV_SPEEDTBL | SV_RELCURPOS,   1, '1', 0 },   /* 1 */
   { SV_SPEEDTBL | SV_ABSPOS,     -4, '2', 0 },   /* 2 */
   { SV_VOLUMETBL | SV_ABSPOS,     1, '3', 0 },   /* 3 */
   { SV_SPEEDTBL | SV_ABSPOS,      1, '4', 0 },   /* 4 */
   { SV_SPEEDTBL | SV_ABSPOS,      1, '5', 0 },   /* 5 */
   { SV_VOLUMETBL | SV_ABSPOS,     1, '6', 0 },   /* 6 */
   { SV_SPEEDTBL | SV_RELCURPOS,  -1, '7', 0 },   /* 7 */
   { SV_SPEEDTBL | SV_ABSPOS,      6, '8', 0 },   /* 8 */
   { SV_VOLUMETBL | SV_RELCURPOS, -1, '9', 0 },   /* 9 */
   { SV_SPEEDTBL | SV_ABSPOS,      10, '0', 0 },   /* 10 */ };

main()
{
   int  dxxxdev;

   /*
    * Open the Voice Channel Device and Enable a Handler
    */
   if ( ( dxxxdev = dx_open( "dxxxB1C1", NULL) ) == -1 ) {
      perror( "dxxxB1C1" );
      exit( 1 );
   }

   /*
    * Set Speed and Volume Adjustment Conditions
    */
   if ( dx_setsvcond( dxxxdev, 10, svcb ) == -1 ) {
      printf( "Unable to Set Speed and Volume" );
      printf( " Adjustment Conditions\n" );
      printf( "Lasterror = %d  Err Msg = %s\n",
         ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
      dx_close( dxxxdev );
      exit( 1 );
   }

   /*
    * Continue Processing
    *   .
    *   .
    *   .
    */

   /*
    * Close the opened Voice Channel Device
    */
   if ( dx_close( dxxxdev ) != 0 ) {
      perror( "close" );
   }
   /* Terminate the Program */
   exit( 0 );
}
```

■ Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	• Invalid Parameter
EDX_BADPROD	• Function not supported on this board
EDX_SVADJBLKS	• Invalid Number of Speed/Volume Adjustment blocks
EDX_SYSTEM	• Windows NT system error - check **errno**

■ See Also

Setting Speed and Volume conditions:

- **dx_clrsvcond( )**
- *DX_SVCB* (*Chapter 4. Voice Data Structures and Device Parameters*)

Related to Speed and Volume:

- **dx_setsvmt( )**
- **dx_getcursv( )**
- **dx_getsvmt( )**
- "Speed and Volume Modification Tables" (*Voice Features Guide for Windows NT*)
- **dx_adjsv( )**

Name:	int dx_setsvmt(chdev,tabletype,svmtp,flag)	
Inputs:	int chdev	• valid channel device handle
	unsigned short tabletype	• table to update (speed or volume)
	DX_SVMT * svmtp	• pointer to *DX_SVMT*
	unsigned short flag	• optional modification flag
Returns:	0 if success -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	Speed and Volume	

■ Description

The **dx_setsvmt( )** function updates the speed or volume Speed/Volume Modification Table for a channel, with the values contained in a specified *DX_SVMT* structure.

NOTE: Refer to the *Voice Features Guide for Windows NT* for a detailed description of the Speed and Volume Modification Tables, including a description of default values and refer to *Section 4.1.6.* DX_SVMT - *speed/volume modification table structure* for a description of the DX_SVMT structure.

This function can also modify the Speed or Volume Modification Table to do one of the following:

- When speed or volume adjustments reach their highest or lowest value, wrap the next adjustment to the extreme opposite value. For example, if volume reaches a maximum level during a play, the next adjustment would modify the volume to its minimum level.
- Reset the Speed/Volume Modification Table to its default values. Defaults are listed in the *Voice Features Guide for Windows NT* which describes the Speed and Volume Modification Tables in full detail.

Parameter	Description
chdev:	specifies the valid channel device handle obtained by a call to **dx_open( )**.

Parameter	Description	
tabletype:	specifies whether to retrieve the Speed or the Volume Modification Table.	
	SV_SPEEDTBL	Update the Speed Modification Table values
	SV_VOLUMETBL	Update the Volume Modification Table values
svmtp:	points to the *DX_SVMT* structure whose contents are used to update either the speed or the volume Speed/Volume Modification Table.	
	This structure is not used when SV_SETDEFAULT has been set in the **mode** parameter.	
flag:	specifies one of the following:	
	SV_WRAPMOD	Wrap around the speed or volume adjustments that occur at the top or bottom of the Speed/Volume Modification Table.
	SV_SETDEFAULT	Reset the table to its default values. See the *Voice Features Guide for Windows NT* for the default values of the table. In this case, the *DX_SVMT* pointed to by **svmtp** is ignored

NOTE: Set **flags** to 0 If you do not want to use either SV_WRAPMOD or SV_SETDEFAULT.

■ Cautions

Speed and volume control are supported on D/21D, D/41D, D/21E, D/41E, D/41ESC, D/81A, D/121B, D/160SC-LS, D/240SC, D/240SC-T1, D/300SC-E1 and D/320SC boards. Speed control is not supported on the D/121A board.

■ Example

```
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>


/*
 * Global Variables
 */


main()
{
   DX_SVMT          svmt;
   int       dxxxdev, index;


   /*
    * Open the Voice Channel Device and Enable a Handler
    */
   if ( ( dxxxdev = dx_open( "dxxxB1C1", NULL) ) == -1 ) {
      perror( "dxxxB1C1" );
      exit( 1 );
   }


   /*
    * Set up the Speed/Volume Modification
    */
   memset( &svmt, 0, sizeof( DX_SVMT ) );
   svmt.decrease[ 0 ] = -128;
   svmt.decrease[ 1 ] = -128;
   svmt.decrease[ 2 ] = -128;
   svmt.decrease[ 3 ] = -128;
   svmt.decrease[ 4 ] = -128;
   svmt.decrease[ 5 ] = -20;
   svmt.decrease[ 6 ] = -16;
   svmt.decrease[ 7 ] = -12;
   svmt.decrease[ 8 ] = -8;
   svmt.decrease[ 9 ] = -4;
   svmt.origin = 0;
   svmt.increase[ 0 ] = 4;
   svmt.increase[ 1 ] = 8;
   svmt.increase[ 2 ] = 10;
   svmt.increase[ 3 ] = -128;
   svmt.increase[ 4 ] = -128;
   svmt.increase[ 5 ] = -128;
   svmt.increase[ 6 ] = -128;
   svmt.increase[ 7 ] = -128;
   svmt.increase[ 8 ] = -128;
   svmt.increase[ 9 ] = -128;


   /*
    * Update the Volume Modification Table without Wrap Mode.
    */
   if (dx_setsvmt( dxxxdev, SV_VOLUMETBL, &svmt, 0 ) == -1){
      printf( "Unable to Set the Volume" );
      printf( " Modification Table\n");
      printf( "Lasterror = %d  Err Msg = %s\n",
         ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
      dx_close( dxxxdev );
      exit( 1 );
```

```
   }

   /*
    * Continue Processing
    *   .
    *   .
    */

   /*
    * Close the opened Voice Channel Device
    */
   if ( dx_close( dxxxdev ) != 0 ) {
      perror( "close" );
   }
   /* Terminate the Program */
   exit( 0 );
}
```

■ Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	• Invalid Parameter
EDX_BADPROD	• Function not supported on this board
EDX_NONZEROSIZE	• Reset to Default was Requested but size was non-zero
EDX_SPDVOL	• Must Specify either SV_SPEEDTBL or SV_VOLUMETBL
EDX_SVMTRANGE	• An Entry in *DX_SVMT* was out of Range
EDX_SVMTSIZE	• Invalid Table Size Specified
EDX_SYSTEM	• Windows NT system error - check **errno**

■ See Also

- **dx_adjsv( )**
- **dx_getcursv( )**
- **dx_getsvmt( )**
- "Speed and Volume Modification Tables" (*Voice Features Guide for Windows NT*)
- *DX_SVMT* (*Chapter 4. Voice Data Structures and Device Parameters*)

Name:	int dx_setuio(uioblk)	
Inputs:	uioblk	• *DX_UIO* structure
Returns:	0 if success -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	miscellaneous function	

■ Description

The **dx_setuio( )** function allows an application to install a user I/O routine **read( )**, **write( )**, and **lseek( )** functions. These functions are then used by the **dx_play( )** and **dx_rec( )** functions to read and/or write to nonstandard storage media.

The application provides the addresses of user-defined **read( )**, **write( )** and, optionally, **lseek( )** functions by initializing the *DX_UIO* structure. The application then installs the functions by invoking the **dx_setuio( )** function.

The application can override the standard I/O functions on a file-by-file basis by setting the IO_UIO flag in the **io_type** field of the *DX_IOTT* structure (see *Chapter 4.* Voice Data Structures and Device Parameters for details).

NOTE: The IO_UIO flag must be ORed with the IO_DEV flag for this feature to function properly.

When using the **dx_setuio( )** function to record, a user-defined **write( )** function must be provided. User-defined **read( )** and **lseek( )** functions are optional.

When using the **dx_setuio( )** function to play, a user-defined **read( )** function must be provided. User-defined **write( )** and **lseek( )** functions are optional.

■ Cautions

In order for the application to work properly, the user-provided functions *must* conform to standard I/O function semantics.

■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
#include "VOXLIB.H"     /* Dialogic voice library header file */
int cd;                 /* channel descriptor */
DX_UIO myio;            /* user definable I/O structure */
/*
 * User defined I/O functions
 */
int my_read9(fd,ptr,cnt)
int fd;
char * ptr;
unsigned cnt;
{
   printf("My read\n");
   return(read(fd,ptr,cnt));
}
/*
 * my write function
 */
int my_write(fd,ptr,cnt)
int fd;
char * ptr;
unsigned cnt;
{
printf("My write \n");
   return(write(fd,ptr,cnt));
}
/*
 * my seek function
 */
long my_seek(fd,offset,whence)
int fd;
long offset;
int whence;
{
   printf("My seek\n");
   return(lseek(fd,offset,whence));
}
void main(argc,argv)
int argc;
char *argv[];
{
   .
   . /* Other initialization */
   .
   DX_UIO uioblk;
/* Initialize the UIO structure */
uioblk.u_read=my_read;
uioblk.u_write=my_write;
uioblk.u_seek=my_seek;
/* Install my I/O routines */
dx_setuio(uioblk);
vodat_fd = dx_fileopen("JUNK.VOX",O_RDWR|O_BINARY);
/*This block uses standard I/O functions */
iott->io_type = IO_DEV|IO_CONT
iott->io_fhandle = vodat_fd;
iott->io_offset = 0;
iott->io_length = 20000;
/*This block uses my I/O functions */
iottp++;
```

```
iottp->io_type = IO_DEV|IO_UIO|IO_CONT
iottp->io_fhandle = vodat_fd;
iott->io_offset = 20001;
iott->io_length = 20000;
/*This block uses standard I/O functions */
iottp++
iott->io_type = IO_DEV|IO_CONT
iott->io_fhandle = vodat_fd;
iott->io_offset = 20002;
iott->io_length = 20000;
/*This block uses my I/O functions */
iott->io_type = IO_DEV|IO_UIO|IO_EOT
iott->io_fhandle = vodat_fd;
iott->io_offset = 10003;
iott->io_length = 20000;
devhandle = dx_open("dxxxB1C1", NULL);
dx_sethook(devhandle, DX-ONHOOK,EV_SYNC)
dx_wtring(devhandle,1,DX_OFFHOOK,EV_SYNC);
dx_clrdigbuf;
   if(dx_rec(devhandle,iott,(DX_TPT*)NULL,RM_TONE|EV_SYNC) == -1) {
  perror("");
  exit(1);
}
dx_clrdigbuf(devhandle);
   if(dx_play(devhandle,iott,(DX_TPT*)EV_SYNC) == -1 {
   perror("");
   exit(1);
   }
   dx_close(devhandle);
```

■ Errors

None.

Name:	int dx_stopch(chdev,mode)	
Inputs:	int chdev	• valid Dialogic channel device handle
	unsigned short mode	• Reserved for future use
Returns:	0 if success -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	I/O	
Mode:	asynchronous/synchronous	

■ Description

The **dx_stopch( )** function forces termination of currently active I/O functions on a channel. It forces a channel in the busy state to become idle. If the channel specified in **chdev** already is idle, **dx_stopch( )** has no effect and will return a success.

Running this function asynchronously will initiate the **dx_stopch( )** without affecting processes on other channels.

Running this function synchronously within a process does not block other processing. Other processes continue to be serviced.

When an I/O function terminates due to a **dx_stopch( )** being issued on the channel, the termination reason returned by **ATDX_TERMMSK( )** will be TM_USRSTOP. (If running the **dx_dial( )** function with Call Analysis, you can call **ATDX_CPTERM( )** to determine the reason for Call Analysis termination. This will return CR_STOPD if Call Analysis stopped due to a **dx_stopch( )**.)

The function parameters are defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.
mode:	Set to EV_ASYNC. The stop will be issued, but the driver does not *sleep* and wait for the channel to become idle before **dx_stopch( )** returns.

Cautions

1. **dx_stopch( )** will have no effect on a channel that has either of the following functions issued:

   - **dx_dial( )** without Call Analysis enabled
   - **dx_wink( )**

   The functions will continue to run normally, **dx_stopch( )** will return a success. For **dx_dial( )**, the digits specified in the **dialstrp** parameter will still be dialed.

2. If **dx_stopch( )** is called on a channel dialing with Call Analysis enabled, the Call Analysis process will stop but dialing will be completed. Any Call Analysis information collected prior to the stop will be returned by Extended Attribute functions.

3. If an I/O function terminates (due to another reason) before **dx_stopch( )** is issued, the reason for termination will not indicate **dx_stopch( )** was called.

4. When calling **dx_stopch( )** from a signal handler, **mode** must be set to EV_ASYNC.

Example

```
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   int chdev, srlmode;

   /* Set SRL to run in polled mode. */
   srlmode = SR_POLLMODE;
   if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1) {
      /* process error */
   }

   /* Open the channel using dx_open( ). Get channel device descriptor in
    * chdev.
    */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
      /* process error */
   }

   /* continue processing */
               .
               .
```

```
   /* Force the channel idle.  The I/O function that the channel is
    * executing will be terminated, and control passed to the handler
    * function previously enabled, using sr_enbhdlr(), for the
    * termination event corresponding to that I/O function.
    * In the asynchronous mode, dx_stopch() returns immediately,
    * without waiting for the channel to go idle.
    */
   if ( dx_stopch(chdev, EV_ASYNC) == -1) {
      /* process error */
   }

}
```

■ Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	• Invalid Parameter
EDX_SYSTEM	• Windows NT system error - check **errno**

■ See Also

Related I/O functions:

- **dx_dial( )**
- **dx_getdig( )**
- **dx_play( )**
- **dx_playf( )**
- **dx_playtone( )**
- **dx_rec( )**
- **dx_recf( )**
- **dx_wink(**)

Retrieving I/O termination reason due to dx_stopch( ):

- **ATDX_TERMMSK( )**
- **ATDX_CPTERM( )** - **dx_dial( )** with Call Analysis

Name:	int dx_wink(chdev,mode)	
Inputs:	int chdev	• valid Dialogic channel device handle
	unsigned short mode	• synchronous/asynchronous setting
Returns:	0 if successful -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	I/O	
Mode:	synchronous/asynchronous	

■ Description

The **dx_wink( )** function generates an outbound wink on the specified channel. A wink from a Voice board is a momentary rise of the A signaling bit, which corresponds to a wink on an E&M line. This is used for signaling T-1 spans. A wink's typical duration of 150 to 250 milliseconds used for communication purposes between the called and calling stations.

NOTE: Do not call this function on a non-E&M line or for a SCbus T-1 digital interface device on a D/240SC or a D/240SC-T1 board. Transparent signaling for SCbus digital interface devices is not supported in System Release 4.1SC. See the *Digital Network Interface Software Reference for Windows NT* for information about E&M lines.

■ Asynchronous Operation

To run this function asynchronously set the **mode** field to EV_ASYNC. When running asynchronously, this function will return 0 to indicate it has initiated successfully, and will generate a TDX_WINK termination event to indicate completion. Use the SRL Event Management functions to handle the termination event. See *Appendix* A for more information about the Event Management functions.

■ Synchronous Operation

By default, this function runs synchronously, and will return a 0 to indicate that it has completed successfully.

The function parameters are defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.
mode:	specifies whether to run **dx_wink( )** asynchronously or synchronously. Specify one of the following: EV_ASYNC: Run **dx_wink( )** asynchronously. EV_SYNC: Run **dx_wink( )** synchronously (default).

NOTES:

1. The **dx_wink( )** function is supported on a T-1 E&M line connected to a DTI/101 board. In addition, the **dx_wink( )** function is supported on the DTI/211 board in transparent mode.
2. The channel must be on-hook when **dx_wink( )** is called.
3. All values referenced for this function are subject to a 10 ms clocking resolution. Actual values will be in a range: (parameter value - 9 ms) $\leq$ actual value $\leq$ (parameter value)

■ Setting Delay Prior to Wink

The default delay prior to generating the outbound wink is 150 ms. To change the delay, use the **dx_setparm( )** function to enter a value for the DXCH_WINKDLY parameter where:

delay = the value entered x 10 ms

The syntax of the function is:

```
int delay;
delay = 15;
dx_setparm(dev,DXCH_WINKDLY,(void*)&delay)
```

If delay = 15, then DXCH_WINKDLY = 15 x 10 or 150 ms.

■ Setting Wink Duration

The default outbound wink duration is 150 ms. To change the wink duration, use the **dx_setparm( )** function to enter a value for the DXCH_WINKLEN parameter where:

duration = the value entered x 10 ms

The syntax of the function is:

```
int duration;
duration = 15;
dx_setparm(dev,DXCH_WINKLEN,(void*)&duration)
```

If duration = 15, then DXCH_WINKLEN = 15 x 10 or 150 ms.

■ Receiving an Inbound Wink

NOTE: The inbound wink duration must be between the values set for DXCH_MINRWINK and DXCH_MAXRWINK. The default value for DXCH_MINRWINK is 100 ms, and the default value for DXCH_MAXRWINK is 200 ms. Use the **dx_setparm( )** function to change the minimum and maximum allowable inbound wink duration.

To receive an inbound wink on a channel:

1. Using the **dx_setparm( )** function, set the off-hook delay interval (DXBD_OFFHDLY) parameter to 1 so that the channel is ready to detect an incoming wink immediately upon going off hook.
2. Using the **dx_setevtmsk( )** function, enable the DM_WINK event.

NOTE: If DM_WINK is not specified in the mask parameter of the **dx_setevtmsk( )** function, and DM_RINGS is specified, a wink will be interpreted as an incoming call.

A typical sequence of events for an inbound wink is:

1. The application calls the **dx_sethook( )** function to initiate a call by going off hook.
2. When the incoming call is detected by the Central Office, the CO responds by sending a wink to the board.

3. When the wink is received successfully, a DE_WINK event is sent to the application.

■ Cautions

Make sure the channel is on-hook when **dx_wink( )** is called.

■ Example 1: Using dx_wink( ) in synchronous mode.

```
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   int chdev;
   DV_TPT tpt;
   DV_DIGIT digitp;
   char buffer[8];

   /* open a channel with chdev as descriptor */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
      /* process error */
   }

   /* set hookstate to on-hook and wink */
   if (dx_sethook(chdev,DX_ONHOOK,EV_SYNC) == -1) {
      /* process error */
   }

   if (dx_wink(chdev,EV_SYNC) == -1) {
      /* error winking channel */
   }

   dx_clrtpt(&tpt,1);

   /* set up DV_TPT */
   tpt.tp_type   = IO_EOT;         /* only entry in the table */
   tpt.tp_termno = DX_MAXDTMF;     /* Maximum digits */
   tpt.tp_length = 1;              /* terminate on the first digit */
   tpt.tp_flags  = TF_MAXDTMF;     /* Use the default flags */

   /* get digits while on-hook */

   if (dx_getdig(chdev,&tpt, &digitp, EV_SYNC) == -1) {
      /* error getting digits */
   }

   /* now we can go off-hook and continue */

   if ( dx_sethook(chdev,DX_OFFHOOK,EV_SYNC)== -1) {
      /* process error */
```

```
   }
   .
   .
}
```

■ Example 2: Using dx_wink( ) in asynchronous mode.

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

#define MAXCHAN 24

int wink_handler();

main()
{
   int i, chdev[MAXCHAN];
   char *chnamep;
   int srlmode;

   /* Set SRL to run in polled mode. */
   srlmode = SR_POLLMODE;
   if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1) {
      /* process error */
   }

   for (i=0; i<MAXCHAN; i++) {

      /* Set chnamep to the channel name - e.g., dxxxB1C1 */

      /* open the channel with dx_open( ). Obtain channel device
       * descriptor in chdev[i]
       */
       if ((chdev[i] = dx_open(chnamep,NULL)) == -1) {
           /* process error */
       }

      /* Using sr_enbhdlr(), set up handler function to handle wink
       * completion events on this channel.
       */
       if (sr_enbhdlr(chdev[i], TDX_WINK, wink_handler) == -1) {
           /* process error */
       }

      /* Before issuing dx_wink(), ensure that the channel is onhook,
       * else the wink will fail.
       */
       if(dx_sethook(chdev[i], DX_ONHOOK, EV_ASYNC)==-1){
          /* error setting channel on-hook */
       }
       /* Use sr_waitevt( ) to wait for the completion of dx_sethook( ). */
       if (dx_wink(chdev[i], EV_ASYNC) == -1) {
          /* error winking channel */
       }
      }
```

```
   /* Use sr_waitevt() to wait for the completion of wink.
    * On receiving the completion event, TDX_WINK, control is transferred
    * to the handler function previously established using sr_enbhdlr().
    */
         .
         .
}

int wink_handler()
{
   printf("wink completed on channel %s\n", ATDX_NAMEP(sr_getevtdev()));
   return 0;
}
```

■ Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	• Invalid Parameter
EDX_SYSTEM	• Windows NT system error - check **errno**

■ See Also

Related Functions:

- **dx_setparm( )**
- **dx_getparm( )**

Handling and Retrieving dx_wink Termination Events:

- Event Management functions (*Standard Runtime Library Programmer's Guide for Windows NT* and *Appendix* A)
- DV_TPT
- **ATDX_TERMMSK( )**

Handling outbound winks:

- **dx_wtring( )**

Handling inbound winks:

- **dx_setevtmsk( )**
- **dx_sethook( )**
- *DX_CST* (*Chapter 4. Voice Data Structures and Device Parameters*)

- **dx_getevt( )** - synchronous
- EV_EBLK - synchronous applications (*Chapter 4. Voice Data Structures and Device Parameters*)
- **sr_getevtdatap( )** - asynchronous (*Standard Runtime Library Programmer's Guide for Windows NT* and *Appendix* A)

Name:	int dx_wtring(chdev,nrings,hstate,timeout)	
Inputs:	int chdev	• valid Dialogic channel device handle
	int nrings	• number of rings to wait for
	int hstate	• hook state to set after rings are detected
	int timeout	• in seconds
Returns:	0 if successful -1 if failure	
Includes:	srllib.h dxxxlib.h	
Category:	Configuration	

■ Description

The **dx_wtring( )** function waits for a specified number of rings and sets the channel to on-hook or off-hook after the rings are detected. Using **dx_wtring( )** is equivalent to using **dx_setevtmsk( )**, **dx_getevt( )**, and **dx_sethook( )** to wait for a ring. When **dx_wtring( )** is called, the specified channel's event is set to DM_RINGS.

NOTE: Do not call this function for a digital T-1 SCbus configuration that includes a D/240SC, D/240SC-T1, or DTI/241SC board. Transparent signaling for SCbus digital interface devices is not supported in System Release 4.1SC.

The function parameters are defined as follows:

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.
nrings:	specifies the number of rings to wait for before setting the hook state.
hstate:	sets the hookstate of the channel after the number of rings specified in nrings are detected. **hstate** can have either of the following values: DX_ONHOOK: channel remains on-hook when **nrings** number of rings are detected

Parameter	Description	
	DX_OFFHOOK:	channel goes off-hook when **nrings** number of rings are detected
timeout:	specifies the maximum length of time in tenths of seconds to wait for a ring. timeout can have one of the following values:	
	# of seconds:	maximum length of time to wait for a ring.
	-1:	**dx_wtring( )** waits forever and never times out.
	0:	**dx_wtring( )** returns -1 immediately if a ring event does not already exist.

■ Cautions

1. **dx_wtring( )** changes the event enabled on the channel to DM_RINGS. For example, "process A" issues **dx_setevtmsk( )** to enable detection of another type of event, e.g., DM_SILON, on channel one. If "process B" issues **dx_wtring( )** on channel one, then process A will now be waiting for a DM_RINGS event since process B has reset the channel event to DM_RINGS with **dx_wtring( )**.
2. A channel can detect rings immediately after going on hook. Rings may be detected during the time interval between **dx_sethook( )** and **dx_wtring( )**. Rings are counted as soon as they are detected.

   NOTE: If the number of rings detected before **dx_wtring( )** returns is equal to or greater than nrings, **dx_wtring( )** will not terminate. This may cause the application to miss calls that are already coming in when the application is first started.
3. Do not use the Windows NT **sigset( )** system call with SIGALRM while waiting for rings.

■ Example

```
#include <srllib.h>
#include <dxxxlib.h>
```

```
#include <windows.h>

main()
{
   int chdev;      /* channel descriptor */
   .
   .
   /* Open Channel */
   if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
      /* process error */
   }

   /* Wait for two rings on this channel - no timeout */
   if (dx_wtring(chdev,2,DX_OFFHOOK,-1) == -1) {
      /* process error */
   }
   .
   .
}
```

■ Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	• Invalid Parameter
EDX_SYSTEM	• Windows NT system error - check **errno**
EDX_TIMEOUT	• Timeout limit is reached

■ See Also

- **dx_setevtmsk( )**
- **dx_getevt( )**
- **dx_sethook( )**
- *DX_EBLK* (*Chapter 4. Voice Data Structures and Device Parameters*)

Name: int r2_creatfsig(chdev,forwardsig)
Inputs: int chdev • channel device handle
int forwardsig • group I/II forward signal
Returns: 0 if success
-1 if failure
Includes: srllib.h
dxxxlib.h
Category: R2MF Convenience

■ Description

r2_creatfsig( ) is a convenience function that defines and enables leading edge detection of an R2MF forward signal on a channel.

NOTE: This function calls the **dx_blddt( )** function to create the template.

User-defined tone IDs 101 through 115 are used by this function.

For detailed information about R2MF signaling see the *Voice Features Guide for Windows NT*.

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.
forwardsig:	specifies the name of a Group I or Group II forward signal which provides the tone ID for detection of the associated R2MF tone (or tones). Set to R2_ALLSIG to enable detection of all 15 tones **or** set to one of the following defines:

Specify one of:		**Associated**
Group I	**Group II**	**Tone ID**
SIGI_1	SIGII_1	101
SIGI_2	SIGII_2	102
SIGI_3	SIGII_3	103
SIGI_4	SIGII_4	104
SIGI_5	SIGII_5	105
SIGI_6	SIGII_6	106

Parameter	Description		
	SIGI_7	SIGII_7	107
	SIGI_8	SIGII_8	108
	SIGI_9	SIGII_9	109
	SIGI_10	SIGII_10	110
	SIGI_11	SIGII_11	111
	SIGI_12	SIGII_12	112
	SIGI_13	SIGII_13	113
	SIGI_14	SIGII_14	114
	SIGI_15	SIGII_15	115

NOTE: Either the Group I or the Group II define can be used to specify the forward signal, because the Group I and Group II defines correspond to the same set of 15 forward signals, and the same user-defined tones are used for Group I and Group II.

■ Cautions

1. The channel must be idle when calling this function.
2. Prior to creating the R2MF tones on a channel, you should delete any previously created user-defined tones (including non-R2MF tones) to avoid getting an error for having too many tones enabled on a channel.
3. This function creates R2MF tones with user-defined tone IDs from 101 to 115, and you should reserve these tone IDs for R2MF. If you attempt to create a forward signal tone with this function and you previously created a tone with the same tone ID, an invalid tone ID error will occur.
4. Maximum number of user-defined tones is on a per board basis.

■ Example

```
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   int  dxxxdev;
```

```
   /*
    * Open the Voice Channel Device and Enable a Handler
    */
   if ( ( dxxxdev = dx_open( "dxxxB1C1", NULL) ) == -1 ) {
      perror( "dxxxB1C1" );
      exit( 1 );
   }

   /*
    * Create all forward signals
    */
   if ( r2_creatfsig( dxxxdev, R2_ALLFSIG ) == -1 ) {
      printf( "Unable to Create the Forward Signals\n" );
      printf( "Lasterror = %d  Err Msg = %s\n",
          ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
      dx_close( dxxxdev );
      exit( 1 );
   }

   /*
    * Continue Processing
    *   .
    *   .
    *   .
    */

   /*
    * Close the opened Voice Channel Device
    */
   if ( dx_close( dxxxdev ) != 0 ) {
      perror( "close" );
   }

   /* Terminate the Program */
   exit( 0 );
}
```

■ Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	• Invalid parameter
EDX_BADPROD	• Function not supported on this board
EDX_SYSTEM	• Windows NT System error - check **errno**
EDX_TONEID	• Invalid tone template ID
EDX_MAXTMPLT	• Maximum number of user-defined tones for the board
EDX_INVSUBCMD	• Invalid sub-command
EDX_FREQDET	• Invalid tone frequency
EDX_CADENCE	• Invalid cadence component value

EDX_ASCII	• Invalid ASCII value in tone template description
EDX_DIGTYPE	• Invalid Dig_Type value in tone template description

■ See Also

- **r2_playbsig( )**
- **dx_addtone( )**
- **dx_blddt( )**
- "R2MF Signaling" - (*Voice Features Guide for Windows NT*)

Name:	int r2_playbsig(chdev,backwardsig,forwardsig,mode)	
Inputs:	int chdev	• channel device handle
	int backwardsig	• group A/B backward signal
	int forwardsig	• group I/II forward signal
	int mode	• asynchronous/synchronous
Returns:	0 if success	
	error return code	
Includes:	srllib.h	
	dxxxlib.h	
Category:	R2MF	
Mode:	asynchronous/synchronous	

■ Description

The **r2_playbsig( )** function plays a specified backward R2MF signal on the specified channel until a tone-off event is detected for the specified forward signal.

The **r2_playbsig( )** function is a convenience function that plays a tone and controls the timing sequence required by the R2MF compelled signaling procedure.

Compelled signaling sends each signal, until it is responded to by a return signal, which in turn is sent until responded to by the other party. See the *Voice Features Guide for Windows NT* for more information about R2MF Compelled signaling.

NOTE: This function calls the **dx_playtone( )** function to play the tone.

■ Asynchronous Operation

1. Enable forward signal detection using **r2_creatfsig( )**.
2. Use SRL to asynchronously wait for TDX_CST event(s).
3. Use **sr_getevtdatap( )** to retrieve the *DX_CST* structure, which will contain a DE_TONEON event in the **cst_event** field.
4. Determine which forward signal was detected by matching the tone ID returned **cst_data** field (from 101 to 115) with the forward signal Group I or Group II defines (see **forwardsig** argument description for a list of the forward signal defines).

5. Decide which backward signal should be played in response to the forward signal.
6. Use the **r2_playbsig( )** function to play the desired backward signal.
7. **r2_playbsig( )** will terminate automatically when a tone-off event is detected. There is a 60-second default duration for playing the backward signal. If the forward signal tone-off is not detected within 60 seconds, the backward signal will terminate with a TDX_PLAYTONE event, and ATDX_TERMMSK will return TM_MAXTIME.

■ Synchronous Operation

8. Enable forward signal detection using **r2_creatfsig( )**.
9. Call **dx_getevt( )** to wait for a DX_TONEON event. Events are returned in the *DX_EBLK* structure.
10. Determine which forward signal was detected by matching the tone ID contained in the **ev_data** field (from 101 to 115) with the forward signal Group I or Group II defines (see **forwardsig** argument description for a list of the forward signal defines).
11. Decide which backward signal should be played in response to the forward signal.
12. Use the **r2_playbsig( )** function to play the desired backward signal.
13. **r2_playbsig( )** will terminate automatically when a tone-off event is detected. There is a 60-second default duration for playing the backward signal. If the forward signal tone-off is not detected within 60 seconds, the backward signal will terminate, and **ATDX_TERMMSK( )** will return TM_MAXTIME.

Parameter	Description
chdev:	specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**.
backwardsig:	specifies the name of a Group A or Group B backward signal to play. Set to one of the defines in Group A or one of the defines in Group B:

Parameter **Description**

Specify one of:		**Associated**
Group A	**Group B**	**Tone ID**
SIGA_1	SIGB_1	101
SIGA_2	SIGB_2	102
SIGA_3	SIGB_3	103
SIGA_4	SIGB_4	104
SIGA_5	SIGB_5	105
SIGA_6	SIGB_6	106
SIGA_7	SIGB_7	107
SIGA_8	SIGB_8	108
SIGA_9	SIGB_9	109
SIGA_10	SIGB_10	110
SIGA_11	SIGB_11	111
SIGA_12	SIGB_12	112
SIGA_13	SIGB_13	113
SIGA_14	SIGB_14	114
SIGA_15	SIGB_15	115

forwardsig: specifies the name of the Group I or Group II forward signal for which a tone-on event was detected, and for which a tone-off event will terminate this function. Set to one of defines from Group I or one of the defines from Group II:

Specify one of:		**Associated**
Group I	**Group II**	**Tone ID**
SIGI_1	SIGII_1	101
SIGI_2	SIGII_2	102
SIGI_3	SIGII_3	103
SIGI_4	SIGII_4	104
SIGI_5	SIGII_5	105
SIGI_6	SIGII_6	106
SIGI_7	SIGII_7	107
SIGI_8	SIGII_8	108
SIGI_9	SIGII_9	109
SIGI_10	SIGII_10	110
SIGI_11	SIGII_11	111
SIGI_12	SIGII_12	112
SIGI_13	SIGII_13	113
SIGI_14	SIGII_14	114

Parameter	Description		
	SIGI_15	SIGII_15	115

The following procedure describes how to use the **r2_playbsig( )** function:

■ Example

```
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
   int  dxxxdev;

   /*
    * Open the Voice Channel Device and Enable a Handler
    */
   if ( ( dxxxdev = dx_open( "dxxxB1C1", NULL) ) == -1 ) {
      perror( "dxxxB1C1" );
      exit( 1 );
   }

   /*
    * Create all forward signals
    */
   if ( r2_creatfsig( dxxxdev, R2_ALLFSIG ) == -1 ) {
      printf( "Unable to Create the Forward Signals\n" );
      printf( "Lasterror = %d  Err Msg = %s\n",
          ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
      dx_close( dxxxdev );
      exit( 1 );
   }

   /*
    * Continue Processing
    *   .
    *   .
    *   .
    *
    * Detect an incoming call using dx_wtring()
    *
    * Enable the detection of all forward signals using
    * dx_enbtone(). In this example, only the first
    * forward signal will be enabled.
    */
   if (dx_enbtone( dxxxdev, SIGI_1, DM_TONEON | DM_TONEOFF ) == -1 ) {
      printf( "Unable to Enable Detection of Tone %d\n", SIGI_1 );
      printf( "Lasterror = %d  Err Msg = %s\n",
          ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
      dx_close( dxxxdev );
      exit( 1 );
```

```
   }

   /*
    * Now wait for the TDX_CST event and event type,
    * DE_TONEON. The data part contains the ToneId of
    * the forward signal detected. Based on the forward
    * signal, determine the backward signal to generate.
    *
    * In this example, we will be generating the Group A
    * backward signal A-1 (send next digit) assuming
    * forward signal received is SIGI_1.
    */

   if ( r2_playbsig( dxxxdev, SIGA_1, SIGI_1, EV_SYNC ) == -1 ) {
      printf( "Unable to generate the backward signals\n" );
      printf( "Lasterror = %d  Err Msg = %s\n",
          ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
      dx_close( dxxxdev );
      exit( 1 );
   }

   /*
    * Continue Processing
    *   .
    *   .
    *   .
    */

   /*
    * Close the opened Voice Channel Device
    */
   if ( dx_close( dxxxdev ) != 0 ) {
      perror( "close" );
   }

   /* Terminate the Program */
   exit( 0 );
}
```

■ Errors

If this function returns -1 to indicate failure, use **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to retrieve one of the following error reasons:

EDX_BADPARM	• Invalid parameter
EDX_BADPROD	• Function not supported on this board
EDX_BADTPT	• Invalid DV_TPT entry
EDX_BUSY	• Busy executing I/O function
EDX_AMPLGEN	• Invalid amplitude value in *TN_GEN* structure
EDX_FREQGEN	• Invalid frequency component in TN_GEN structure
EDX_FLAGGEN	• Invalid tn_dflag field in TN_GEN structure
EDX_SYSTEM	• Windows NT system error - check **errno**

■ Cautions

The channel must be idle when calling this function.

■ See Also

- **r2_creatfsig( )**
- **dx_blddt( )**
- **dx_playtone( )**
- "R2MF Signaling" - (*Voice Features Guide for Windows NT*)

4. Voice Data Structures and Device Parameters

This chapter provides a description of the voice library data structures and voice board parameters. The following topics are included:

- Data Structures and Tables from the Voice Library (*Section 4.1*)
- Parameters used by **dx_setparm( )** and **dx_getparm( )** (*Section 4.2*)

4.1. Voice Library Data Structures

The following sections describe each of the data structures used by the Voice Library functions.

The voice software includes structures that indicate I/O termination, contain a device's status information, or provide other function-specific information. The following structures are used:

DV_DIGIT	• User Digit Buffer Structure
DX_CST	• Call Status Transition Structure
DX_CAP	• Call Analysis Parameter Structure
DX_EBLK	• Event Block Structure
DX_IOTT	• I/O Transfer Table Structure
DX_SVCB	• Speed/Volume Adjustment Condition Block
DX_SVMT	• Speed/Volume Modification Block
DX_TPB	• Test Parameter Block Structure
DX_UIO	• User-definable I/O Structure
DX_XPB	• I/O Transfer Parameter Block
TN_GEN	• Tone Generation Template structure

NOTE: DV_TPT termination parameter structure, which is defined in the Standard Runtime Library is described in *Appendix* A.

4.1.1. DV_DIGIT - *user digit buffer*

When a **dx_getdig( )** 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.

The typedef for the structure is shown below:

```
typedef struct DV_DIGIT {
    char dg_value[DG_MAXDIGS +1];  /* ASCII values of digits */
    char dg_type[DG_MAXDIGS +1];   /* Type of digits */
} DV_DIGIT;
```

where:

dg_value is a NULL-terminated string of the ASCII values of the digits collected.

dg_type is an array (terminated by DG_END) of the digit types corresponding to each of the digits contained in dg_value. The defines for the digit types are:

DG_DTMF	• DTMF digit
DG_LPD	• Loop Pulse digit
DG_MF	• MF digit
DG_USER1	• User defined tone
DG_USER2	• User defined tone
DG_USER3	• User defined tone
DG_USER4	• User defined tone
DG_USER5	• User defined tone

DG_MAXDIGS is the define for the maximum number of digits (31) that can be returned by a single call to **dx_getdig( )**. Refer to *dxxxlib.h* for the value of DG_MAXDIGS.

4.1.2. DX_CAP - *change default call analysis parameters*

The DX_CAP structure modifies parameters that control Frequency Detection, Cadence Detection, Loop Current, and Positive Voice Detection. DX_CAP structure is used for input to the **setcparm( )** function to modify call analysis channel parameters when using **dx_dial( )**.

This structure provides the ability to change the default Call Analysis parameters when using **dx_dial( )**. This structure may be used only under the following circumstances:

- When dialing on D/41ESC, D/xxxSC (D/160SC-LS, D/240SC, D/240SC-T1, D/300SC-E1, D/320SC, DTI/241SC, DTI/301SC, LSI/81SC, or LSI/161SC) channels
- When clearing DX_CAP fields using **dx_clrcap( )**

NOTE: For more detail about Call Analysis and how and when to use the DX_CAP structure see the *Voice Features Guide for Windows NT*.

To clear the fields in a DX_CAP structure, use the **dx_clrcap( )** function.

If you set any DX_CAP field to 0, the field will be reset to the default value for the field. The setting used by a previously called **dx_dial( )** function is ignored.

The typedef for the structure is shown on the following pages.

```
 * DX_CAP
 *
 * Call Analysis parameters
 * [NOTE: All user-accessible structures must be defined so as to be
 *        unaffected by structure packing.]
 */
typedef struct DX_CAP {
   unsigned short ca_nbrdna;         /* # of rings before no answer. */
   unsigned short ca_stdely;         /* Delay after dialing before
                                        analysis. */
   unsigned short ca_cnosig;         /* Duration of no signal time out
                                        delay. */
   unsigned short ca_lcdly;          /* Delay after dial before lc drop
                                        connect */
   unsigned short ca_lcdly1;         /* Delay after lc drop con. before
                                        msg. */
   unsigned short ca_hedge;          /* Edge of answer to send connect
                                        message. */
   unsigned short ca_cnosil;         /* Initial continuous noise timeout
                                        delay. */
   unsigned short ca_lo1tola;        /* % acceptable pos. dev of short low
                                        sig. */
   unsigned short ca_lo1tolb;        /* % acceptable neg. dev of short low
                                        sig. */
   unsigned short ca_lo2tola;        /* % acceptable pos. dev of long low
                                        sig. */
   unsigned short ca_lo2tolb;        /* % acceptable neg. dev of long low
                                        sig. */
   unsigned short ca_hi1tola;        /* % acceptable pos. dev of high
                                        signal. */
   unsigned short ca_hi1tolb;        /* % acceptable neg. dev of high
                                        signal. */
```

```
unsigned short ca_lo1bmax;        /* Maximum interval for shrt low for
                                     busy. */
unsigned short ca_lo2bmax;        /* Maximum interval for long low for
                                     busy. */
unsigned short ca_hi1bmax;        /* Maximum interval for 1st high for
                                     busy */
unsigned short ca_nsbusy;         /* Num. of highs after nbrdna busy
                                     check. */
unsigned short ca_logltch;        /* Silence deglitch duration. */
unsigned short ca_higltch;        /* Non-silence deglitch duration. */
unsigned short ca_lo1rmax;        /* Max. short low  dur. of double
                                     ring. */
unsigned short ca_lo2rmin;        /* Min. long low  dur. of double
                                     ring. */
unsigned short ca_intflg;         /* Operator intercept mode. */
unsigned short ca_intfltr;        /* Minimum signal to qualify freq.
                                     detect. */
unsigned short rfu1;              /* reserved for future use */
unsigned short rfu2;              /* reserved for future use */
unsigned short rfu3;              /* reserved for future use */
unsigned short rfu4;              /* reserved for future use */
unsigned short ca_hisiz;          /* Used to determine which lowmax to
                                     use. */
unsigned short ca_alowmax;        /* Max. low before con. if high
                                     >hisize. */
unsigned short ca_blowmax;        /* Max. low before con. if high
                                     <hisize. */
unsigned short ca_nbrbeg;         /* Number of rings before analysis
                                     begins. */
unsigned short ca_hi1ceil;        /* Maximum 2nd high dur. for a
                                     retrain. */
unsigned short ca_lo1ceil;        /* Maximum 1st low dur. for a
                                     retrain. */
unsigned short ca_lowerfrq;       /* Lower allowable frequency in hz. */
unsigned short ca_upperfrq;       /* Upper allowable frequency in hz. */
unsigned short ca_timefrq;        /* Total duration of good signal
                                     required. */
unsigned short ca_rejctfrq;       /* Allowable % of bad signal. */
unsigned short ca_maxansr;        /* Maximum duration of answer. */
unsigned short ca_ansrdgl;        /* Silence deglitching value for
                                     answer. */
unsigned short ca_mxtimefrq;      /* max time for 1st freq to remain in
                                     bounds */
unsigned short ca_lower2frq;      /* lower bound for second frequency */
unsigned short ca_upper2frq;      /* upper bound for second frequency */
unsigned short ca_time2frq;       /* min time for 2nd freq to remains in
                                     bounds */
 unsigned short ca_mxtime2frq;    /* max time for 2nd freq to remain in
                                     bounds */
 unsigned short ca_lower3frq;     /* lower bound for third frequency */
 unsigned short ca_upper3frq;     /* upper bound for third frequency */
 unsigned short ca_time3frq;      /* min time for 3rd freq to remains in
                                     bounds */
 unsigned short ca_mxtime3frq;    /* max time for 3rd freq to remain in
                                     bounds */
 unsigned short ca_dtn_pres;      /* Length of a valid dial tone
                                     (def=1sec) */
 unsigned short ca_dtn_npres;     /* Max time to wait for dial tone
                                     (def=3sec)*/
 unsigned short ca_dtn_deboff;    /* The dialtone off debouncer
                                     (def=100ms) */
```

```
    unsigned short ca_pamd_failtime; /* Wait for AMD/PVD after cadence
                                        break(default=4sec)*/
    unsigned short ca_pamd_minring;  /* min allowable ring duration
                                        (def=1.9sec)*/
    byte ca_pamd_spdval;             /* Set to 2 selects quick decision
                                        (def=1) */
    byte ca_pamd_qtemp;              /* The Qualification template to use
                                        for PAMD */
    unsigned short ca_noanswer;      /* time before no answer after first
                                        ring (default=30sec) */
    unsigned short ca_maxintering;   /* Max inter ring delay before connect
                                        (8 sec) */
} DX_CAP;
```

DX_CAP P*arameter Descriptions*

ca_nbrdna

Number of Rings Before Detecting No Answer: The number of single or double rings to wait before returning a *no answer*. (CA: Basic only)

Length: 1. Default: 4. Units: rings.

ca_stdely

Start Delay: The delay after dialing has been completed and before starting analysis for Cadence Detection, Frequency Detection, and Positive Voice Detection. (CA)

Length: 2. Default: 25. Units: 10 ms.

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. (CA)

Length: 2. Default: 4000. Units: 10 ms.

ca_lcdly

Loop Current Delay: The delay after dialing has been completed and before beginning Loop Current Detection. (CA)

-1: Disable Loop Current Detection.

Length: 2. Default: 400. Units: 10 ms.

ca_lcdly1

Loop Current Delay 1: The delay after Loop Current Detection detects a transient drop in loop current and before Call Analysis returns a *connect* to the application. (CA)

	Length: 2. Default: 10. Units: 10 ms.
ca_hedge	Hello Edge: The point at which a *connect* will be returned to the application. (CA)
	1: Rising Edge (immediately when a connect is detected).
	2: Falling Edge (after the end of the salutation).
	Length: 1. Default: 2. Units: edge.
ca_cnosil	Continuous Nonsilence: The maximum length of the first or second period of nonsilence allowed. If exceeded, a *no ringback* is returned. (CA)
	Length: 2. Default: 650. Units: 10 ms.
ca_lo1tola	Low 1 Tolerance Above: Percent acceptable positive deviation of short low signal. (CA: Basic only)
	Length: 1. Default: 13. Units: %.
ca_lo1tolb	Low 1 Tolerance Below: Percent acceptable negative deviation of short low signal. (CA: Basic only)
	Length: 1. Default: 13. Units: %.
ca_lo2tola	Low 2 Tolerance Above: Percent acceptable positive deviation of long low signal. (CA: Basic only)
	Length: 1. Default: 13. Units: %.
ca_lo2tolb	Low 2 Tolerance Below: Percent acceptable negative deviation of long low signal. (CA: Basic only)
	Length: 1. Default: 13. Units: %.
ca_hi1tola	High 1 Tolerance Above: Percent acceptable positive deviation of high signal. (CA: Basic only)
	Length: 1. Default: 13. Units: %.
ca_hi1tolb	High 1 Tolerance Below: Percent acceptable negative deviation of high signal. (CA: Basic only)
	Length: 1. Default: 13. Units: %.

ca_lo1bmax	Low 1 Busy Maximum: Maximum interval for short low for busy. (CA: Basic only) Length: 2. Default: 90. Units: 10 ms.
ca_lo2bmax	Low 2 Busy Maximum: Maximum interval for long low for busy. (CA: Basic only) Length: 2. Default: 90. Units: 10 ms.
ca_hi1bmax	High 1 Busy Maximum: Maximum interval for first high for busy. (CA: Basic only) Length: 2. Default: 90. Units: 10 ms.
ca_nsbusy	Nonsilence Busy: The number of nonsilence periods in addition to nbrdna to wait before returning a *busy*. (CA: Basic only) Length: 1. Default: 0. Negative values are valid.
ca_logltch	Low Glitch: The maximum silence period to ignore. Used to help eliminate spurious silence intervals. (CA) Length: 2. Default: 15. Units: 10 ms.
ca_higltch	High Glitch: The maximum nonsilence period to ignore. Used to help eliminate spurious nonsilence intervals. (CA) Length: 2. Default: 19. Units: 10 ms.
ca_lo1rmax	Low 1 Ring Maximum: Maximum short low duration of double ring. (CA: Basic only) Length: 2. Default: 90. Units: 10 ms.
ca_lo2rmin	Low 2 Ring Minimum: Minimum long low duration of double ring. (CA: Basic only) Length: 2. Default: 225. Units: 10 ms.
ca_intflg	Intercept Mode Flag: This parameter enables or disables SIT Frequency Detection, Positive Voice Detection (PVD), and/or Positive Answering Machine Detection (PAMD), and selects the mode of operation for Frequency

	Detection. (CA)
	DX_OPTEN: Enable Frequency Detection and wait for detection of a connect using Cadence Detection or Loop Current Detection before returning an *intercept*.
	DX_OPTDIS: Disable Frequency Detection and PVD.
	DX_OPTNOCON: Enable Frequency Detection return an *intercept* immediately after detecting a valid frequency.
	DX_PVDENABLE: Enable PVD.
	DX_PVDOPTEN: Enable PVD and DX_OPTEN.
	DX_PVDOPTNOCON: Enable PVD and DX_OPTNOCON.
	DX_PAMDENABLE: Enable PAMD.
	DX_PAMDOPTEN: Enable PAMD and DX_OPTEN.
	Length: 1. Default: DX_OPTEN.
ca_intfltr	Not used.
ca_hisiz	High Size: Used to determine whether to use alowmax or blowmax. (CA: Basic only)
	Length: 2. Default: 90. Units: 10 ms.
ca_alowmax	A Low Maximum: Maximum low before connect if high > hisiz. (CA: Basic only)
	Length: 2. Default: 700. Units: 10 ms.
ca_blowmax	B Low Maximum: Maximum low before connect if high < hisiz. (CA: Basic only)
	Length: 2. Default: 530. Units: 10 ms.
ca_nbrbeg	Number Before Beginning: Number of nonsilence periods before analysis begins. (CA: Basic only)

	Length: 1. Default: 1. Units: rings.
ca_hi1ceil	High 1 Ceiling: Maximum 2nd high duration for a retrain. (CA: Basic only)
	Length: 2. Default: 78. Units: 10 ms.
ca_lo1ceil	Low 1 Ceiling: Maximum 1st low duration for a retrain. (CA: Basic only)
	Length: 2. Default: 58. Units: 10 ms.
ca_lowerfrq	Lower Frequency: Lower bound for 1st tone in an SIT. (CA)
	Length: 2. Default: 900. Units: Hz.
ca_upperfrq	Upper Frequency: Upper bound for 1st tone in an SIT. (CA)
	Length: 2. Default: 1000. Units: Hz.
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 upperfrq and lowerfrq for it to be considered valid. (CA)
	Length: 1. Default: 5. Units: 10 ms.
ca_rejctfrq	Not used.
ca_maxansr	Maximum Answer: The maximum allowable length of ansrsize. When ansrsize exceeds maxansr, a *connect* is returned to the application. (CA)
	Length: 2. Default: 1000. Units: 10 ms.
ca_ansrdgl	Answer Deglitcher: The maximum silence period allowed between words in a salutation. This parameter should be enabled only when you are interested in measuring the length of the salutation. (CA)
	-1: Disable this condition.

	Length: 2. Default: -1. Units: 10 ms.
ca_pvdmxper	Not used.
ca_pvdszwnd	.Not used.
ca_pvddly	Not used.
ca_mxtimefrq	Maximum Time Frequency: Maximum allowable time for 1st tone in an SIT to be present.
	Default: 0. Units: 10 ms..
ca_lower2frq	Lower Bound for 2nd Frequency: Lower bound for 2nd tone in an SIT.
	Default: 0. Units: Hz.
ca_upper2frq	Upper Bound for 2nd Frequency: Upper bound for 2nd tone in an SIT.
	Default: 0. Units: Hz. .
ca_time2frq	Time for 2nd Frequency: Minimum time for 2nd tone in an SIT to remain in bounds.
	Default: 0. Units: 10 ms.
ca_mxtime2frq	Maximum Time for 2nd Frequency: Maximum allowable time for 2nd tone in an SIT to be present.
	Default: 0. Units: 10 ms.
ca_lower3frq	Lower Bound for 3rd Frequency: Lower bound for 3rd tone in an SIT.
	Default: 0. Units: Hz.
ca_upper3frq	Upper Bound for 3rd Frequency: Upper bound for 3rd tone in an SIT.
	Default: 0. Units: Hz.
ca_time3frq	Time for 3rd Frequency: Minimum time for 3rd tone in an SIT to remain in bounds.

	Default: 0. Units: 10 ms.
ca_mxtime3frq	Maximum Time for 3rd Frequency: Maximum allowable time for 3rd tone in an SIT to be present.
	Default: 0. Units: 10 ms.
ca_dtn_pres	Dial Tone Present: Length of time that a dial tone must be continuously present. (CA: Enhanced only)
	Default: 100. Units: 10 ms.
ca_dtn_npres	Dial Tone Not Present: Maximum length of time to wait before declaring dial tone failure. (CA: Enhanced only)
	Default: 300. Units: 10 ms.
ca_dtn_deboff	Dial Tone Debounce: Maximum gap allowed in an otherwise continuous dial tone before it is considered invalid. (CA: Enhanced only)
	Default: 10. Units: 10 ms.
ca_pamd_failtime	PAMD Fail Time: Maximum time to wait for Positive Answering Machine Detection or Positive Voice Detection after a cadence break. (CA: Enhanced only)
	Default: 400. Units: 10 ms.
ca_pamd_minring	Minimum PAMD Ring: Minimum allowable ring duration for Positive Answering Machine Detection. (CA: Enhanced only)
	Default: 190. Units: 10 ms.
ca_pamd_spdval	PAMD Speed Value: Quick or full evaluation for PAMD detection.
	PAMD_FULL = Full evaluation of response
	PAMD_QUICK = Quick look at connect circumstances (CA: Enhanced only)
	Default: PAMD_FULL.

ca_pamd_qtemp	PAMD Qualification Template: Which PAMD template to use. Options are PAMD_QUAL1TMP or PAMD_QUAL2TMP; at present, only PAMD_QUAL1TMP is available. (CA: Enhanced only) Default: PAMD_QUAL1TMP.
ca_noanswer	No Answer: Length of time to wait after first ringback before deciding that the call is not answered. (CA: Enhanced only) Default: 3000. Units: 10 ms.
ca_maxintering	Maximum Inter-ring Delay: Maximum time to wait between consecutive ringback signals before deciding that the call has been connected. (CA: Enhanced only) Default: 800. Units: 10 ms.

4.1.3. DX_CST - *call status transition structure*

DX_CST contains call status transition information after an asynchronous TDX_CST termination or TDX_SETHOOK event occurs. Use the Event Management function, **sr_getevtdatap( )** (see *Appendix* A) to retrieve the structure.

The typedef for DX_CST is shown below:

```
typedef struct DX_CST {
   unsigned short cst_event;
   unsigned short cst_data;
} DX_CST;
```

cst_event contains one of the following events:

DE_DIGITS	• received a digit
DE_LCOFF	• loop current off event
DE_LCON	• loop current on event
DE_LCREV	• loop current reversal event
DE_RINGS	• rings received event
DE_RNGOFF	• caller hang up (incoming call is dropped before being

accepted) event

- DE_SILOFF • silence off event
- DE_SILON • silence on event
- DE_TONEOFF • tone off event
- DE_TONEON • tone on event
- DE_WINK • received a wink
- DX_OFFHOOK • offhook event
- DX_ONHOOK • onhook event

NOTE: DX_ONHOOK and DX_OFFHOOK are returned if a TDX_SETHOOK termination event is received.

cst_data contains data associated with **cst_event**. Valid values are:

CST event type	CST event data
DE_DIGITS	ASCII digit (low byte) and the digit type (high byte)
DE_LCOFF	time previous last loop current "on" transition in 10 ms units
DE_LCON	time since previous loop current "off" transition in 10 ms units
DE_LCREV	time since previous loop current reversal transition in 10 ms units
DE_RINGS	0
DE_SILOFF	time since previous silence started in 10 ms units
DE_SILON	time since previous silence stopped in 10 ms units
DE_TONEOFF	user-specified tone ID
DE_TONEON	user-specified tone ID
DE_WINK	N/A
DX_OFFHOOK	N/A
DX_ONHOOK	N/A

4.1.4. DX_EBLK- *call status event block structure*

This structure is returned by **dx_getevt( )** indicates which Call Status Transition event occurred.

NOTE: **dx_getevt( )** is a synchronous function which blocks until an event occurs. For information about asynchronously waiting for CST events, see **dx_setevtmsk( )**.

The typedef for the structure is shown below:

```
typedef struct DX_EBLK {
  unsigned short ev_event;      /* Event that occurred */
  unsigned short ev_data;       /* Event specific data */
  unsigned char ev_rfu[12];     /* Reserved for future use*/
  }DX_EBLK;
```

where:

ev_event specifies the event that occurred on the device. The following defines can be entered in **ev_event:**

DE_DIGITS	• digit received
DE_LCOFF	• loop current off
DE_LCON	• loop current on
DE_LCREV	• loop current reversal
DE_RINGS	• rings received
DE_SILOFF	• non-silence detected
DE_SILON	• silence detected
DE_TONEOFF	• tone off event occurred
DE_TONEON	• tone on event occurred
DE_WINK	• wink has occurred

ev_data contains data specific to the event contained in **ev_event**. Table 6 shows what data is returned for each event that appears in **ev_event**. All the lengths of time are in 10 ms units.

Table 6. Values Returned in ev_data

Event	Data Returned in ev_data
DE_DIGITS	Digit and digit type
DE_LCOFF	The length of time that loop current was on before the loop-current-off event was detected
DE_LCON	The length of time that loop current was off before the loop-current-on event was detected
DE_LCREV	The length of time that loop current was reversed before the loop-current-reversal event was detected
DE_RINGS	Returns no data
DE_SILOFF	The length of time that silence occurred before non-silence (noise or meaningful sound) was detected
DE_SILON	The length of time that non-silence occurred before silence was detected
DE_TONEOFF	The tone ID for the tone-off event
DE_TONEON	The tone ID for the tone-on event
DE_WINK	Returns no data

ev_rfu is reserved for future use.

4.1.5. DX_IOTT - *I/O transfer table*

The DX_IOTT structure identifies a source or destination for voice data. It is used with the **dx_play( )** and **dx_rec( )** functions.

The typedef for the structure is shown below:

```
typedef struct dx_iott {
   unsigned short io_type;          /* Transfer type */
   unsigned short rfu;              /* Reserved */
   int              io_fhandle;     /* File descriptor */
   char *           io_bufp;        /* Pointer to base memory */
   unsigned long    io_offset;      /* File/Buffer offset */
   long int         io_length;      /* Length of data */
   DX_IOTT          *io_nextp;      /* Ptr to next DX_IOTT if IO_LINK set */
   DX_IOTT          *io_prevp;      /* (Optional) Ptr to previous DX_IOTT */
}DX_IOTT;
```

where:

io_type specifies whether the data is stored in a file or in memory. It also determines if the next DX_IOTT structure is contiguous in memory, linked, or if this is the last DX_IOTT in the chain. Set the **io_type** field to an OR combination of the required defines listed below.

Specify data transfer type as follows:

IO_DEV	• data is being played from a file
IO_MEM	• data is being played from memory

Specify structure linkage as follows:

IO_CONT	• the next *DX_IOTT* structure is contiguous (default)
IO_LINK	• the next DX_IOTT structure is part of a linked list
IO_EOT	• this is the last DX_IOTT structure in the chain

If none of IO_CONT, IO_LINK, or IO_EOT are specified, IO_CONT is assumed.

io_fhandle contains a unique file descriptor if IO_DEV is set in **io_type**. If IO_DEV is not set in **io_type**, **io_fhandle** should be set to 0.

io_bufp field indicates a base memory address if IO_MEM is set in **io_type**.

io_offset indicates

- an offset from the beginning of a file if IO_DEV is specified in **io_type**
- an offset from the base buffer address specified in **io_bufp** if IO_MEM is specified in **io_type**.

io_length indicates the number of bytes allocated for recording or the byte length of the playback file. Specify -1 to play until end of data. During **dx_play( )**, a value of -1 causes playback to continue until an EOF is received or one of the terminating conditions is satisfied. During **dx_rec( )**, a value of -1 in **io_length** causes recording to continue until one of the terminating conditions is satisfied.

io_nextp points to the next DX_IOTT structure in the linked list if IO_LINK is set in **io_type**.

io_prevp points to the previous DX_IOTT structure. This field is automatically filled in when **dx_rec( )** or **dx_play( )** are called. The **io_prevp** field of the first DX_IOTT structure is set to NULL.

A DX_IOTT structure describes a single data transfer to or from one file, memory block, or custom device. If the voice data is stored on a custom device, the device must have a standard Windows NT device interface. The device must support **open( )**, **close( )**, **read( )**, and **write( )** and **lseek( )**.

To use multiple combinations, each source or destination of I/O is specified as one element in an array of DX_IOTT structures. The last DX_IOTT entry must have IO_EOT specified in the **io_type** field.

For example, to use different sources for playback, an array or linked list of DX_IOTT structures can be specified as follows:

Playback Array Example

```
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
DX_IOTT iott[3];
/* first iott: voice data in a file with descriptor fd1*/
iott[0].io_fhandle = fd1;
iott[0].io_offset = 0;
iott[0].io_length = -1;
iott[0].io_type = IO_DEV;
/* second iott: voice data in a file with descriptor fd2 */
iott[1].io_fhandle = fd2;
iott[1].io_offset = 0;
iott[1].io_length = -1;
iott[1].io_type = IO_DEV;
/* third iott: voice data in a file with descriptor fd3 */
iott[2].io_fhandle = fd3;
iott[2].io_offset = 0;
iott[2].io_length = -1;
iott[2].io_type = IO_DEV|IO_EOT;
```

```
      .
      .
/* play all three voice files: pass &iott[0] as argument to dx_play( )
      .
      .
/* form a linked list of iott[0] and iott[2] */
iott[0].io_nextp=&iott[2];
iott[0].io_type|=IO_LINK
/* pass &iott[0] as argument to dx_play( ). This time only files 1 and 3
 * will be played.
 */
    .
```

4.1.6. DX_SVMT - *speed/volume modification table structure*

The DX_SVMT structure has 21 entries that represent different levels of speed or volume. This structure is used to set or retrieve the Speed/Volume Modification Table values (using **dx_setsvmt( )** and **dx_getsvmt( )** respectively).

The DX_SVMT typedef is shown below:

```
typedef struct DX_SVMT {
   char   decrease[10];       /* Ten Downward Steps */
   char   origin;             /* Regular Speed or Volume */
   char   increase[10];       /* Ten Upward Steps */
} DX_SVMT;
```

Table 7 describes the valid speed and volume settings for each of the DX_SVMT entries.

NOTE: Although there are 21 entries available in the DX_SVMT structure, they do not all have to be utilized for changing speed or volume - the range can be as small as you require. Ensure that you insert -128 (80h) in any entries that do not contain a speed or volume adjustment.

Table 7. DX_SVMT Entries

Field	Description
decrease[10]	Array that provides a maximum of 10 downward steps in speed or volume. The size of the steps is specified in this table. Specify the value -128 (80h) in any entries you are not using. Valid Values:

Speed - Percentage decrease from the origin (which is set to 0). Valid Values must be between -1 and -50.

Volume - Step in dB from the origin (which is set to 0). Valid Values must be between -1 and -30.

origin Represents the standard play speed or volume. This is the original setting or starting point for Speed and Volume Control.

Valid Values:

Set to 0 for speed or volume.

increase[10] Array that provides a maximum of 10 upward steps in speed or volume. The size of the steps is specified in this table. Specify the value -128 (80h) in any entries you are not using.

Valid Values:

Speed - Percentage increase from the origin (which is set to 0). Valid Values must be between 1 and 50. Specify the value -128 (80h) in any entries you are not using.

Volume - Step in dB from the origin (which is set to 0). Valid Values must be between 1 and 10. Specify the value -128 (80h) in any entries you are not using.

4.1.7. DX_SVCB - *speed/volume adjustment condition block*

This structure is used by **dx_setsvcond( )** to set the following:

- which Speed/Volume Modification Table to use (speed or volume)
- adjustment type (increase/decrease, absolute value, toggle)
- adjustment conditions (incoming digit, beginning of play)
- level/edge sensitivity for incoming digits

The typedef for the DX_SVCB structure is described below:

```
typedef struct DX_SVCB {
   unsigned short type;      /* Bit Mask */
   short adjsize;            /* Adjustment Size */
   unsigned char digit;      /* ASCII digit value that causes the action */
   unsigned char digtype;    /* Digit Type (e.g., 0 = DTMF) */
} DX_SVCB;
```

NOTES: **1.** To clear the DX_SVCB adjustment condition blocks, call the **dx_clrsvcond( )** function.

2. DX_SVCB adjustment condition blocks can only be added to the existing conditions. To reset or remove any DX_SVCB adjustment condition blocks, all conditions must be cleared (using **dx_clrsvcond( )**).

Table 8 describes each of the valid values for the fields.

Table 8. DX_SVCB Entries

Defines (type field)	Description (for type field)	Description (for adjsize field)
Speed or Volume		
Choose one:		
SV_SPEEDTBL	Modify speed table.	N/A.
SV_VOLUMETBL	Modify volume table.	N/A.
Adjustment Type		
Choose one:		
SV_ABSPOS	Sets **adjsize** field to indicate an absolute volume adjustment position position in the Speed or Volume Modification Tables.	Specify the required speed or between -10 and +10 in the Volume Modification Speed or Tables.

Defines (type field)	Description (for type field)	Description (for adjsize field)
SV_RELCURPOS	Sets **adjsize** field to indicate a number of steps by which to adjust speed or volume.	Specify how many positive or negative "steps" in the Speed or Volume Modification Tables by which to adjust the speed or volume. For example, specify -2 to lower the speed or volume by 2 steps in the Speed or Volume Modification Tables.
SV_TOGGLE	Sets **adjsize** field to use one of the toggle defines	Set the "toggle values by specifying one of the following: SV_TOGORIGIN - sets the current speed or volume to toggle between the origin and the last modified level of speed or volume. SV_CURORIGIN - resets the current speed or volume level to the origin (i.e., regular speed or volume). SV_CURLASTMOD - sets the current speed or volume to the last modified speed volume level.

Defines (type field)	**Description (for type field)**	**Description (for adjsize field)**
		SV_RESETORIG - resets the current speed or volume to the origin and the last modified speed or volume in the origin.
Optional -Choose one:		
SV_LEVEL	Sets the digit adjustmentcondition to be level sensitive. At the start of play, adjustments will be made according to adjustment condition digits contained in the digit buffer. If SV_LEVEL is not specified, the digit adjustment condition is edge sensitive, and will wait for a new occurrence of the digit before play adjusting.	N/A.
SV_BEGINPLAY	Adjusts speed or volume at the beginning of each play. In this case, digit and digtype fields are ignored.	N/A.
Choose one:		

Defines (type field)	Description (for type field)	Description (for adjsize field)
0-9, a-d, #, *	ASCII digit that adjusts play.	N/A.
Digit Type		
Specify the following: DG_DTMF	Specifies DTMF digits.	N/A.

Volume Example

The following *DX_SVCB* structure is set to decrease the volume by one step whenever the DTMF digit "1" is detected:

```
svcb[0].type    = SV_VOLUMETBL | SV_RELCURPOS;
svcb[0].adjsize = - 1;
svcb[0].digit    = '1';
svcb[0].digtype    = DG_DTMF;
```

The following DX_SVCB structure will set speed to the value in Speed Modification Table position 5 whenever the DTMF digit "2" is detected:

```
svcb[0].type    = SV_SPEEDTBL | SV_ABSPOS;
svcb[0].adjsize = 5;
svcb[0].digit    = '2';
svcb[0].digtype    = DG_DTMF;
```

4.1.8. DX_UIO - *user-definable I/O structure*

This structure, returned by **dx_setuio( )**, contains pointers to user-defined I/O functions for accessing nonstandard storage devices.

```
/*
 * Structure for user-defined I/O functions
 */
typedef struct DX_UIO {

   int  (*u_read) ( );
   int  (*u_write) ( );
   int  (*u_seek) ( );
} DX_UIO;
```

The **u_read** field is a pointer to the user-defined **read( )** function, which returns an integer equal to the number of bytes read or -1 for error.

The **u_write** field is a pointer to the user-defined **write( )** function, which returns an integer equal to the number of bytes written or -1 for error.

The **u_seek** field is a pointer to the user-defined **lseek( )** function, which returns a long equal to the offset into the I/O device where the read or write is to start or -1 for error.

4.1.9. TN_GEN - *tone generation template structure*

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 shown below:

```
typedef struct {
   unsigned short tg_dflag;  /* Dual Tone - 1, Single Tone - 0 */
   unsigned short tg_freq1;  /* Frequency for Tone 1 (HZ)      */
   unsigned short tg_freq2;  /* Frequency for Tone 2 (HZ)      */
   short          tg_ampl1;  /* Amplitude for Tone 1 (dB)      */
   short          tg_ampl2;  /* Amplitude for Tone 2 (dB)      */
   short          tg_dur;    /* Duration of the Generated Tone */
                             /*  Units = 10ms          */
} TN_GEN;
```

Table 9 lists the valid values for each field.

Table 9. TN_GEN Values

TN_GEN Field	Description	
tg_dflag	Specifies single or dual tone. If single, the values in tg_freq2 and tg_ampl2 will be ignored.	
	Choose one:	
	TN_SINGLE	single tone
	TN_DUAL	dual tone

TN_GEN Field	Description
tg_freq1	Frequency in Hz for tone 1 (range 200 to 2000 Hz)
tg_freq2	Frequency in Hz for tone 2; (range 200 to 2000 Hz)
tg_ampl1	Amplitude in dB for tone 1; (range 0 to -40 dB)
tg_ampl2	Amplitude in dB for tone 2; (range 0 to -40 dB)
tg_dur	Duration of the tone in 10 ms units (-1 = infinite duration)

4.1.10. DX_XPB - *I/O transfer parameter block*

The *k*I/O Transfer Parameter Block (DX_XPB) data structure is used by the extended play and record functions to specify the file format (either VOX file or WAVE file), the data format (ADPCM, Mu-law PCM, A-law PCM, Linear PCM), the sampling rate (6, 8, or 11 KHz), and the resolution (4 or 8 bits per sample).

```
typedef struct {
      USHORT      wFileFormat;   // file format
      USHORT      wDataFormat;            // audio data format
      ULONG       nSamplesPerSec;         // sampling rate
      ULONG       nBitsPerSample;         // bits per sample
} DX_XPB;
```

The **dx_playwav( )** convenience function does not specify a DX_XPB structure because the WAVE file contains the necessary format information.

DX_XPB Field	Description	
wFileFormat	Specifies one of the following audio file formats. Note that this field is ignored by the convenience functions **dx_recwav( )** , **dx_playwav( )** , **dx_recvox( )** ,.and **dx_playvox( )** .	
	Choose one:	
	FILE_FORMAT_VOX	Dialogic VOX file format
	FILE_FORMAT_WAVE	Microsoft WAVE file format

DX_XPB Field	Description	
wDataFormat	specifies one of the following data formats	
	Choose one:	
	DATA_FORMAT_DIALOGIC_ADPCM	4-bit OKI ADPCM (Dialogic registered)
	DATA_FORMAT_MULAW	8-bit mu-law PCM
	DATA_FORMAT_ALAW	8-bit a-law PCM
	DATA_FORMAT_PCM	8-bit Linear PCM
nSamplesPerSec	specifies one of the following digitization rates	
	Choose one:	
	DRT_6KHZ	6 KHz sampling rate.
	DRT_8KHZ	8 KHz sampling rate.
	DRT_11KHZ	11 KHz sampling rate. Note: 11KHz OKI ADPCM is not supported.
nBitsPerSample	specifies number of bits per sample. This field must be set to 4 for ADPCM and 8 for all others.	

4.2. Voice Board Parameter Defines for dx_getparm( )

The and dx_setparm( ) file <*install drive:*>\<*install directory*>*dialogic \include\dxxxlib.h* contains defined masks for the D/4x parameters that can be examined and set using **dx_getparm( )** and **dx_setparm( )**. These parameters apply to any real or emulated D/4x board.

Channels and boards have different parameters. The board parameters, their default settings, read/write privileges, and descriptions are listed in Table 10.

The channel parameters are listed in Table 11. All time units are in 10 ms unless otherwise noted.

Table 10. Voice Board Parameters

Define	Bytes	Read/ Write	Default	Description
DXBD_CHNUM	1	R	-	Channel Number - number of channels on the board
DXBD_FLASHCHR	1	R/W	&	Flash character - character that causes a hook flash when detected.
DXBD_FLASHTM	2	R/W	50	Flash Time - length of time onhook during flash
DXBD_HWTYPE	1	R	-	Hardware Type - value can be:
	TYP_D40			D/40 board
	TYP_D41			D/21, D/41, D/xxxSC board
DXBD_MAXPDOFF	2	R/W	50	Maximum Pulse Digit Off - max. time loop current may be off before the existing loop pulse digit is considered invalid and reception is reinitialized

Define	Bytes	Read/ Write	Default	Description
DXBD_MAXSLOFF	2	R/W	25	Maximum Silence Off - maximum time for silence being off, during audio pulse detection
DXBD_MINIPD	2	R/W	25	Minimum Loop Interpulse Detection - minimum time between loop pulse digits during loop pulse detection
DXBD_MINISL	2	R/W	25	Minimum Interdigit Silence - minimum time for silence on between pulse digits for audio pulse detection
DXBD_MINLCOFF	2	R/W	0	Minimum Loop Current Off - minimum time before loop current drop message is sent
DXBD_MINPDOFF	1	R/W	2	Minimum Pulse Detection Off - minimum break interval for valid loop pulse detection
DXBD_MINPDON	1	R/W	2	Minimum Pulse Detection On - minimum make interval for valid

Define	Bytes	Read/ Write	Default	Description
				loop pulse detection
DXBD_MINSLOFF	1	R/W	2	Minimum Silence Off - min. time for silence to be off for valid audio pulse detection
DXBD_MINSLON	1	R/W	1	Minimum Silence On - min. time for silence to be on for valid audio pulse detection
DXBD_MINTIOF	1	R/W	5	Minimum DTI Off - minimum time required between rings-received events
DXBD_MINTION	1	R/W	5	Minimum DTI On - minimum time required for rings received event
DXBD_OFFHDLY	2	R/W	50	Offhook Delay - period after offhook, during which no events are generated e.g., no DTMF digits will be detected during this period.
DXBD_PAUSETM	2	R/W	200	Pause Time - delay caused by a comma in the dialing string

Define	Bytes	Read/ Write	Default	Description
DXBD_P_BK	2	R/W	6	Pulse Dial Break - duration of pulse dial off-hook interval
DXBD_P_IDD	2	R/W	100	Pulse Interdigit Delay - time between digits in pulse dialing
DXBD_P_MK	2	R/W	4	Pulse Dial Make - duration of pulse dial offhook interval
DXBD_R_EDGE	1	R/W	ET_ROFF	Ring Edge - detection of ring edge, values can be: ET_RON · beginning of ring ET_ROFF · end of ring
DXBD_R_IRD	2	R/W	80	Inter-ring Delay - maximum time to wait for the next ring (100 ms units). Used to distinguish between calls. Set to 1 for T-1 applications.
DXBD_R_OFF	2	R/W	5	Ring-off Interval - minimum time for ring not to be present before qualifying as "not ringing" (100 ms units)
DXBD_R_ON	2	R/W	3	Ring-on Interval - minimum time ring must be present to

Define	Bytes	Read/ Write	Default	Description
				qualify as a ring (100 ms units)
DXBD_SYSCFG	1	R	-	System Configuration - JP8 status for D/4x boards: 0 = loop start interface (JP8 in).; 1 = DTI/xxx interface (JP8 out):
DXBD_S_BNC	2	R/W	4	Silence and Non-silence Debounce - length of a changed state before Call Status Transition message is generated
DXBD_TTDATA	1	R/W	10	Duration of DTMF digits for dialing.
DXBD_MFMINON	2	R/W	0	Minimum MF On - The duration to be added to the standard MF tone duration before the tone is detected. The minimum detection duration is 65 ms for KP tones and 40 ms for all other tones. This parameter affects all the channels on the board. (10 ms units)
DXBD_MFTONE	2	R/W	6	MF Minimum Tone Duration - The duration of a dialed

Define	Bytes	Read/ Write	Default	Description
				MF tone. This parameter affects all the channels on the board. Maximum value: 10 (10 ms units)
DXBD_MFDELAY	2	R/W	6	MF Interdigit Delay - The length of the silence period between tones during MF dialing. This parameter affects all the channels on the board. (10 ms units)
DXBD_MFLKPTONE	2	R/W	10	MF Length of LKP Tone - The length of the LKP tone during MF dialing. This parameter affects all the channels on the specified board. Maximum value: 15 (10 ms units)
DXBD_T_IDD	2	R/W	5	DTMF Interdigit Delay - time between digits in DTMF dialing
DXBD_MINOFFHKTM	2	R/W	250	Minimum offhook time (10 ms)

Define	Bytes	Read/ Write	Default	Description
DXCH_DFLAGS	2	R/W	0	DTMF detection edge select
DXCH_DTINITSET	2	R/W	0	Specifies which DTMF digits to initiate play on. Values of different DTMF digits may be ORed together to form the bit mask. Possible values are listed below:
	Value			**DTMF Digit**
	-DM_1			1
	-DM_2			2
	-DM_3			3
	-DM_4			4
	-DM_5			5
	-DM_6			6
	-DM_7			7
	-DM_8			8
	-DM_9			9
	-DM_0			0
	-DM_S			*
	-DM_P			#
	-DM_A			a
	-DM_B			b
	-DM_C			c
	-DM_D			d
DXCH_DTMFTLK	2	R/W	5	Sets the minimum time for DTMF to be present during playback to be considered valid.

Define	Bytes	Read/ Write	Default	Description
				Increasing the value provides more immunity to talk-off/playoff.
Set to -1 to disable.				
DXCH_DTMFDEB	2	R/W	0	DTMF debounce time - maximum length of time in which DTMF can be absent and then come back on again and still be considered the same DTMF tone.
DXCH_MFMODE	2	R/W	2	This is a word-length bit mask that selects the minimum length of KP tones to be detected. The possible values of this field are: 0 - detect KP tone > 40 ms 2 - detect KP tone > 65 ms
				If the value is set to 2, any KP tone greater than 65ms will be returned to the application during MF detection. This ensures that only standard-length KP tones (100ms) are detected. If set to 0 (zero), any KP tone longer than 40ms will be detected.

Define	Bytes	Read/ Write	Default	Description
DXCH_MAXRWINK	1	R/W	20	Maximum Loop Current for Wink - The maximum time that loop current needs to be on before recognizing a wink (10 ms units)
DXCH_MINRWINK	1	R/W	10	Minimum Loop Current for Wink - The minimum time that loop current needs to be on before recognizing a wink (10 ms units)
DXCH_WINKDLY	1	R/W	15	Wink Delay - The delay after a ring is received before issuing a wink (10 ms units)
DXCH_RINGCNT	2	R/W	4	Number of rings to wait before returning a ring event.
DXCH_WINKLEN	1	R/W	15	Wink Length - The duration of a wink in the off-hook state (10 ms units)
DXCH_PLAYDRATE	2	R/W	6000	Play Digitization Rate - This parameter sets the digitization rate of the voice data that is

Define	Bytes	Read/ Write	Default	Description
				played on this channel. Voice Data can be played at 6k or 8k sampling rates. Valid parameter values are: 6000 - 6K sampling rate 8000 - 8k sampling rate Voice data must be played at the same rate it was recorded at.
DXCH_RECRDRATE	2	R/W	6000	Record Digitization Rate - This parameter sets the rate at which the recorded voice data is digitized. Voice Data can be digitized at 6k or 8k sampling rates. Valid values are: 6000 - 6K sampling rate 8000 - 8k sampling rate

Table 11. Voice Channel Parameters

Define	Bytes	Read/ Write	Default	Description
DXCH_DFLAGS	2	R/W	0	DTMF detection edge

Define	Bytes	Read/ Write	Default	Description
				select
DXCH_DTINITSET	2	R/W	0	Specifies which DTMF digits to initiate play on. Values of different DTMF digits may be ORed together to form the bit mask. Possible values are listed below: Value — DTMF Digit -DM_1 — 1 -DM_2 — 2 -DM_3 — 3 -DM_4 — 4 -DM_5 — 5 -DM_6 — 6 -DM_7 — 7 -DM_8 — 8 -DM_9 — 9 -DM_0 — 0 -DM_S — * -DM_P — # -DM_A — a -DM_B — b -DM_C — c -DM_D — d
DXCH_DTMFTLK	2	R/W	5	Sets the minimum time for DTMF to be present during playback to be considered valid.

Define	Bytes	Read/ Write	Default	Description
				Increasing the value provides more immunity to talk-off/playoff.
Set to -1 to disable.				
DXCH_DTMFDEB	2	R/W	0	DTMF debounce time - maximum length of time in which DTMF can be absent and then come back on again and still be considered the same DTMF tone.
DXCH_MFMODE	2	R/W	2	This is a word-length bit mask that selects the minimum length of KP tones to be detected. The possible values of this field are: 0 - detect KP tone > 40 ms 2 - detect KP tone > 65 ms If the value is set to 2, any KP tone greater than 65ms will be returned to the application during MF detection. This ensures that only standard-length KP tones (100ms) are detected.

Define	Bytes	Read/ Write	Default	Description
				If set to 0 (zero), any KP tone longer than 40ms will be detected.
DXCH_MAXRWINK	1	R/W	20	Maximum Loop Current for Wink - The maximum time that loop current needs to be on before recognizing a wink (10 ms units)
DXCH_MINRWINK	1	R/W	10	Minimum Loop Current for Wink - The minimum time that loop current needs to be on before recognizing a wink (10 ms units)
DXCH_WINKDLY	1	R/W	15	Wink Delay - The delay after a ring is received before issuing a wink (10 ms units)
DXCH_RINGCNT	2	R/W	4	Number of rings to wait before returning a ring event.
DXCH_WINKLEN	1	R/W	15	Wink Length - The duration of a wink in the off-hook state (10 ms units)

Define	Bytes	Read/ Write	Default	Description
DXCH_PLAYDRATE	2	R/W	6000	Play Digitization Rate - This parameter sets the digitization rate of the voice data that is played on this channel. Voice Data can be played at 6k or 8k sampling rates. Valid parameter values are: 6000 - 6K sampling rate 8000 - 8k sampling rate Voice data must be played at the same rate it was recorded at.
DXCH_RECRDRATE	2	R/W	6000	Record Digitization Rate - This parameter sets the rate at which the recorded voice data is digitized. Voice Data can be digitized at 6k or 8k sampling rates. Valid values are: 6000 - 6K

Define	Bytes	Read/ Write	Default	Description
				sampling rate 8000 - 8k sampling rate

5. Voice Programming Conventions

This chapter provides several techniques that you can use to simplify programming with the Dialogic Voice Library.

5.1. Always Check Return Code in Voice Programming

All the Dialogic Voice Library functions return a value to indicate success or failure of the function. All Voice Library functions indicate success by a return value of zero or a non-negative number.

NOTE: Asynchronous I/O functions return immediately to indicate success or failure of the function *initiating*.

Extended Attribute functions that return pointers return a pointer to the ASCIIZ string "Unknown device" if they fail.

Extended Attribute functions that do not return pointers return a value of AT_FAILURE if they fail.

Non-attribute functions return a value of -1 to indicate a failure.

If a function has failed, the reason for failure can be found by calling the Standard Attribute functions **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )**. These functions are described in the *Standard Runtime Library Programmer's Guide for Windows NT*.

If the error is EDX_SYSTEM check **errno**.

When using the asynchronous programming model you should always install a handler to get TDX_ERROR events.

5.2. Clearing Voice Structures

Two library functions are provided to clear structures. **dx_clrcap( )** clears DX_CAP structures and **dx_clrtpt( )** clears DV_TPT structures. See the function descriptions for details.

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

5.3. Using the Voice dx_playf( ) and dx_recf( ) Convenience Functions

dx_playf( ) and **dx_recf( )** are synchronous Voice Library functions provided as a convenience to the programmer. These functions are specific cases of the **dx_play( )** and **dx_rec( )** functions.

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. The **dx_recf( )** provides the same single file convenience for the **dx_rec( )** function.

5.4. Using the Voice Asynchronous Programming Model

Asynchronous programming allows you to have multiple threads of control within the one process. Each of the I/O functions can operate synchronously or asynchronously. See the *Standard Runtime Library Programmer's Guide for Windows NT* for information about asynchronous programming models.

5.5. Using Multiple Processes in Voice Synchronous Applications

When writing multiple processes for synchronous applications, you should use the following model: Create a master control process and spawn of a child process for each channel. Each child process is responsible for:

- opening and closing the channel
- adjusting the channel parameters
- performing channel-specific operations
- monitoring events that occur on the channel

NOTE: In an application that spawns a child process from a parent process, a device handle is not inheritable by the child process. Devices must be opened in the child process.

Appendix A
Standard Runtime Library

Voice Device Entries and Returns

The Standard Runtime Library is a device-independent library containing Event Management functions, Standard Attribute functions and the DV_TPT Termination Parameter Table. Dialogic SRL functions and data structures are described fully in the *Standard Runtime Library Programmer's Guide for Windows NT*.

This appendix lists the Voice board entries and returns for each of the Standard Runtime Library (SRL) components.

Event Management Functions

The Event Management functions retrieve and handle Voice device termination events for the following functions:

- **dx_dial( )**
- **dx_getdig( )**
- **dx_play( )**
- **dx_rec( )**
- **dx_playtone( )**
- **dx_sethook( )**
- **dx_wink( )**
- **r2_playbsig( )**

Each of the Event Management functions applicable to the Voice boards are listed in the following tables. Table 12 and Table 13 list values that are required by or returned for event management functions that are used with Voice devices.

Table 12. Voice Device Inputs for Event Management Functions

Event Management Function	Voice Device Input	Valid Value
sr_enbhdlr( ) *Enable event handler*	evt_type	TDX_PLAY TDX_PLAYTONE TDX_RECORD TDX_GETDIG TDX_DIAL TDX_CALLP TDX_CST TDX_SETHOOK TDX_WINK TDX_ERROR
sr_dishdlr( ) *Disable event handler*	evt_type	As Above

Table 13. Voice Device Returns from Event Management Functions

Event Management Function	Return Description	Returned Value
sr_getevtdev( ) *Get Dialogic Device handle*	device	Voice device handle
sr_getevttype( )*Get event type*	event type	TDX_PLAY TDX_PLAYTONE TDX_RECORD TDX_GETDIG TDX_DIAL TDX_CALLP TDX_CST TDX_SETHOOK TDX_WINK TDX_ERROR

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

Standard Attribute Functions

Standard Attribute functions return general Dialogic device information, such as the device name or the last error that occurred on the device. The Standard Attribute functions and the Voice device information they return are listed in Table 14.

Table 14. Standard Attribute Functions

Standard Attribute Function	Information Returned for Voice Devices
ATDV_ERRMSGP( )	Pointer to string describing the error that occurred during the last function call on the specified device. (See *Appendix* B for a list of all the possible errors, and see each function description for possible errors for that function).
ATDV_IOPORT( )	0 for D/21D, D/41D, D/21E, D/41E, D/41ESC, D/160SC-LS, D/240SC, D/240SC-T1, D/300SC-E1, and D/320SC boards. Valid port address of the SpringBoard device.
ATDV_IRQNUM( )	Interrupt number for the specified device.
ATDV_LASTERR( )	The error that occurred during the last function call on a specified device. See the function description for possible errors for the function.
ATDV_NAMEP( )	Pointer to device name (e.g., dxxxB*b*C*c*). Refer

	to the *System Release Software Installation Reference for Windows NT* for information about device names.
ATDV_SUBDEVS( )	Number of sub-devices (channels) (Refer to the *Standard Runtime Library Programmer's Guide for Windows NT* for information on sub-devices): 4 for a D/4x board (emulated or real) 2 for a D/2x board

DV_TPT Structure

The DV_TPT termination parameter table sets termination conditions for a range of Dialogic products. The valid values for the DV_TPT when using a Voice board are contained in this section.

The DV_TPT structure is used to set I/O function termination conditions. This structure is used by the following I/O functions:

- **dx_clrtpt( )**
- **dx_getdig( )**
- **dx_play( )**
- **dx_rec( )**
- **dx_playtone( )**

The I/O functions will terminate when one of the conditions set in the DV_TPT structure occurs. 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 terminating condition.

The structure has the following format:

```
typedef struct DV_TPT {
   unsigned short   tp_type;          /* Flags describing this entry */
   unsigned short   tp_termno;        /* Termination Parameter number */
   unsigned short   tp_length;        /* Length of terminator */
   unsigned short   tp_flags;         /* Parameter attribute flag */
   unsigned short   tp_data;          /* Optional additional data */
   unsigned short   rfu;              /* Reserved                 */
   DV_TPT           *tp_nextp;        /* Pointer to next termination
                                      * parameter if IO_LINK set
```

```
                                                    */
}DV_TPT;
```

Each field is defined in the sections that follow. Table 15 located after the field descriptions, contains a summary of the valid field settings for each termination condition.

tp_type

tp_type specifies whether the structure is part of a linked list, part of an array, or the last DV_TPT entry in the DV_TPT table. Enter one of the following defines in **tp_type**:

IO_LINK	• **tp_nextp** points to next DV_TPT structure
IO_EOT	• last DV_TPT in the chain
IO_CONT	• next DV_TPT entry is contiguous

tp_termno

tp_termno specifies the termination condition. The Voice device termination defines are

DX_MAXDTMF	• Maximum number of digits received
DX_MAXSIL	• Maximum length of silence
DX_MAXNOSIL	• Maximum length of non-silence
DX_LCOFF	• Loop current drop
DX_IDDTIME	• Maximum delay between digits
DX_MAXTIME	• Maximum function time
DX_DIGMASK	• Specific digit received
DX_PMOFF	• Pattern of non-silence
DX_PMON	• Pattern of silence
	• Digit termination for user-defined tone. (D/21D, D/41D, D/21E, D/41E, D/41ESC, D/81A, D/12x, D/160SC-LS, D/240SC, D/240SC-T1, D/300SC-E1 and D/320SC only).
DX_TONE	• Tone On/Off termination (D/21D, D/41D, D/41E, D/41ESC, D/81A, D/12x, D/160SC-LS, D/240SC, D/240SC-T1, D/300SC-E1 and D/320SC onlyGTD term conditions)

A more detailed description of these I/O terminations is contained in the *Voice Features Guide for Windows NT*.

NOTE: When using the DX_PMON and DX_PMOFF termination conditions, some of the DV_TPT fields are set differently from the other termination conditions. See the section, *Using DX_PMOFF and DX_PMON*, located at the end of this appendix for information.

You can call the Extended Attribute function **ATDX_TERMMSK( )** to determine all the terminating conditions that occurred. This function returns a bitmap of terminating conditions. The "TM_" defines corresponding to this bitmap of terminating conditions are provided in the function description for **ATDX_TERMMSK( )**.

tp_length

tp_length refers to the length or size for each specific terminating condition. When **tp_length** represents length of time for a terminating condition, the maximum value allowed is 6000. The field can represent the following:

Table 15. tp_length Settings

tp_length value	tp_length description
time in 10 or 100ms units	Applies to any terminating condition that specifies termination after a specific period of time, up to 6000.
# of digits	Applies when using DX_MAXDTMF which specifies termination after a certain number of digits is received.
digit type description	Applies when using DX_DIGTYPE which specifies termination on a user-specified digit. Specify the digit type in the high byte and the ASCII digit value in the low byte. See the Global Tone Detection functions in the *Voice Features Guide for Windows NT* for information.
digit bit mask	Applies to DX_DIGMASK, which specifies a bit mask of digits to terminate on. Set the digit bitmask using one or

more of the appropriate "Digit Defines" from the table below:

Digit	Digit Define	Digit Define	Digit Define	Digit	Digit Define
0	DM_0				
1	DM_1	6	DM_6	#	DM_P
2	DM_2	7	DM_7	a	DM_A
3	DM_3	8	DM_8	b	DM_B
4	DM_4	9	DM_9	c	DM_C
5	DM_5	*	DM_S	d	DM_D

number of pattern repetitions — Applies to DX_PMOFF, which specifies the number of times a pattern should repeat before termination.

NOTE: Then DX_PMON is the termination condition, **tp_length** contains the **tp_flags** information. See the **tp_flags** description and the *Using DX_PMON and DX_PMOFF* section (at the end of this Appendix) for information.

tp_flags

tp_flags is a bit mask representing various characteristics of the termination condition to use.

The defines for the termination flags are:

- TF_EDGE — termination condition is edge-sensitive
- TF_LEVEL — termination condition is level-sensitive
- TF_CLREND — history cleared when function terminates
- TF_CLRBEG — history cleared when function begins
- TF_USE — terminator used for termination
- TF_SETINIT — DX_MAXSIL only - initial length of silence to terminate on
- TF_10MS — set units of time to 10 ms (default is 100 ms)
- TF_FIRST — DX_IDDTIME only - start looking for termination condition (interdigit delay) to be satisfied after first digit is received

A set of default **tp_flags** values for the termination conditions is available. These default values are:

TF_MAXDTMF	(TF_LEVEL|TF_USE)
TF_MAXSIL	(TF_EDGE|TF_USE)
TF_MAXNOSIL	(TF_EDGE|TF_USE)
TF_LCOFF	(TF_LEVEL|TF_USE|TF_CLREND)
TF_IDDTIME	(TF_EDGE)
TF_MAXTIME	(TF_EDGE)
TF_DIGMASK	(TF_LEVEL)
	(TF_EDGE)
TF_TONE	(TF_LEVEL|TF_USE|TF_CLREND)
TF_DIGTYPE	(TF_LEVEL)

NOTES: **1.** DX_PMOFF and DX_PMON

2. DX_PMOFF does not have a default **tp_flags** value.

The **tp_flags** value for is set in **tp_length** (i.e., TF_PMON is set in tp_length). See the **tp_length** description and the *Using DX_PMON and DX_PMOFF* section (at the end of this Appendix) for information.

3. TF_IDDTIME or TF_MAXTIME must be specified in **tp_flags** if DX_IDDTIME or DX_MAXTIME are specified in **tp_termno**. Other flags may be set at the same time using an OR combination.

The bitmap for the **tp_flags** field is as follows:

rfu	rfu	units	ini	use	beg	end	level
7			bits			0	

The descriptions of each bit are listed below:

bit 0 (level): If set, the termination condition is level-sensitive.

Level-sensitive means that if the condition is satisfied when the function starts, termination will occur immediately.

Terminating conditions that can be level have a 'history' associated with them which records the state of the terminator before the function started. If this bit is not set, the termination condition is **edge-sensitive** and the function will not terminate unless the condition occurs after the function starts. The table below shows which terminating conditions can be edge-sensitive and which can be level-sensitive.terminationsedge-sensitive

NOTE: A level-sensitive termination condition only has to have occurred sometime in the history associated with that terminator to cause the function to terminate; the condition does not have to be present when the function starts in order to terminate the function.

Term. Condition	Level-sensitive	Edge-sensitive
DX_DIGTYPE	X	X
DX_MAXDTMF	X	X
DX_MAXSIL	X	X
DX_MAXNOSIL	X	X
DX_LCOFF	X	X
DX_DIGMASK	X	X
DX_IDDTIME	-	X
DX_MAXTIME	-	X
DX_PMON/OFF	-	X
DX_TONE	X	X

bit 1 (end): If set, the history of this terminator will be cleared when the function terminates. This bit has special meaning for DX_IDDTIME. If set, the terminator will be started after the first digit is received. Otherwise it will be started as soon as the function is started.

bit 2 (beg): If set, the history of this terminator will be cleared when the function starts. This bit will override the level bit (bit 0). If both are set, the history will be cleared and no past history of this terminator will be taken into account.

bit 3 (use): If this bit is set, the terminator will be used for termination. If the bit is not set, the history for the terminator will be cleared (depending on bits 1 & 2), but the terminator will still not be used for termination. This bit is not valid for the following terminating conditions:

- DX_MAXTIME
- DX_IDDTIME
- DX_DIGMASK
- DX_PMOFF
- DX_PMON

bit 4 (ini): This bit is only used for DX_MAXSIL termination. If the termination is edge-sensitive and this bit is set, the **tp_data** field should contain an initial length of silence to terminate upon if silence is detected before non-silence. In general, the initial length of silence to terminate on in **tp_data** should be greater than the value in **tp_length**.

If the termination is level sensitive then this bit must be set to 0 and tp_length will be used for the termination.

bit 5 (units): If set the units of time will be in 10 ms. The default is 100 ms units.

NOTES: 1. The 10 ms timer resolution is only available with version 0.62 or later of the D/4x firmware.

2. **tp_flags** is used differently for DX_PMON. See Using DX_PMOFF and DX_PMON at the end of this Appendix for information.

tp_data

tp_data specifies optional additional data. This bit can be set as follows:

Table 16. tp_data Valid Values

Value in tp_termno	tp_data Entry
DX_MAXSIL	initial length of silence to terminate on
DX_PMOFF	maximum time of silence off
DX_PMON	maximum time of silence on DX_TONEON - terminate after a "tone-on" event DX_TONEOFF - terminate after a "tone-off" event

tp_nextp

tp_nextp contains a pointer to the next DV_TPT structure in a linked list. The **tp_type** field must be set to IO_LINK in this case.

The table that follows indicates how DV_TPT fields should be filled.

NOTE: An asterisk indicates the default **tp_flags** setting, defined when **tp_flags** is set to TF_(term name), where (term name) is the suffix of the **tp_termno** setting, as in DX_(term name). To override defaults, set the bits in **tp_flags** individually, as required.

Table 17. DV_TPT Fields Settings Summary

NOTE: The **tp_flags** column describes the effect of the field when set to one and not set to one. "*" indicates the default value for each bit. The defaults defines for the **tp_flags** field are listed in the **tp_flags** description in this Appendix.

tp_termno	tp_type	tp_length	tp_flags: not set	tp_flags: set	tp_data	tp_nextp
DX_MAXDTMF	IO_LINK IO_EOT IO_CONT	max number of digits	bit 0: TF_EDGE bit 1: no clr* bit 2: no clr* bit 3: clr hist	TF_LEVEL* TF_CLREND TF_CLRBEG TF_USE*	N/A	pointer to next DV_TPT if linked list
DX_MAXSIL	IO_LINK IO_EOT IO_CONT	max length silence	bit 0: bit 1: no clr* bit 2: no clr* bit 3: clr hist bit 4: no-setinit bit 5: 100ms*	TF_EDGE* TF_LEVEL TF_CLREND TF_CLRBEG TF_USE* TF_SETINIT TF_10MS	length of init silence	pointer to next DV_TPT in linked list
DX_MAXNOSIL	IO_LINK IO_EOT IO_CONT	max length non-silence	bit 0: TF_EDGE* bit 1: no clr* bit 1: no clr* bit 2: no clr* bit 3: clr hist bit 4: N/A bit 5: 100ms*	TF_LEVEL TF_CLREND TF_CLRBEG TF_USE* N/A TF_10MS	N/A	pointer to next DV_TPT if linked list
DX_LCOFF	IO_LINK IO_EOT IO_CONT	max length loop current drop	bit 0: TF_EDGE bit 1: no clr bit 2: no clr* bit 3: clr hist bit 4: N/A bit 5: 100ms*	TF_LEVEL* TF_CLREND* TF_CLRBEG TF_USE* N/A TF_10MS	N/A	pointer to next DV_TPT if linked list
DX_IDDTIME	IO_LINK IO_EOT IO_CONT	max length interdigit delay	bit 0: TF_EDGE* bit 1: strt@call* bit 2: N/A bit 3: N/A bit 4: N/A bit 5: 100ms*	N/A strt@1st N/A N/A N/A TF_10MS	N/A	pointer to next DV_TPT if linked list
DX_MAXTIME	IO_LINK IO_EOT IO_CONT	max length function time	bit 0: TF_EDGE* bit 1: N/A bit 2: N/A bit 3: N/A bit 4: N/A bit 5: 100ms*	N/A N/A N/A N/A N/A TF_10MS	N/A	pointer to next DV_TPT if linked list
DX_DIGMASK	IO_LINK	bit 0: d (set)	bit 0:	TF_LEVEL*	N/A	pointer to

	IO_EOT IO_CONT	bit 1: 1 bit 2: 2 bit 3: 3 bit 4: 4 bit 5: 5 bit 6: 6 bit 7: 7 bit 8: 8 bit 9: 9 bit 10: 0 bit 11: * bit 12: # bit 13: a bit 14: b bit 15: c	TF_EDGE			next DV_TPT if linked list
DX_PM-+OFF	IO_LINK IO_EOT IO_CONT	number of pattern repetitions	minimum time silence off		max time silence off	pointer to next DV_TPT iflinked list
DX_PMON	IO_LINK IO_EOT IO_CONT	bit 0: TF_EDGE*/ TF_LEVEL bit 1: N/A bit 2: N/A bit 3: N/A bit 4: N/A bit 5: 100ms/ TF_10MS	maximum time silence on	max time silence on	pointer to next DV_TPT if linked list	
DX_TONE	IO_LINK IO_EOT IO_CONT	Tone ID	bit 0: TF_EDGE bit 1: no clr bit 2: no clr* bit 3: clr hist	TF_LEVEL* TF_CRLREND* TF_CLRBEG TF_USE*	DX_ TONEON DX_ TONEOFF	pointer to next DV_TPT if linked list
DX_DIGTYPE	IO_LINK IO_EOT IO_CONT	low byte:ASCII val. *hi byte:digit type	bit 0: TF_EDGE	TF_LEVEL	N/A	pointer to next DV_TPT if linked list

*The **tp_flags** column describes the effect of the field when set to one and not set to one. "*" indicates the default value for each bit. The defaults defines for the **tp_flags** field are listed in the **tp_flags** description in this Appendix.

Since DX_PMON requires that the **tp_length** field is set with **tp_flags** values, the previous statements apply to **tp_length** for DX_PMON.

Using DX_PMOFF and DX_PMON

The DX_PMOFF and DX_PMON termination conditions must be used in tandem. In other words, 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. When used, both must be specified together or else an error will result in the execution of the function.

In the first block, **tp_termno** is set to DX_PMOFF. The **tp_length** holds the number of patterns before termination. **tp_flags** holds the minimum time for silence off while tp_data holds the maximum time for silence off. In the next DV_TPT structure, **tp_termno** is DX_PMON, and the **tp_length** field holds the flag bit mask as shown above. Only the "units" bit is valid; all other bits must be 0. The **tp_flags** field holds the minimum time for silence on, while tp_data holds the maximum time for silence on. An example of this would be:

DV_TPT Example

```
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
DV_TPT   tpt[2];


/*
 * detect a pattern which repeats 4 times of approximately 2 seconds
 * off 2 seconds on.
 */
tpt[0].tp_type   = IO_CONT;  /* next entry is contiguous */
tpt[0].tp_termno = DX_PMOFF; /* specify pattern match off */
tpt[0].tp_length = 4;        /* terminate if pattern repeats 4 times */
tpt[0].tp_flags  = 175;      /* minimum silence off is 1.75 seconds
                              * (10 ms units) */
tpt[0].tp_data   = 225;      /* maximum silence off is 2.25 seconds
                              * (10 ms units) */
tpt[1].tp_type   = IO_EOT;   /* This is the last in the chain */
tpt[1].tp_termno = DX_PMON;  /* specify pattern match on */
tpt[1].tp_length = TF_10MS;  /* use 10 ms timer units */
tpt[1].tp_flags  = 175;      /* minimum silence on is 1.75 seconds
                              * (10 ms units) */
tpt[1].tp_data   = 225;      /* maximum silence on is 2.25 seconds
                              * (10 ms units) */
/* issue the function */
```

Appendix B
Error Defines

Errors - *Voice Library*

This appendix lists the error defines that may be returned for the Voice Library functions.

For error codes returned for SCbus functions and a description of the error, refer to the *SCbus Routing Function Reference for Windows NT*.

The following table contains the list of errors that can be returned using **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** functions

Table 18. Voice Library Function Errors

Error Define	Error String
EDX_AMPLGEN	Invalid Amplitude Value in Tone Generation Template
EDX_ASCII	Invalid ASCII Value in Tone Template Description
EDX_BADDEV	Device Descriptor error
EDX_BADIOTT	*DX_IOTT* structure error
EDX_BADPARM	Parameter error
EDX_BADPROD	Function Not Supported on this Board
EDX_BADTPT	DX_TPT structure error
EDX_BUSY	Device or channel is Busy
EDX_CADENCE	Invalid Cadence Component Values in Tone Template Description
EDX_CHANNUM	Invalid Channel Number Specified
EDX_DIGTYPE	Invalid Dig_type Value in Tone Template Description
EDX_FLAGGEN	Invalid tn_dflag field in Tone Generation Template
EDX_FREQDET	Invalid Frequency Component Values in Tone Template Description
EDX_FREQGEN	Invalid Frequency Component in Tone

	Generation Template
EDX_FWERROR	Firmware Error
EDX_IDLE	Device is Idle
EDX_INVSUBCMD	Invalid sub-command number
EDX_MAXTMPLT	Max number of Tone Templates Exist or user-defined tones for the board[from r2_creatfsig()]
EDX_MSGSTATUS	Invalid Message Status Setting
EDX_NOERROR	No Error
EDX_NONZEROSIZE	Reset to Default was Requested but size was non-zero
EDX_SPDVOL	Must Specify either SV_SPEEDTBL or SV_VOLUMETBL
EDX_SVADJBLKS	Invalid Number of Speed/Volume Adjustment Blocks
EDX_SVMTRANGE	An Entry in SV_SVMT was out of Range
EDX_SVMTSIZE	Invalid Table Size Specified
EDX_SYSTEM	Windows NT System Error; check the global variable errno for more information
EDX_TIMEOUT	I/O Function Timed Out
EDX_TONEID	Invalid Tone Template ID

Appendix C

DTMF and MF Tone Specifications

The following two charts show the tone specifications for MF and DTMF tones.

MF Tone Specifications (CCITT R1 Tone Plan)

Code	Tone Pair Frequencies (Hz)	Default Length (ms)	Name
1	700, 900	60	1
2	700, 1100	60	2
3	900, 1100	60	3
4	700, 1300	60	4
5	900, 1300	60	5
6	1100, 1300	60	6
7	700, 1500	60	7
8	900, 1500	60	8
9	1100, 1500	60	9
0	1300, 1500	60	0
*	1100, 1700	100	KP
#	1500, 1700	60	ST
a	900, 1700	60	ST1
b	1300, 1700	60	ST2
c	700, 1700	60	ST3

* The standard length of a KP tone is 100 ms.

DTMF Tone Specifications

Code	Tone Pair Frequencies (Hz)	Default Length (ms)
1	697, 1209	100
2	697, 1336	100
3	697, 1477	100
4	770, 1209	100
5	770, 1336	100
6	770, 1477	100
7	852, 1209	100
8	852, 1336	100
9	852, 1477	100
0	941, 1336	100
*	941, 1209	100
#	941, 1477	100
a	697, 1633	100
b	770, 1633	100
c	852, 1633	100
d	941, 1633	100

Using MF Detection

Some MF digits use approximately the same frequencies as DTMF digits (see above charts). Because there is a frequency overlap, if you have the incorrect kind of detection enabled, MF digits may be mistaken for DTMF digits, and vice versa. To ensure that digits are correctly detected, only one kind of detection should be enabled at any time.

Digit detection accuracy depends on two things:

- which digit is sent
- the kind of detection enabled when the digit is detected

The two tables that follow show the digits that are detected when each type of detection is enabled. Table 19 shows which digits are detected when MF digits are sent. Table 20 shows which digits are detected when DTMF digits are sent.

Table 19. Detecting MF Digits

String Received			
MF Digit Sent	Only MF Detection Enabled	Only DTMF Detection Enabled	MF and DTMF Detection Enabled
1	1		1
2	2		2
3	3		3
4	4	2*	4,2*
5	5		5
6	6		6
7	7	3*	7,3*
8	8		8
9	9		9
0	0		0
*	*		*
#	#		#
a	a		a
b	b		b
c	c		c

* = detection error

Table 20. Detecting DTMF Digits

String Received			
DTMF Digit Sent	Only DTMF Detection Enabled	Only MF Detection Enabled	DTMF & MF Detection Enabled
1	1		1
2	2	4*	4,2*
3	3	7*	7,3*
4	4		4
5	5	4*	4,5*
6	6	7*	7,6*
7	7		7
8	8	5*	5,8*
9	9	8*	8,9*
0	0	5*	5,0*
*	*		*
#	#	8*	8,#*
a	a	c*	c,a*
b	b	c*	c,b*
c	c	a*	a,c*
d	d	a*	a,d*

* = detection error

Appendix D
Related Voice Publications

For more information on Voice hardware and software products see the following Dialogic publications:

- For information about installing Voice software, see the *System Release Software Installation Reference for Windows NT*.
- For information about the Standard Runtime Library, see the *Standard Runtime Library Programmer's Guide for Windows NT*.
- For information about the SCbus, see the *SCbus Routing Guide* and the *SCbus Routing Function Reference for Windows NT*.
- For information about the D/2x, D/4x, D/81A, D/12x, and D/xxxSC (D/160SC-LS, D/240SC, D/240SC-T1, D/300SC-E1 and D/320SC) voice boards, see the *Quick Install Cards* shipped with the boards.
- For information about the digital interface device functions for the D/240SC-T1 and D/300SC-E1 boards, see the *Digital Network Interface Software Reference for Windows NT*.
- For information about the primary rate functions, see the *Primary Rate Software Reference for Windows NT*.

Glossary

Mu-law: (1) A pulse-code modulation (PCM) algorithm used in digitizing telephone audio signals in T-1 areas. (2) The PCM coding and compounding standard used in Japan and North America.

A-LAW: Pulse Code Modulation (PCM) algorithm used in digitizing telephone audio signals in E-1 areas.

Adaptive Differential Pulse Code Modulation: See ADPCM.

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

ADSI: Analog Display Services Interface. A Bellcore standard defining a protocol on the flow of information between a switch, a server, a voice mail system, a service bureau, or a similar device and a subscriber's telephone, PC, data terminal, or other communicating device with a screen. The idea of ADSI is to add words to a system that usually only uses touch tones. In a typical voice mail system, you call up and hear choices: "to listen to new messages, press 1, to hear saved messages, press 2," etc. ADSI is designed to display the choices you're hearing on a screen attached to your phone. Your response is the same: a touch tone button. ADSI's signaling is DTMF and standard Bell 202 modem signals from the service to your 202-modem equipped phone. From the phone to the service it's only touch tone. ADSI works on every phone line in the world.

AGC: Automatic Gain Control. An electronic circuit used to maintain the audio signal volume at a constant level.

AMIS: Audio Messaging Interchange Specification. A series of standards aimed at addressing the problem of how voice messaging systems produced by different vendors can network or inter-network. It deals specifically with the interaction of the systems and does not affect the systems themselves. There are two specifications: 1. AMIS-digital: All the control information and the voice messages are ported between systems digitally. 2. AMIS-analog: Control information and messages are transferred in analog form. For AMIS specifications, call Hartfield Associates (Boulder, CO) at (303) 442-5395.

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

Analog Expansion Bus (AEB): Analog electrical connection (bus) between Dialogic network interface modules and analog resource modules. The AEB interfaces network boards and voice boards, which fit in the AT-expansion slot of a PC. See Also PEB, SCSA

ANI: Automatic Number Identification.

Antares: A Dialogic open platform for easily incorporating speech recognition, Text-To-Speech, fax and many other DSP technologies. Dialogic PC-based expansion board with fourTI floating point DSPs, SPOX DSP operating system, and the Antares board downloadable firmware and device driver.

API: See Application Programming Interface

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

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

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

AT: Used to describe an IBM or IBM-compatible Personal Computer (PC) containing an 80286 or higher microprocessor, a 16-bit bus architecture, and a compatible BIOS.

AT bus: The common communication channel in a PC AT. The channel uses a 16-bit data path architecture, which allows up to 16 bits of data transfer. This bus architecture includes the standard PC bus plus a set of 36 lines for additional data transmission, addressing, and interrupt request handling.

Automatic Gain Control: See AGC.

base memory address: A starting memory location (address) from which other addresses are referenced.

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

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

board device: A board-level object that can be manipulated by a physical library. Board devices can be real physical devices, such as a D/4x board, or emulated devices, such as one of the D/4x boards that is emulated by a D/81A, D/12x or D/xxxSC board.

Board Locator Technology: Operates in conjunction with a rotary switch to determine and set non-conflicting slot and IRQ interrupt-level parameters, thus eliminating the need to set confusing jumpers or DIP switches.

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

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

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

cadence: A rhythmic sequence or pattern. Once 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 establish parameters.

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

Call Progress Analysis: The process used to automatically determine what happened after an outgoing call is dialed. Also referred to as call analysis or call progress

Call Status Transition Event Functions: Functions that set and monitor events on devices.

CCITT: Comite Consultatif Internationale de Telegraphique et Telephonique. One of the four permanent parts of the International Telecommunications Union, a United Nations agency based in Geneva. The CCITT is divided into three sections: 1. Study Groups set up standards for telecommunications equipment, systems, networks, and services. 2. Plan Committees develop

general plans for the evolution of networks and services. 3. Specialized Autonomous Groups produce handbooks, strategies, and case studies to support developing countries.

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

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

CO: Central Office. A local phone exchange. In general, "CO' refers to the phone network exchange that provides your phone lines. The term "Central Office" is used in North America. The rest of the world calls it PTT, for Post, Telephone and Telegraph. The telephone company facility where subscriber lines are linked, through switches, to other subscriber lines (including local and long distance lines).

computer telephony: The extension of computer-based intelligence and processing over the telephone network to a telephone. Lets you interact with computer databases or applications from a telephone and also enables computer-based applications to access the telephone network. Computer telephony makes computer-based information readily available over the world-wide telephone network from your telephone. Computer telephony technology incorporated into PCs supports applications such as: automatic call processing; automatic speech recognition; text-to-speech conversion for information-on-demand; call switching and conferencing; unified messaging that lets you access or transmit voice, fax, and E-mail messages from a single point; voice mail and voice messaging; fax systems including fax broadcasting, fax mailboxes, fax-on-demand, and fax gateways; transaction processing such as Audiotex and Pay-Per-Call information systems; call centers handling a large number of agents or telephone operators for processing requests for products, services or information; etc.

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

Configuration Functions: Functions that alter the configuration of devices.

Convenience Functions: Functions that simplify application writing.

D/81A: 8 port DSP-based voice board that runs SpringWare firmware. Connects via PEB to a standalone telephone network interface board.

D/120: A 12-channel voice board from Dialogic that consists of a SpringBoard-based expansion device and downloaded software. On the PEB bus, the D/120 serves as a resource module to the installed network module.

D/121: A 12-channel voice-store-and-forward product from Dialogic with all the features of the D/120 plus patented call analysis algorithms for outbound applications and multifrequency (MF) tone capability.

D/12x System: A Voice System that uses D/12x boards. See Voice System.

D/121A: A 12-channel voice board from Dialogic with all the features of the D/121 plus additional RAM, increased performance and reliability, and improved downstream compatibility.

D/121B: 12 port DSP-based voice board that runs SpringWare firmware. Connects via PEB to a standalone telephone network interface board.

D/12x: Any model of the Dialogic series of 12-channel voice-store-and-forward expansion boards for the AT-bus architecture. Includes: D/120 and D/121 boards.

D/160SC-LS: 16 port DSP-based voice board that runs SpringWare firmware and has onboard analog loop start telephone interfaces and an SCbus interface.

D/21D, D/41D: 2 and 4 port DSP-based voice boards with onboard analog telephone interface; runs SpringWare downloadable firmware.

D/21E, D/41E: 2 and 4 port DSP-based voice boards with onboard analog telephone interface; runs SpringWare downloadable firmware.

D/2x: A term used to refer to any 2-channel voice-store-and-forward expansion board made by Dialogic.

D/40: A model of 4-channel voice-store-and-forward expansion board by Dialogic with an on-board processor and shared RAM. The D/40 features real-time digitization, compression and playback of audio, DTMF reception, automatic answering, DTMF or rotary pulse dialing, and direct connection to telephone lines.

D/41: A model of the four-channel voice-store-and-forward expansion boards by Dialogic that has all of the features of a D/40 plus patented call analysis algorithms for outbound applications.

D/4x: Any model of the Dialogic series of 4-channel voice-store-and-forward expansion boards for the AT-bus architecture. Includes D/4xD and D/4xE boards.

D/240SC: 24 port DSP-based voice board that runs SpringWare firmware and has an onboard SCbus interface. Connects to a standalone telephone network interface board.

D/240SC-T1: 24 port DSP-based voice board that runs SpringWare firmware and has an onboard digital T-1 telephone interface and an SCbus interface.

D/300SC-E1: 30 port DSP-based voice board that runs SpringWare firmware and has an onboard digital E-1 telephone interface and an SCbus interface.

D/320SC: 30 port DSP-based voice board that runs SpringWare firmware and has an onboard SCbus interface. Connects to a standalone telephone network interface board.

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

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

deglitching: Eliminating false signal detection by filtering out rapid signal changes. Any signal change shorter than that specified by the deglitching parameters is ignored.

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

device channel: A Dialogic voice data path that processes one incoming or outgoing call at a time (equivalent to the terminal equipment terminating a phone line). There are 4 device channels on a D/4x, 12 on a D/12x, 16 on a

D/160SC-LS, 24 on a D/240SC or D/240SC-T1, 30 on a D/300SC-E1, and 32 on a D/320SC board.

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

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

Device Management Functions: Functions that open and close devices.

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

DIALOG/HD Series: Dialogic High Density products, including the D/160SC-LS, D/240SC, D/240SC-T1, D/300SC-E1, and D/320SC, provide a powerful set of advanced computer telephony features that developers can use to create cost-efficient, high-density systems.

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

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

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

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

DSP: 1. Digital signal processor. A specialized microprocessor designed to perform speedy and complex operations with digital signals. 2. Digital signal processing.

DTI/: (Digital Telephony Interface) The naming convention used with Dialogic boards such as the DTI/211. This interface is designed to work with the T-1 telephony standard used in North American and Japanese markets. A general term used to refer to any Dialogic digital telephony interface device.

DTI/124: A model of Dialogic's digital telephony interface device designed for use with the T-1 telephony standard used in North American and Japanese markets. This model connects to D/4x devices.

DTI/211: 24 port standalone telephone network interface for use with voice-only boards; digital T-1 interface.

DTI/212: 24 port standalone telephone network interface for use with voice-only boards; digital E-1 interface.

DTI/2xx: Refer's to Dialogic's DTI/211 or DTI/212 digital telephony interface boards.

DTI/xxx: Refers to any of Dialogic's second-generation digital telephony interface boards.

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

1: 697,1209	2: 697,1336	3:697,1477
4: 770,1209	5: 770,1336	6: 770,1477
7: 852,1209	8: 852,1336	9: 852,1477
0: 941,1336	*: 941,1209	#: 941,1477

E-1: A CEPT digital telephony format devised by the CCITT. A digital transmission channel that carries data at the rate of 2.048 Mbps (DS-1 level).

emulated device: A virtual device whose software interface mimics the interface of a particular physical device, such as a D/4x boards that is emulated by a D/12x or a D/xxxSC board. On a functional level, a D/12x board is perceived by an application as three D/4x boards. See physical device.

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

event handler: A portion of a Dialogic application program designed to trap and control processing of device-specific events. The rules for creating a DTI/1xx event handler are the same as those for creating a Windows NT signal handler.

Event Management functions: Class of device-independent functions (contained in the Standard Runtime Library) that connect events to

application-specified event handlers, allowing users to retrieve and handle events that occur on the device. See Standard Runtime Library.

Extended Attribute functions: Class of functions that take one input parameter (a valid Dialogic device handle) and return device-specific information. For instance, a Voice device's Extended Attribute function returns information specific to the Voice devices. Extended Attribute function names are case-sensitive and must be in capital letters. See Standard Runtime Library.

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

flash: A signal which consists of a momentary on-hook condition used by the Voice hardware to alert a telephone switch. This signal usually initiates a call transfer.

frequency detection: A voice driver feature that detects the tri-tone Special Information Tone (SIT) sequences and other single-frequency tones for call progress analysis.

Global Tone Detection: A feature that allows the creation and detection of user-defined tone descriptions on a channel by channel basis.

hook state: A general term for the current line status of the channel: either on-hook or off-hook. A telephone station is said to be on-hook when the conductor loop between the station and the switch is open and no current is flowing. When the loop is closed and current is flowing the station is off-hook. These terms are derived from the position of the old fashioned telephone set receiver in relation to the mounting hook provided for it.

hook switch: The name given to the circuitry which controls on-hook and off-hook state of the Voice device telephone interface.

I/O Functions: Functions that transfer data to and from devices.

I/O: Input-Output

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

in-band: Refers to the use of robbed-bit signaling (T-1 systems only) on the network or PEB. "In-band" refers to the fact that the signaling for a particular channel or time slot is carried within the voice samples for that time slot, thus within the 64 kbps (kilobits per second) voice bandwidth.

in-band signaling: (1) In an analog telephony circuit, in-band refers to signaling that occupies the same transmission path and frequency band used to transmit

voice tones. (2) In digital telephony, "in-band" means signaling transmitted within an 8-bit voice sample or time slot, as in T-1 "robbed-bit" signaling. (3) On the Dialogic PCM Expansion Bus (PEB), signaling is considered "in-band" only if it occupies the same transmission path and frequency band used to transmit voice data.

interrupt request level: A signal sent to the central processing unit (CPU) to temporarily suspend normal processing and transfer control to an interrupt handling routine. Interrupts may be generated by conditions such as completion of an I/O process, detection of hardware failure, power failures, etc.

IRQ: Interrupt ReQuest. A signal sent to the CPU to temporarily suspend normal processing and transfer control to an interrupt handling routine. A means of toggling between applications so that your system does not crash.

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

loop: The physical circuit between the telephone switch and the D/xxx board.

loop current: The current that flows through the circuit from the telephone switch when the Voice device is off-hook.

loop current detection: A voice driver feature that returns a connect after detecting a loop current drop.

loop start: In an analog environment, an electrical circuit consisting of two wires (or leads) called tip and ring, which are the two conductors of a telephone cable pair. The CO provides voltage (called "talk battery" or just "battery") to power the line. When the circuit is complete, this voltage produces a current called loop current. The circuit provides a method of starting (seizing) a telephone line or trunk by sending a supervisory signal (going off-hook) to the CO.

LSI/120: A Dialogic 12-line loop start interface expansion board.

off-hook: The state of a telephone station when the conductor loop between the station and the switch is closed and current is flowing. When a telephone handset is lifted from its cradle (or equivalent condition), the telephone line state is said to be off-hook.

on-hook: When a telephone handset is returned to its cradle (or equivalent condition), the telephone line state is said to be on-hook.

PC: Personal Computer. In this manual, the term refers to an IBM Personal Computer or compatible machine.

PCM Expansion Bus: See PEB.

PEB: PCM Expansion Bus. A Dialogic open platform, digital voice bus for electrically and digitally connecting different voice processing components. Information on the PEB is encoded using the Pulse Code Modulation (PCM) method. Non-Dialogic products using PCM encoding may interface with Dialogic products by using this bus.

PerfectDigit: Dialogic SpringWare DTMF or MF signaling.

PerfectLevel: Dialogic SpringWare Volume control

PerfectPitch: Dialogic SpringWare Speed control

PerfectVoice: Dialogic SpringWare Enhanced voice coding

physical device: A device that is an actual piece of hardware, such as a D/4x board; not an emulated device. See emulated device.

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

polling functions: Voice library functions check the current status of a voice device. Polling functions are also used to examine the number and configuration of devices in the system and to detect when events occur on a device.

Pulse Code Modulation: PCM. A sophisticated technique for reducing voice data storage requirements that is used by Dialogic in the DSP voice boards. Dialogic supports either m-law Pulse Code Modulation, which is used in North America and Japan, or A-law Pulse Code Modulation, which is used in the rest of the world.

resource: Functionality (e.g. voice-store-and-forward) that can be assigned to call. Resources are shared when functionality is selectively assigned to a call (usually via a PEB time slot) and may be shared among multiple calls. Resources are dedicated when functionality is fixed to the one call.

RFU: Reserved for future use.

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

route: Assign a resource to a time slot.

robbed-bit signaling: The type of signaling protocol implemented in areas using the T-1 telephony standard. In robbed-bit signaling, signaling information is carried in-band, within the 8-bit voice samples. These bits are later stripped away, or "robbed," to produce the signaling information for each of the 24 time slots.

routing functions: For SCbus, functions that assign analog and digital channels to specific SCbus time slots; these SCbus time slots can then be connected to transmit or listen to other SCbus time slots. For PEB, functions that change the routing of channels to the time slots on the PCM Expansion Bus (PEB).

sampling rate: Frequency with which a digitizer takes measurements of the analog voice signal.

SCbus: Signal Computing Bus. Third generation TDM (Time Division Multiplexed) resource sharing bus that allows information to be transmitted and received among resources over multiple data lines.

SCSA: See Signal Computing System Architecture.

Signal Computer System Architecture: SCSA. A Dialogic standard open development platform. An open hardware and software standard that incorporates virtually every other standard in PC-based switching. All signaling is out of band. In addition, SCSA offers time slot bundling and allows for scalability.

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

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

solicited event: An expected event. It is specified using one of the device library's asynchronous functions. For example, for dx_play(), the solicited event is "play complete."

Special Information Tones: SIT. (1) Standard Information Tones. Tones sent out by a central office to indicate that the dialed call has been answered by the

distant phone. (2) Special Information Tone. Detection of a SIT sequence indicates an operator intercept or other problem in completing the call.

speed and volume control: Voice software that contains functions and data structures to control the speed and volume of play on a channel. The end user controls the speed or volume of a message by entering a DTMF tone.

speed and volume modification table: Each channel on a voice board has a table with twenty entries that allow for a maximum of ten increases and decreases in speed or volume, and one "origin" entry that represents regular speed or volume.

SpringBoard: A Dialogic expansion board using digital signal processing to emulate the functions of other products. SpringBoard is a development platform for Dialogic products such as the D/120 and D/121. The SpringBoard-MC is a development platform for Dialogic Micro Channel products such as the D/81-MC.

SpringBoard functions: Functions used on SpringBoard devices only.

SpringWare: Software algorithms build into the downloadable firmware that provides the voice processing features available on all Dialogic voice boards.

SRL: See Standard Runtime Library.

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

Standard Runtime Library: A Dialogic software resource containing Event-Management and Standard Attribute functions and data structures used by all Dialogic devices, but which return data unique to the device. See the Standard Runtime Library Programmer’s Guide for Windows NT.

string: An array of ASCII characters.

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

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

System Release Development Package: The software and user documentation provided by Dialogic that is required to develop applications.

T-1: The digital telephony format used in North America. In T-1, 24 voice conversations are time-division multiplexed into a single digital data stream containing 24 time slots, and signaling data are carried "in-band." Since all available time slots are used for conversations, signaling bits are substituted for voice bits in certain frames. Hardware at the receiving end must use the robbed-bit" technique for extracting signaling information. T-1 carries data at the rate of 1.544 Mbps (DS-1 level).

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

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

time slot: In a digital telephony environment, a normally continuous and individual communication (for example, someone speaking on a telephone) is (1) digitized, (2) broken up into pieces consisting of a fixed number of bits, (3) combined with pieces of other individual communications in a regularly repeating, timed sequence (multiplexed), and (4) transmitted serially over a single telephone line. The process happens at such a fast rate that, once the pieces are sorted out and put back together again at the receiving end, the speech is normal and continuous. Each individual pieced-together communication is called a time slot.

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

transparent signaling: The mode in which a network interface device accepts signaling data from a resource device transparently, or without modification. In transparent signaling, outgoing T-1 signaling bits are generated by a PEB or SCbus resource device. In effect the resource device performs signaling to the network.

Universal Dialogic Diagnostic program: Software diagnostic routines for testing board-level functions of Dialogic hardware.

voice processing: Science of converting human voice into data that can be reconstructed and played back at a later time. Dialogic equipment can place 2-30 ports in one PC slot. They also use common API's for scalability and the SCbus to connect to a broad range of technologies.

Voice System: A combination of expansion boards and software that let you develop and run high-density voice processing applications.

wink: In T-1 or E-1 systems, a signaling bit transition from on to off, or off to on, and back again to the original state. In T-1 systems, the wink signal can be transmitted on either the A or B signaling bit. In E-1 systems, the wink signal can be transmitted on either the A, B, C, or D signaling bit. Using either system, the choice of signaling bit and wink polarity (on-off-on or off-on-off hook) is configurable through DTI/2xx board download parameters.

Index

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Z

Dialogic Sales Offices

North American Sales
1-800-755-4444 or 201-993-3030
fax: 201-631-9631

Corporate Headquarters
1515 Route 10
Parsippany, NJ 07054-4596
USA
201-993-3000
fax: 201-993-3093

Northeastern US
70 Walnut Street
Wellesley, MA 02181

Southeastern US
1040 Crown Pointe Pkwy.
Suite 360
Atlanta, GA 30338

North Central US
1901 North Roselle Road
Suite 800
Schaumburg, IL 60195

South Central US
3307 Northland Drive
Suite 300
Austin, TX 78731

Western US
1314 Chesapeake Terrace
Sunnyvale, CA 94089

Northwestern US
19125 North Creek Parkway #120
Bothell, WA 98011

GammaLink Division
1314 Chesapeake Terrace
Sunnyvale, CA 94089

Computer-Telephone Division
100 Unicorn Park Drive
Woburn, MA 01801

Spectron Microsystems Division
315 Bollay Drive
Santa Barbara, CA 93117
805-968-5100
fax: 805-968-9770

Dialogic On-Line Information Retrieval System (fax-on-demand)
1-800-755-5599 or 201-993-1063

GammaLink Fax-on-Demand
408-734-9906

computer telephony BBS
(ctBBS)
201-993-0864

CTI@Dialogic WWW Site
http://www.dialogic.com

Dialogic Sales Internet
sales@dialogic.com

Canada
Dialogic Corporation
1033 Oak Meadow Road
Oakville, Ontario
L6M 1J6
Canada

Latin America & Caribbean
Dialogic Latin America and Caribbean
Av. R. S. Peña
730 3º Piso
Oficina 34
(1035) Buenos Aires, Argentina
541-328-1531 or -9943
fax: 541-328-5425

European Headquarters (serving Europe, Middle East, & Africa)
Dialogic Telecom Europe N.V.-S.A.
Airway Park
Lozenberg 23
Building D, 3rd floor
B-1932 Sint Stevens Woluwe
Belgium
32-2-712-4311
fax : 32-2-712-4300

DTE On-Line Information Retrieval System
32-2-712-4322

DTE BBS
32-2-725-7846

Dialogic Sales Offices

Germany, Switzerland, & Austria
Dialogic Telecom Deutschland GmbH
Industriestrasse 1
D82110 Munich
Germany
49-89-894-362-0
fax: 49-89-894-362-77

France
Dialogic Telecom France S.r.l.
42, Avenue Montaigne
F-75008
Paris
France
33-1-53-67-52-80
fax: 33-1-53-67-52-79

Italy
Dialogic Telecom Italy S.r.l.
Strada Pavese, 1/3
I-20089 Rozzano
Milan
Italy
39-2-575-54302
fax: 39-2-575-54310

United Kingdom, Ireland, & Scandinavian Countries
Dialogic Telecom U.K. Ltd.
Dialogic House
Dairy Walk
Hartley Wintney
Hampshire
RG27 8XX
United Kingdom
44-1252-844000
fax: 44-1252-844525

People's Republic of China, Hong Kong, Macau, and Taiwan
Dialogic Beijing Representative Office
8 North Dongsanhuan Road
Landmark Building, Suite 1308
Chaoyang District
Beijing 100004
People's Republic of China
86-10-504-5364
fax: 86-10-506-7989

Japan & Korea
Dialogic Systems K.K.
Suntowers Center
Building 18F
2-11-22 Sangenjaya
Setagayaku, Tokyo 154
Japan
81-3-5430-3252
fax: 81-3-5430-3373

East Asia, Southeast Asia, West Asia, & Australia
Dialogic SEA Pte. Ltd.
150 Beach Road
#17-08 Gateway West Bldg.
Singapore 189720
65-298-8208
fax: 65-298-1820

DSEA BBS
65-291-3249

New Zealand
Dialogic (N.Z.) Ltd.
Level 6
Tower 2
Shortland Towers
55-63 Shortland Street
Auckland
New Zealand
64-9-366-1133
fax: 64-9-302-1793

NOTES

NOTES

NOTES