Contents

- dx_open( ) – open a voice device and return a unique device handle ................................................................. 163
- dx_listen( ) – connect a voice listen channel to TDM bus time slot ................................................................. 160
- dx_OpenStreamBuffer( ) – create and initialize a circular stream buffer ......................................................... 166
- dx_play( ) – play recorded voice data .................................................................................................................. 168
- dx_PlayStreamData( ) – play back recorded voice data from multiple sources .................................................. 178
- dx_playtone( ) – play tone defined by TN_GEN structure ..................................................................................... 182
- dx_playtoneEx( ) – play the cadenced tone defined by TN_GENCAD ................................................................. 186
- dx_playvox( ) – play voice data stored in a single VOX file ................................................................................ 190
- dx_playwav( ) – play voice data stored in a single WAVE file ............................................................................. 193
- dx_PutStreamData( ) – place data into a circular stream buffer ......................................................................... 196
- dx_querytone( ) – get tone information for a specific call progress tone ............................................................ 198
- dx_rec( ) – record voice data from a single channel ............................................................................................. 201
- dx_recf( ) – record voice data to a single file ..................................................................................................... 208

- dx_bldstdig( ) – define a user-defined single-frequency digit tone .................................................................... 70
- dx_createpote( ) – create a new tone definition for a specific call progress tone ................................................. 90
- dx_covertnum( ) – delete all user-defined tones .................................................................................................. 97
- dx_dial( ) – dial an ASCIIZ string ...................................................................................................................... 99
- dx_dialton( ) – disable detection of a user-defined tone ...................................................................................... 105
- dx_enbtone( ) – enable detection of a user-defined tone ...................................................................................... 108
- dx_deltones( ) – delete all user-defined tones ................................................................................................... 97
- dx_fileclose( ) – close a file ............................................................................................................................... 111
- dx_filerrno( ) – return the system error value ..................................................................................................... 113
- dx_fileopen( ) – open a file ................................................................................................................................ 116
- dx_fileread( ) – read data from a file .................................................................................................................. 118
- dx_fileseek( ) – move a file pointer .................................................................................................................... 121
- dx_filewrite( ) – write data from a buffer into a file ........................................................................................... 124
- dx_getctinfo( ) – get information about a voice device ....................................................................................... 127
- dx_getcursv( ) – return the specified current volume settings ......................................................................... 129
- dx_getdigbuf( ) – clear all digits in the firmware digit buffer ............................................................................ 132
- dx_getdllversion( ) – retrieve the voice DLL version number ............................................................................ 138
- dx_getevt( ) – monitor channel events synchronously ......................................................................................... 140
- dx_geteventlist( ) – retrieve feature support information for the device .............................................................. 143
- dx_getfeaturelist( ) – retrieve feature support information for the device .......................................................... 143
- dx_getctinfo( ) – get information about a voice device ....................................................................................... 127
- dx_getcursv( ) – return the specified current volume settings ....................................................................... 129
- dx_getdig( ) – collect digits from a channel digit buffer ...................................................................................... 132
- dx_getdllversion( ) – retrieve the voice DLL version number ............................................................................ 138
- dx_getenum( ) – return the current volume modification table ........................................................................ 152
- dx_getemitslot( ) – get TDM bus time slot number of voice transmit channel .................................................. 155
- dx_libinit( ) – initialize the voice library DLL .................................................................................................. 158
- dx_listens( ) – connect a voice listen channel to TDM bus time slot ................................................................. 160
- dx_open( ) – open a voice device and return a unique device handle ............................................................... 163
- dx_OpenStreamBuffer( ) – create and initialize a circular stream buffer ......................................................... 166
- dx_playiottdata( ) – play back recorded voice data from multiple sources ...................................................... 178
- dx_play( ) – play recorded voice data .................................................................................................................. 168
- dx_playf( ) – synchronously play voice data ....................................................................................................... 175
- dx_playiotdata( ) – play back recorded voice data from multiple sources ......................................................... 178
- dx_placetone( ) – play tone defined by TN_GEN structure .................................................................................. 182
- dx_playtoneEx( ) – play the cadenced tone defined by TN_GENCAD ............................................................... 186
- dx_playvox( ) – play voice data stored in a single VOX file ................................................................................. 190
- dx_playwav( ) – play voice data stored in a single WAVE file ............................................................................ 193
- dx_PutStreamData( ) – place data into a circular stream buffer ........................................................................ 196
- dx_querytone( ) – get tone information for a specific call progress tone ............................................................ 198
- dx_rec( ) – record voice data from a single channel ............................................................................................. 201
- dx_recf( ) – record voice data to a single file ..................................................................................................... 208
dx_reciottdata( ) – record voice data to multiple destinations ........................................... 212
dx_recvox( ) – record voice data to a single VOX file ..................................................... 216
dx_recwav( ) – record voice data to a single WAVE file .................................................... 219
dx_ResetStreamBuffer( ) – reset internal data for a circular stream buffer .................. 222
dx_setdevio( ) – install and retrieve user-defined I/O functions .................................. 224
dx_setdigtyp( ) – control the types of digits detected by the voice channel ........... 227
dx_setevtmsk( ) – enable detection of call status transition (CST) events ............ 230
dx_setgtdamp( ) – set up the tone detection amplitudes ............................................. 235
dx_setparm( ) – set physical parameters of a channel or board device ...................... 237
dx_setsvcond( ) – set conditions that adjust volume of play ..................................... 241
dx_setsvmt( ) – change default values of the volume modification table .............. 245
dx_setuio( ) – install user-defined I/O functions ......................................................... 249
dx_SetWaterMark( ) – set water mark for the circular stream buffer ....................... 252
dx_stopch( ) – force termination of currently active I/O functions ......................... 254
dx_unlisten( ) – disconnect voice receive channel from TDM bus .......................... 257
nr_scunroute( ) – break a full or half-duplex connection ........................................... 265
nr_scroute( ) – make a full or half-duplex connection ................................................. 259

3 Events .......................................................................................................................... 271
3.1 Overview of Events ................................................................................................. 271
3.2 Termination Events ............................................................................................... 271
3.3 Unsolicited Events ................................................................................................. 273
3.4 Call Status Transition (CST) Events ................................................................. 273

4 Data Structures .......................................................................................................... 275
CT_DEVINFO – channel/time slot device information ............................................... 276
DV_DIGIT – user digit buffer ...................................................................................... 278
DV_TPT – termination parameter table ...................................................................... 279
DX_CAP – call progress analysis parameters ............................................................. 285
DX_CST – call status transition (CST) information .................................................... 288
DX_EBLK – call status transition event block ............................................................. 289
DX_IOTT – input/output transfer table ....................................................................... 290
DX_STREAMSTAT – status of stream buffer ............................................................. 293
DX_SVCB – speed and volume adjustment condition block ..................................... 295
DX_SVMT – speed and volume modification tables .................................................... 298
DX_UIO – user-defined input/output ......................................................................... 300
DX_XPB – input/output transfer parameter block ..................................................... 301
FEATURE_TABLE – feature information ................................................................. 303
SC_TSINFO – TDM bus time slot information ............................................................. 306
TN_GEN – tone generation template ......................................................................... 307
TN_GENCAD – cadenced tone generation template .................................................... 308
TON_DATA – tone information .................................................................................. 310

5 Error Codes ................................................................................................................. 313

6 Supplementary Reference Information ................................................................... 317
6.1 DTMF and MF Tone Specifications ..................................................................... 317
6.2 DTMF and MF Detection Errors ......................................................................... 318
Contents

Glossary ................................................................. 321
Index ................................................................. 329
Tables

1  Valid Dial String Characters ............................................................. 101
2  System Error Values ........................................................................ 113
3  Play Mode Selections ....................................................................... 169
4  Record Mode Selections ................................................................... 203
5  Voice Board Parameters ................................................................. 238
6  Voice Channel Parameters ............................................................... 238
7  DV_TPT Field Settings Summary .................................................... 283
8  G.711 Voice Coder Support Fields ................................................... 302
9  Linear PCM Voice Coder Support Fields ......................................... 302
10 OKI ADPCM Voice Coder Support Fields ......................................... 302
11 G.726 Voice Coder Support Fields ................................................... 302
12 DTMF Tone Specifications ............................................................... 317
13 MF Tone Specifications (CCITT R1 Tone Plan) ................................ 318
14 Detecting MF Digits .......................................................................... 319
15 Detecting DTMF Digits ..................................................................... 319
Revision History

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

<table>
<thead>
<tr>
<th>Document No.</th>
<th>Publication Date</th>
<th>Description of Revisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>05-2333-001</td>
<td>September 2004</td>
<td>Initial version of document.</td>
</tr>
</tbody>
</table>
About This Publication

The following topics provide information about this publication:

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

Purpose

This guide provides details about the voice API that is supplied with the Intel® NetStructure™ Host Media Processing (HMP) product, including function descriptions, data structures, and error codes supported on the Linux® and Windows® operating systems. This document is a companion guide to the *Voice API for Host Media Processing Programming Guide*, which provides instructions for developing applications using the voice API.

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

Applicability

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

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

Intended Audience

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

- Distributors
- System Integrators
About This Publication

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

How to Use This Publication

Refer to this publication after you have installed the hardware and the system software which includes the voice software. This publication assumes that you are familiar with the Linux or Windows operating systems and the C programming language.

The information in this guide is organized as follows:

- **Chapter 1, “Function Summary by Category”** introduces the categories of voice functions and provides a brief description of each function.
- **Chapter 2, “Function Information”** provides an alphabetical reference to all voice functions supported on HMP.
- **Chapter 3, “Events”** provides an alphabetical reference to events that may be returned by the voice software on HMP.
- **Chapter 4, “Data Structures”** provides an alphabetical reference to all voice data structures supported on HMP.
- **Chapter 5, “Error Codes”** provides a listing of all error codes that may be returned by the voice software on HMP.
- **Chapter 6, “Supplementary Reference Information”** provides additional reference information on topics such as DTMF and MF Tone Specifications.

A glossary and index are provided for your reference.

Related Information

Refer to the following sources for more information:

- For information about Voice library features and guidelines for building applications using voice software, see the *Voice API for Host Media Processing Programming Guide*.
- For details on the Standard Runtime Library (SRL), supported programming models, and programming guidelines for building all applications, *Standard Runtime Library API Programming Guide*. The Standard Runtime Library is a device-independent library that consists of event management functions and standard attribute functions.
- For details on all functions and data structures in the Standard Runtime Library (SRL) library, see the *Standard Runtime Library API Library Reference*.
- For information on the software release, system requirements, features, and documentation, see the Release Guide for the software release you are using.
For details on known problems and late-breaking updates or corrections to the release documentation, see the Release Update. Be sure to check the Release Update for the software release you are using for any updates or corrections to this publication. Release Updates are available on the Telecom Support Resources website at http://resource.intel.com/telecom/support/documentation/releases/index.htm

For details on installing the HMP software, see the installation guide for your HMP software release.

For details on obtaining, activating, and working with HMP licenses, see the License Manager Administration Guide for your HMP software release.

For guidelines on building applications using Global Call software (a common signaling interface for network-enabled applications, regardless of the signaling protocol needed to connect to the local telephone network), see the Global Call API for Host Media Processing Programming Guide.

For details on all functions and data structures in the Global Call library, see the Global Call API for Host Media Processing Library Reference.

For Global Call IP-specific information, see the Global Call IP for Host Media Processing Technology Guide.

For details on all functions and data structures in the IP Media Library, see the IP Media Library API for Host Media Processing Library Reference.

For guidelines on building applications using the IP Media software, see the IP Media Library API for Host Media Processing Programming Guide.
1 Function Summary by Category

This chapter describes the categories into which the voice library functions can be logically grouped.

- Device Management Functions .................................................. 13
- Configuration Functions ......................................................... 14
- I/O Functions ............................................................................. 14
- I/O Convenience Functions ....................................................... 15
- Streaming to Board Functions .................................................. 16
- Call Status Transition (CST) Event Functions .............................. 17
- TDM Routing Functions ............................................................ 17
- Global Tone Detection (GTD) Functions ..................................... 18
- Global Tone Generation (GTG) Functions ................................... 19
- Volume Functions ....................................................................... 19
- Call Progress Analysis Functions .............................................. 20
- File Manipulation Functions ...................................................... 20
- Structure Clearance Functions .................................................. 21
- Extended Attribute Functions .................................................... 21

1.1 Device Management Functions

Device management functions open and close devices, which include boards and channels.

Before you can call any other library function on a device, that device must be opened using a device management function. The dx_open() function returns a unique voice device handle. This 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, these functions will work on a device whether the device is busy or idle.

For more information about opening and using voice devices, see the Voice API Programming Guide. Also see this guide for more information about naming conventions for board and channel devices.

Use Standard Runtime Library device mapper functions to return information about the structure of the system, such as a list of all boards. This device information is used as input to device
management functions. For more information on device mapper functions, see the *Standard Runtime Library API Library Reference*.

**Note:** These device management functions are separate and distinct from the Device Management API library, which provides run-time control and management of configurable system devices.

The device management functions are:

- **dx_close()**: closes a board or channel device handle
- **dx_open()**: opens a board or channel device handle

### 1.2 Configuration Functions

Configuration functions allow you to alter, examine, and control the physical configuration of an open device. In general, configuration functions operate on an idle device. Configuration functions cause a device to be busy and return the device to an idle state when the configuration is complete.

See the *Voice API Programming Guide* for information about busy and idle states.

The configuration functions are:

- **dx_clrdigbuf()**: clears all digits in the firmware digit buffer
- **dx_GetDllVersion()** *(Windows only)*
  - returns the voice dynamic link library (DLL) version number
- **dx_getfeaturelist()**: returns information about the features supported on the device
- **dx_getparm()**: gets the current parameter settings for an open device
- **dx_libinit()** *(Windows only)*
  - initializes the voice dynamic link library (DLL)
- **dx_setdigtyp()**: controls the types of digits detected by the device
- **dx_setparm()**: sets physical parameters for the device

### 1.3 I/O Functions

An I/O function transfers 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.

I/O functions can be run synchronously or asynchronously, with some exceptions (for example, **dx_setuio()** can be run synchronously only). When running synchronously, they return after
Function Summary by Category

completing successfully or after an error. When running asynchronously, they return immediately to indicate successful initiation (or an error), and continue processing until a termination condition is satisfied. See the *Standard Runtime Library API Programming Guide* for more information on asynchronous and synchronous operation.

A set of termination conditions can be specified for I/O functions, except for `dx_stopch()`. 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 the *Voice API Programming Guide* for information about I/O terminations.

The I/O functions are:

- `dx_dial()`
  - dials an ASCIIZ string of digits
- `dx_getdig()`
  - collects digits from a channel digit buffer
- `dx_play()`
  - plays voice data from any combination of data files, memory, or custom devices
- `dx_playiottdata()`
  - plays voice data from any combination of data files, memory, or custom devices, and lets the user specify format information
- `dx_rec()`
  - records voice data to any combination of data files, memory, or custom devices
- `dx_recioiottdata()`
  - records voice data to any combination of data files, memory, or custom devices, and lets the user specify format information
- `dx_setdevuio()` (Windows only)
  - installs and retrieves user-defined I/O functions in your application
- `dx_setuio()`
  - installs user-defined I/O functions in your application
- `dx_stopch()`
  - forces termination of currently active I/O functions

**Notes:**

1. The `dx_playtone()` function, which is grouped with global tone generation functions, can also be classified as an I/O function and all I/O characteristics apply.

2. The `dx_playvox()` and `dx_recvox()` functions, which are grouped with I/O convenience functions, can also be classified as I/O functions and all I/O characteristics apply.

### 1.4 I/O Convenience Functions

Convenience functions enable you to easily implement certain basic functionality of the library functions. I/O convenience functions simplify synchronous play and record.
The `dx_playf()` function 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_recf()` provides the same single-file convenience for the `dx_rec()` function.

The `dx_playvox()` function also plays voice data stored in a single VOX file. This function internally calls `dx_playiottdata()`. Similarly, `dx_recvox()` records VOX files using `dx_reciottdata()`.

The I/O convenience functions are:

- `dx_playf()`: plays voice data from a single VOX file without the need to specify DX_IOTT
- `dx_playvox()`: plays voice data from a single VOX file using `dx_playiottdata()`
- `dx_playwav()`: plays voice data stored in a single WAVE file
- `dx_recf()`: records voice data from a channel to a single VOX file without the need to specify DX_IOTT
- `dx_recvox()`: records voice data from a channel to a single VOX file using `dx_reciottdata()`
- `dx_recwav()`: records voice data to a single WAVE file

### 1.5 Streaming to Board Functions

The streaming to board feature enables real-time data streaming to the board. Streaming to board functions allow you to create, maintain, and delete a circular stream buffer within the library. These functions also provide notification when high and low water marks are reached. See the *Voice API Programming Guide* for more information about the streaming to board feature.

The streaming to board functions include:

- `dx_CloseStreamBuffer()`: deletes a circular stream buffer
- `dx_GetStreamInfo()`: retrieves information about the circular stream buffer
- `dx_OpenStreamBuffer()`: creates and initializes a circular stream buffer
- `dx_PutStreamData()`: places data into the circular stream buffer
- `dx_ResetStreamBuffer()`: resets internal data for a circular stream buffer
Function Summary by Category

1.6 Call Status Transition (CST) Event Functions

Call status transition (CST) event functions set and monitor CST events that can occur on a device. CST events indicate changes in the status of the call, such as rings or a tone detected, or the line going on-hook or off-hook. See the call status transition structure (DX_CST) description for a full list of CST events.

The `dx_getevt` function retrieves CST events in a synchronous environment. To retrieve CST events in an asynchronous environment, use the Standard Runtime Library event management functions.

The call status transition event functions are:

- `dx_getevt` gets a CST event in a synchronous environment
- `dx_setevtmsk` enables detection of CST events

1.7 TDM Routing Functions

TDM routing functions are used in time division multiplexing (TDM) bus configurations, which include the CT Bus and SCbus. A TDM bus is a resource sharing bus that allows information to be transmitted and received among resources over multiple time slots. On HMP, no physical TDM bus exists but its functionality is implemented in the software.

TDM routing functions enable the application to make or break a connection between voice, telephone network interface, and other resource channels connected via TDM bus time slots. Each device connected to the bus has a transmit component that can transmit on a time slot and a receive component that can listen to a time slot.

The transmit component of each channel of a device is assigned to a time slot at system initialization and download. To listen to other devices on the bus, the receive component of the device channel is connected to any one time slot. Any number of device channels can listen to a time slot.

TDM routing convenience functions, `nr_scroute` and `nr_scunroute`, are provided to make or break a half or full-duplex connection between any two channels transmitting on the bus. These functions are not a part of any library but are provided in a separate C source file called `sctools.c`. The functions are defined in `sctools.h`. 
The TDM routing functions are:

- **dx_getctinfo( )**
  returns information about voice device connected to TDM bus

- **dx_getxmitslot( )**
  returns the number of the TDM bus time slot connected to the transmit component of a voice channel

- **dx_listen( )**
  connects the listen (receive) component of a voice channel to a TDM bus time slot

- **dx_unlisten( )**
  disconnects the listen (receive) component of a voice channel from TDM bus time slot

- **nr_scroute( )**
  makes a half or full-duplex connection between two channels transmitting on the TDM bus

- **nr_scunroute( )**
  breaks a half or full-duplex connection between two TDM bus devices

### 1.8 Global Tone Detection (GTD) Functions

The global tone detection (GTD) functions define and enable detection of single and dual frequency tones that fall outside the range of those automatically provided with the voice driver. They include 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 API Programming Guide for a full description of global tone detection.

The global tone detection functions are:

- **dx_addtone( )**
  adds a user-defined tone

- **dx_blddt( )**
  builds a user-defined dual frequency tone description

- **dx_blddtcad( )**
  builds a user-defined dual frequency tone cadence description

- **dx_bldst( )**
  builds a user-defined single frequency tone description

- **dx_bldstcad( )**
  builds a user-defined single frequency tone cadence description

- **dx_deltones( )**
  deletes all user-defined tones

- **dx_distone( )**
  disables detection of user-defined tones
Function Summary by Category

1.9 Global Tone Generation (GTG) Functions

Global tone generation (GTG) functions define and play single and dual tones that fall outside the range of those automatically provided with the voice driver.

The `dx_bldtnge` function defines a tone template structure, TN_GEN. The `dx_playtone` function can then be used to generate the tone.

See the Voice API Programming Guide for a full description of global tone generation.

The global tone generation functions are:

- `dx_bldtnge` builds a user-defined tone template structure, TN_GEN
- `dx_playtone` plays a user-defined tone as defined in TN_GEN structure
- `dx_playtoneEx` plays the cadenced tone defined by TN_GENCAD structure

Note: The `dx_playtone` and `dx_playtoneEx` functions can also be classified as an I/O function and all I/O characteristics apply.

1.10 Volume Functions

Volume functions adjust the volume of the play. A volume modification table is associated with each channel, and can be used for increasing or decreasing the volume. These tables have default values which can be changed using the `dx_setsvmt` function.

See the Voice API Programming Guide for more information about the volume feature and volume modification tables.

The volume functions are:

- `dx_adjsv` adjusts volume immediately
- `dx_clrsvcond` clears volume conditions
- `dx_getcursv` returns current volume settings

- `dx_enbtone` enables detection of user-defined tones
- `dx_setgtdmp` sets amplitudes used by global tone detection (GTD)
Function Summary by Category

dx_getsvmt( )
returns current volume modification table

dx_setsvcond( )
sets conditions (such as digit) for volume adjustment

dx_setsvmt( )
changes default values of volume modification table

1.11 Call Progress Analysis Functions

Call progress analysis functions are used to change the default definition of call progress analysis tones. See the Voice API Programming Guide for more information about call progress analysis.

The call progress analysis functions are:

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

dx_deletetone( )
deletes a specific call progress tone

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

1.12 File Manipulation Functions

Supported on Windows only. These file manipulation functions map to C run-time functions, and can only be used if the file is opened with the function. The arguments for these Intel® Dialogic® functions are identical to the equivalent Microsoft® Visual C++ run-time functions.

dx_fileclose( )
closes the file associated with the handle

dx_fileerrno( )
obeats the system error value

dx_fileopen( )
opens the file specified by filep

dx_fileread( )
reads data from the file associated with the handle

dx_fileseek( )
moves a file pointer associated with the handle

dx_filewrite( )
writes data from a buffer into a file associated with the handle
1.13 Structure Clearance Functions

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 data structures. These structures are discussed in Chapter 4, “Data Structures”.

- `dx_clrcap()` clears all fields in a DX_CAP structure
- `dx_clrtpt()` clears all fields in a DV_TPT structure

1.14 Extended Attribute Functions

Voice library extended attribute functions return information specific to the voice device specified in the function call.

- `ATDX_BDNAMEP()` returns a pointer to the board device name string
- `ATDX_BDTYPE()` returns the board type for the device
- `ATDX_CHNAMES()` returns a pointer to an array of channel name strings
- `ATDX_CHNUM()` returns the channel number on board associated with the channel device handle
- `ATDX_CONNTYPE()` returns the connection type for a completed call
- `ATDX_CPERROR()` returns call progress analysis error
- `ATDX_CPTERM()` returns last call progress analysis termination
- `ATDX_DEVTYPE()` returns device type (board or channel)
- `ATDX_STATE()` returns the current state of the device
- `ATDX_TERMMSK()` returns the reason for last I/O function termination in a bitmap
- `ATDX_TONEID()` returns the tone ID (used in global tone detection)
- `ATDX_TRCOUNT()` returns the last record or play transfer count
This chapter provides an alphabetical reference to the functions in the voice library. A general description of the function syntax convention is provided before the detailed function information.

2.1 Function Syntax Conventions

The voice functions use the following syntax:

\[
\text{data_type } \text{voice_function}(\text{device_handle, parameter1, ..., parameterN})
\]

where:

- **data type** refers to the data type, such as integer, long or void
- **voice_function** represents the function name. Typically, voice functions begin with “dx” although there are exceptions. Extended attribute functions begin with “ATDX.”
- **device_handle** represents the device handle, which is a numerical reference to a device, obtained when a device is opened. The device handle is used for all operations on that device.
- **parameter1** represents the first parameter
- **parameterN** represents the last parameter
ATDX_BDNAMEP( ) — return a pointer to the board device name

Name: char * ATDX_BDNAMEP(chdev)

Inputs: int chdev * valid channel device handle

Returns: pointer to board device name string if successful
pointer to ASCII string “Unknown device” if error

Includes: srllib.h
dxxxlib.h

Category: Extended Attribute

Mode: synchronous

Description

The ATDX_BDNAMEP( ) function returns a pointer to the board device name 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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using dx_open( )</td>
</tr>
</tbody>
</table>

Cautions

None.

Errors

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

Example

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

main()
{
    int chdev, bddev;
    char *bdnamep;
    ...
    /* Open the channel device */
    if ((chdev = dx_open("dxxxBlc1", 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 */
}

See Also

None.
ATDX_BDTYPE( ) — return the board type for the device

ATDX_BDTYPE( )

**Name:** long ATDX_BDTYPE(dev)

**Inputs:** int dev
- valid board or channel device handle

**Returns:** board or channel device type if successful
AT_FAILURE if error

**Includes:** srlib.h
dxxxlib.h

**Category:** Extended Attribute

**Mode:** synchronous

### Description

The ATDX_BDTYPE( ) function returns the board type for the device specified in `dev`.

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dev</code></td>
<td>specifies the valid device handle obtained when a board or channel was opened using <code>dx_open( )</code></td>
</tr>
</tbody>
</table>

Possible return values are the following:

- **DI_D41BD**
  - D/41 Board Device. This value represents the “dxxxBn type” devices (virtual boards).

- **DI_D41CH**
  - D/41 Channel Device. This value represents the “dxxxBnCm” type devices (channel device).

The values `DI_D41BD` and `DI_D41CH` will be returned for any D/41 board, and any board which emulates the voice resources of multiple D/41 boards.

### Cautions

None.

### Errors

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

```c
#include <stdio.h>
#include <srilib.h>
#include <dxxxlib.h>

#define ON 1

int 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;
    }
}
```

See Also

None.
ATDX_CHNAMES( ) — retrieve all channel names for a board

**Name:** char ** ATDX_CHNAMES(bddev)

**Inputs:** int bddev • valid board device handle

**Returns:** pointer to array of channel names if successful
point to “Unknown device” if error

**Includes:** srllib.h
dxxxlib.h

**Category:** Extended Attribute

**Mode:** synchronous

---

### Description

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

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bddev</td>
<td>specifies the valid board device handle obtained when the board was opened using dx_open( )</td>
</tr>
</tbody>
</table>

### Cautions

None.

### 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*.

### Example

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

main()
{
    int bddev, cnt;
    char **chnames;
    long subdevs;

    /* Open the board device */
    if ((bddev = dx_open("dxxxB1",NULL)) == -1) { /* Process error */
```
retrieve all channel names for a board — ATDX_CHNAMES()

} /* 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
 */

See Also

None.
ATDX_CHNUM( ) — return the channel number

ATDX_CHNUM( )

| Name:    | long ATDX_CHNUM(chdev) |
| Inputs:  | int chdev              |
| Returns: | channel number if successful |
|          | AT_FAILURE if error    |
| Includes:| srllib.h               |
|          | dxxxlib.h              |
| Category:| Extended Attribute     |
| Mode:    | synchronous            |

### Description

The **ATDX_CHNUM( )** function returns the channel number associated with the channel device `chdev`. Channel numbering starts at 1.

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <code>dx_open()</code></td>
</tr>
</tbody>
</table>

### Cautions

None.

### Errors

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

### Example

```c
#include <srllib.h>
#include <dxxxlib.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 */
```
return the channel number — ATDX_CHNUM( )

} /* Use chno for application-specific purposes */

See Also

None.
ATDX_CONNTYPE( )

**Name:** long ATDX_CONNTYPE(chdev)

**Inputs:** int chdev  • valid channel device handle

**Returns:** connection type if success
AT_FAILURE if error

**Includes:** srllib.h
dxxlib.h

**Category:** Extended Attribute

**Mode:** synchronous

---

### Description

The ATDX_CONNTYPE( ) function returns the connection type for a completed call on the channel device **chdev**. Use this function when a CR_CNCT (called line connected) is returned by **ATDX_CPTERM( )** after termination of **dx_dial( )** with call progress analysis enabled.

See the *Voice API Programming Guide* for more information about call progress analysis.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <strong>dx_open( )</strong></td>
</tr>
</tbody>
</table>

Possible return values are the following:

- **CON_CAD**
  Connection due to cadence break

- **CON_PVD**
  Connection due to positive voice detection

- **CON_PAMD**
  Connection due to positive answering machine detection

### Cautions

None.

### Errors

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

```c
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.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 call progress analysis 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 progress analysis, using
     * the default call progress 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 progress 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;
}

 See Also

- dx_dial()
- ATDX_CPTERM()
- DX_CAP data structure
return the call progress analysis error — ATDX_CPERROR( )

ATDX_CPERROR( )

Name: long ATDX_CPERROR(chdev)
Inputs: int chdev • valid channel device handle
Returns: call progress analysis error if success
          AT_FAILURE if function fails
Includes: srllib.h
dxxxlib.h
Category: Extended Attribute
Mode: synchronous

Description

The ATDX_CPERROR( ) function returns the call progress analysis error that caused dx_dial( ) to terminate when checking for operator intercept Special Information Tone (SIT) sequences. See the Voice API Programming Guide for more information about call progress analysis.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using dx_open( )</td>
</tr>
</tbody>
</table>

Cautions

None.

Errors

When dx_dial( ) terminates due to a call progress analysis error, CR_ERROR is returned by ATDX_CPTERM( ).

If CR_ERROR is returned, use ATDX_CPERROR( ) to determine the call progress analysis error. One of the following values will be returned:

CR_LGTUERR
lower frequency greater than upper frequency

CR_MEMERR
out of memory trying to create temporary Special Information Tone (SIT) tone templates (exceeds maximum number of templates)

CR_MXFRQERR
invalid ca_maxtimefreq field in DX_CAP. If the ca_mxtimefreq parameter for each SIT is nonzero, it must have a value greater than or equal to the ca_timefreq parameter for the same SIT.
ATDX_CPERROR() — return the call progress analysis error

CR_OVRLPERR
overlap in selected SIT tones

CR_TMOUTOFF
timeout waiting for SIT tone to terminate (exceeds a ca_mxtimefrq parameter)

CR_TMOUTON
timeout waiting for SIT tone to commence

CR_UNEXPTN
unexpected SIT tone (the sequence of detected tones did not correspond to the SIT sequence)

CR_UPFRQERR
invalid upper frequency selection. This value must be nonzero for detection of any SIT.

Example

```c
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.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
",
        ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
    dx_close( dxxxdev );
    exit( 1 );
  }

  /*
   * Perform an outbound dial with call progress analysis, using
   * the default call progress 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

- dx_dial()
- ATDX_CPTERM()
- DX_CAP data structure
ATDX_CPTERM( ) — return the last result of call progress analysis termination

ATDX_CPTERM( )

Name:  long ATDX_CPTERM(chdev)

Inputs:  int chdev  • valid channel device handle

Returns:  last call progress analysis termination if successful
          AT_FAILURE if error

Includes:  srllib.h
dxxxlib.h

Category:  Extended Attribute

Mode:  synchronous

Description

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

See the Voice API Programming Guide for more information about call progress analysis.

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  
Called line was busy.

CR_CEPT  
Called line received Operator Intercept (SIT). Extended attribute functions provide
information on detected frequencies and duration.

CR_CNCT  
Called line was connected.

CR_FAXTONE  
Called line was answered by fax machine or modem.

CR_NOANS  
Called line did not answer.

CR_NORB  
No ringback on called line.

CR_STOPD  
Call progress analysis stopped due to dx_stopch( ).
return the last result of call progress analysis termination — ATDX_CPTERM()
ATDX_CPTERM( ) — return the last result of call progress analysis termination

    case AT_FAILURE: /* Error */
    
See Also

- dx_dial()
- DX_CAP data structure
**ATDX_DEVTYPE()**

**Name:** long ATDX_DEVTYPE(dev)

**Inputs:**
- int dev • valid board or channel device handle

**Returns:**
- device type if successful
- AT_FAILURE if error

**Includes:**
- srllib.h
- dxxxlib.h

**Category:** Extended Attribute

**Mode:** synchronous

---

**Description**

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dev</td>
<td>specifies the valid device handle obtained when a board or channel was opened using <code>dx_open()</code></td>
</tr>
</tbody>
</table>

Possible return values are the following:

- **DT_DXBD**
  - Board device (indicates virtual board)
- **DT_DXCH**
  - Channel device

**Cautions**

None.

**Errors**

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

**Example**

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

main()
{
    int bddev;
    long devtype;
```
/* Open the board device */
if ((bddev = dx_open("dxxxBl",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 */
.
.

**See Also**

None.
ATDX_STATE( )

Name: long ATDX_STATE(chdev)

Inputs: int chdev • valid channel device handle

Returns: current state of channel if successful
AT_FAILURE if error

Includes: srllib.h
dxxlib.h

Category: Extended Attribute

Mode: synchronous

Description

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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

Note: A device is idle if there is no I/O function active on it.
ATDX_STATE( ) — return the current state of the channel

■ Cautions

This function extracts the current state from the driver and requires the same processing resources as many other functions. For this reason, applications should not base their state machines on this function.

■ Errors

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

■ Example

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

main()
{
  int chdev;
  long chstate;

  /* Open the channel device */
  if ((chdev = dx_open("dxxxBlC1",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);
  .
  .
}
```

■ See Also

None.
return the reason for the last I/O function termination — ATDX_TERMMSK( )

ATDX_TERMMSK( )

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

Description

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using dx_open( )</td>
</tr>
</tbody>
</table>

Possible return values are the following:

TM_DIGIT
Specific digit received

TM_EOD
End of data reached (on playback, receive)

TM_ERROR
I/O device error

TM_IDDTIME
Inter-digit delay

TM_MAXDTMF
Maximum DTMF count

TM_MAXSIL
Maximum period of silence

TM_MAXTIME
Maximum function time exceeded

TM_NORMTERM
Normal termination (for dx_dial( ))

TM_TONE
Tone-on/off event
ATDX_TERMMSK() — return the reason for the last I/O function termination

TM_USRSTOP
Function stopped by user

■ Cautions

- If several termination conditions are met at the same time, several bits will be set in the termination bitmap.
- When both DX_MAXDTMF and DX_DIGMASK termination conditions are specified in the DV_TPT structure, and both conditions are satisfied, the ATDX_TERMMSK() function will return the TM_MAXDTMF termination event only.

For example, with a DX_MAXDTMF condition of 2 digits maximum and a DX_DIGMASK condition of digit “1”, if the digit string “21” is received, both conditions are satisfied but only TM_MAXDTMF will be reported by ATDX_TERMMSK().

This behavior differs from Springware products, where both TM_MAXDTMF and TM_DIGIT will be returned when both DX_MAXDTMF and DX_DIGMASK termination conditions are specified in the DV_TPT structure and both are satisfied by the user input.

■ Errors

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

■ Example

```c
#include <stdio.h>
#include <fcntl.h>
#include <srllib.h>
#include <dxxxlib.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_CONT;
tpt[3].tp_termno = DX_LCOFF; /* last entry in the table */
tpt[3].tp_length = 10; /* terminate on loop current drop */
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");
    
}

See Also

- **DV_TPT** data structure to set termination conditions
- Event Management functions to retrieve termination events asynchronously (in the *Standard Runtime Library API Programming Guide* and *Standard Runtime Library API Library Reference*)
- **ATEC_TERMMSK()** in the *Continuous Speech Processing API Library Reference*
**ATDX_TONEID( )**

**Name:** long ATDX_TONEID(chdev)

**Inputs:** int chdev  • valid channel device handle

**Returns:** user-defined tone ID if successful
AT_FAILURE if error

**Includes:** srllib.h
dxxxlib.h

**Category:** Extended Attribute

**Mode:** synchronous

---

**Description**

The **ATDX_TONEID( )** function returns the user-defined tone ID that terminated an I/O function. This termination is indicated by **ATDX_TERMMSK( )** returning TM_TONE.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <strong>dx_open( )</strong></td>
</tr>
</tbody>
</table>

**Cautions**

None.

**Errors**

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

**Example**

```c
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.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, amplitude 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 = TP_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 = TP_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 = TP_MAXTIME;

if (dx_playtone( chdev, &tngen, tp, 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 ) );
}
ATDX_TONEID( ) — return user-defined tone ID that terminated I/O function

```c
/*
 * Continue Processing
 */

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

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

- **See Also**

None.
The ATDX_TRCOUNT() function returns the number of bytes transferred during the last play or record on the channel chdev.

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

**Cautions**

None.

**Errors**

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

**Example**

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

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

    /* Open the channel device */
    if ((chdev = dx_open("DxxxBlCl",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 */
io_t.io_type = IO_DEV|IO_EOT;
io_t.io_buffp = 0;
io_t.io_offset = 0L;
io_t.io_length = 50000L;
if((io_t.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,&io_t,tpt,EV_SYNC) == -1) {
    /* process error */
}

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

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

See Also

None.
**dx_addtone( )**

---

**Name:** int dx_addtone(chdev, digit, digtype)

**Inputs:**
- int chdev • valid 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

**Mode:** synchronous

---

### Description

The `dx_addtone( )` function adds a user-defined 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.

For more information on user-defined tones and global tone detection (GTD), see the Voice API Programming Guide.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <code>dx_open( )</code></td>
</tr>
<tr>
<td>digit</td>
<td>specifies an optional digit to associate with the tone. When the tone is detected, the digit will be placed in the <code>DV_DIGIT</code> digit buffer. These digits can be retrieved using <code>dx_getdig( )</code> (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.</td>
</tr>
</tbody>
</table>
dx_addtone( ) — add a user-defined tone

Cautions

- 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.
- Do not use dx_addtone( ) to change a tone that has previously been added.
- There are limitations to the number of tones or tone templates that can be added to a channel, depending on the type of board and other factors. See the global tone detection topic in the Voice API Programming Guide for details.
- When using this function in a multi-threaded application, use critical sections or a semaphore around the function call to ensure a thread-safe application. Failure to do so will result in “Bad Tone Template ID” errors.

Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. One of the following error codes may be returned:

- 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 dg_type value in tone template description
- EDX_FREQDET
  Invalid tone frequency
- EDX_INV_SUBCMD
  Invalid sub-command
- EDX_MAX_TMPLT
  Maximum number of user-defined tones for the board
add a user-defined tone — dx_addtone()

EDX_SYSTEM
Error from operating system

EDX_TONEID
Invalid tone template ID

Example

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.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 );
```

Voice API for HMP Library Reference — September 2004
dx_addtone( ) — add a user-defined tone

```c
printf( "Lasterror = %d  Err Msg = %s\n",
    ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ));
dx_close( dxxxdev );
exi( 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 );
exi( 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 );
exi( 1 );
}

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

/* Terminate the Program */
exi( 0 );
"

See Also

- dx_blddt( ), dx_bldst( ), dx_blddtcad( ), dx_bldstcad( )
```
add a user-defined tone — dx_addtone()

- dx_distone()
- dx_enbtone()
- global tone detection in the Voice API Programming Guide
- dx_getev()
- DX_CST data structure
- sr_getevtdatap() in the Standard Runtime Library API Library Reference
- dx_getdig()
- dx_setdigtyp()
- DV_DIGIT data structure
dx_addvoldig( ) — set a DTMF digit to adjust volume

**dx_addvoldig( )**

**Name:** int dx_addvoldig(chdev, digit, adjval)

**Inputs:**
- int chdev  
  • valid 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:** Volume

**Mode:** synchronous

---

**Description**

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

This function assumes that the volume modification table has not been modified using the **dx_setsvmt( )** function.

For more information about volume control, see the *Voice API Programming Guide*. For information about volume data structures, see the **DX_SVMT** and the **DX_SVCB** data structures.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <strong>dx_open( )</strong></td>
</tr>
<tr>
<td>digit</td>
<td>specifies a DTMF digit (0-9, *, #) that will modify volume by the amount specified in <strong>adjval</strong></td>
</tr>
<tr>
<td>adjval</td>
<td>specifies a volume adjustment value to take effect whenever the digit specified in <strong>digit</strong> occurs</td>
</tr>
</tbody>
</table>

The following are valid values:
- **SV_ADD2DB** – increase play volume by 2 dB
- **SV_SUB2DB** – decrease play volume by 2 dB
- **SV_NORMAL** – set play volume to origin when the play begins (**digit** must be set to NULL)

To start play volume at the origin, set **digit** to NULL and set **adjval** to **SV_NORMAL**.
Cautions

- Calls to this function are cumulative. To reset or remove any condition, you should clear all adjustment conditions and reset if required. For example, if DTMF digit “1” has already been set to increase play volume by one step, a second call that attempts to redefine digit “1” to the origin will have no effect on the volume, but will be added to the array of conditions; the digit will retain its original setting.

- 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.

- Digits that are used for play adjustment may also be used as a terminating condition. If a digit is defined as both, then both actions are applied upon detection of that digit.

Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function `ATDV_LASTERR()` to obtain the error code or use `ATDV_ERRMSGP()` to obtain a descriptive error message. One of the following error codes may be returned:

- `EDX_BADPARM`  
  Invalid parameter

- `EDX_BADPROD`  
  Function not supported on this board

- `EDX_SVADJBLKS`  
  Invalid number of play adjustment blocks

- `EDX_SYSTEM`  
  Error from operating system

Example

```c
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.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"
  );
```
dx_addvoldig( ) — set a DTMF digit to adjust volume

```c
printf( "cassert error = %d  Err Msg = %s\n",
       ATDV_LAESTERR( 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 );

[See Also]
- dx_adjsv()
- dx_clrsvcond()
- dx_getcursv()
- dx_getsvmt()
- dx_setsvcond()
- dx_setsvmt()```
dx_adjsv( )

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

**Inputs:**
- int chdev • valid channel device handle
- unsigned short tabletype • type of table to set (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:** Volume

**Mode:** synchronous

---

### Description

The `dx_adjsv( )` function adjusts volume immediately, and for all subsequent plays on a specified channel (until changed or cancelled). The volume can be set to a specific value, adjusted incrementally, or can be set to toggle. See the `action` parameter description for information. On HMP, speed adjustment is not supported.

The `dx_adjsv( )` function uses the volume modification table to make adjustments to play volume. This table has 21 entries that represent different levels of volume. There are up to ten levels above and below the regular volume. This table can be set with explicit values using `dx_setsvmt( )` or default values can be used. See the *Voice API Programming Guide* for detailed information about this table.

**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 volume is based on the most recent modification.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <code>dx_open( )</code></td>
</tr>
<tr>
<td>tabletype</td>
<td>specifies the volume modification table:</td>
</tr>
<tr>
<td></td>
<td>• SV_VOLUMETBL – retrieve the volume modification table values</td>
</tr>
</tbody>
</table>
dx_adjsv() — adjust volume immediately

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>action</td>
<td>specifies the type of adjustment to make. Set to one of the following:</td>
</tr>
<tr>
<td></td>
<td>• SV_ABSPOS — set volume to a specified position in the appropriate table. (The position is set using the adjsize parameter.)</td>
</tr>
<tr>
<td></td>
<td>• SV_RELCURPOS — adjust volume by the number of steps specified using the adjsize parameter</td>
</tr>
<tr>
<td></td>
<td>• SV_TOGGLE — toggle between values specified using the adjsize parameter</td>
</tr>
<tr>
<td>adjsize</td>
<td>specifies the size of the adjustment. The adjsize parameter has a different value depending on how the adjustment type is set using the action parameter.</td>
</tr>
<tr>
<td></td>
<td>• If action is SV_ABSPOS, adjsize specifies the position between -10 to +10 in the volume modification table that contains the required volume adjustment. The origin (regular volume) has a value of 0 in the table.</td>
</tr>
<tr>
<td></td>
<td>• If action is SV_RELCURPOS, adjsize specifies the number of positive or negative steps in the volume modification table by which to adjust the volume. For example, specify -2 to lower the volume by 2 steps.</td>
</tr>
<tr>
<td></td>
<td>• If action is SV_TOGGLE, adjsize specifies the values between which volume will toggle. SV_CURLASTMOD sets the current volume to the last modified volume level. SV_CURORIGIN resets the current volume level to the origin (that is, regular volume). SV_RESETORIG resets the current volume to the origin and the last modified volume to the origin. SV_TOGORIGIN sets the volume to toggle between the origin and the last modified level of volume.</td>
</tr>
</tbody>
</table>

■ Cautions
None.

■ Errors
If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function ATDV_LASTERR() to obtain the error code or use ATDV_ERRMSGP() to obtain a descriptive error message. One of the following error codes may be returned:

EDX_BADPARM
Invalid parameter

EDX_BADPROD
Function not supported on this board

EDX_SYSTEM
Error from operating system

■ Example

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

// ...
main()
{
    int dxxxdev;

    /*
     * Open the Voice Channel Device and Enable a Handler
     */
    if ( ( dxxxdev = dx_open("dxxx81c1", 0 ) ) == -1 ) {
        perror("dxxx81c1");
        exit(1);
    }

    /*
     * Modify the Volume of the playback so that it is 4dB
     * higher than normal.
     */
    if ( dx_adjsv( dxxxdev, SV_VOLUME, 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);
}

See Also

- dx_setsvcond()
- dx_clrsvcond()
- dx_getcursv()
- dx_getsvmt()
- volume modification table in the Voice API Programming Guide
- DX_SVMT data structure
dx_blddt( ) — define a user-defined dual-frequency tone

dx_blddt( )

**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

**Mode:** synchronous

**Description**

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

Issuing `dx_blddt( )` defines a new tone. You must use `dx_addtone( )` to add the tone to the channel and enable its detection.

For more information about global tone detection, see the *Voice API Programming Guide*.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tid</td>
<td>specifies a unique identifier for the tone. See Cautions for more information about the tone ID.</td>
</tr>
<tr>
<td>freq1</td>
<td>specifies the first frequency (in Hz) for the tone</td>
</tr>
<tr>
<td>fq1dev</td>
<td>specifies the allowable deviation (in Hz) for the first frequency</td>
</tr>
<tr>
<td>freq2</td>
<td>specifies the second frequency (in Hz) for the tone</td>
</tr>
<tr>
<td>fq2dev</td>
<td>specifies the allowable deviation (in Hz) for the second frequency</td>
</tr>
<tr>
<td>mode</td>
<td>specifies whether tone detection notification will occur on the leading or trailing edge of the tone. Set to one of the following:</td>
</tr>
<tr>
<td></td>
<td>• TN_LEADING</td>
</tr>
<tr>
<td></td>
<td>• TN_TRAILING</td>
</tr>
</tbody>
</table>
define a user-defined dual-frequency tone — dx_blddt( )

■ Cautions

- Only one tone per process can be defined at any time. Ensure that `dx_blddt()` is called for each `dx_addtone()` function. 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.

- On Windows, do not use tone IDs 261, 262 and 263; they are reserved for library use.

- When using this function in a multi-threaded application, use critical sections or a semaphore around the function call to ensure a thread-safe application. Failure to do so will result in “Bad Tone Template ID” errors.

■ Errors

If this function returns -1 to indicate failure, call the Standard Runtime Library (SRL) Standard Attribute function `ATDV_LASTERR()` to obtain the error code, or use `ATDV_ERRMSGP()` to obtain a descriptive error message. For a list of error codes returned by `ATDV_LASTERR()`, see the Error Codes chapter.

■ Example

```c
#include <stdio.h>
#include <srllib.h>
#include <dxlib.h>
#define TID_1   101

main()
{
  int dxxxdev;

  /*
  *  Open the Voice Channel Device and Enable a Handler
  */
  if ( ( dxxxdev = dx_open( "dxxxB1C1", 0 ) ) == -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" );
  }
```

Voice API for HMP Library Reference — September 2004
dx_blddt( ) — define a user-defined dual-frequency tone

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

See Also

- global tone detection topic in *Voice API Programming Guide*
- dx_bldst( )
- dx_blddtcad( )
- dx_bldstcad( )
- dx_addtone( )
- dx_distone( )
- dx_enbtone( )
**dx_blddtcad( )**

**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 10 msec
- unsigned int ontdev: tone-on time deviation in 10 msec
- unsigned int offtime: tone-off time in 10 msec
- unsigned int offtdev: tone-off time deviation in 10 msec
- unsigned int repcnt: number of repetitions if cadence

**Returns:**
- 0 if success
- -1 if failure

**Includes:** srlib.h
dxxlib.h

**Category:** Global Tone Detection

**Mode:** synchronous

---

### Description

The **dx_blddtcad( )** function defines a user-defined dual frequency cadenced 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 **dx_blddtcad( )** defines a new tone. You must use **dx_addtone( )** to add the tone to the channel and enable its detection.

For more information about global tone detection, see the *Voice API Programming Guide*.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tid</td>
<td>specifies a unique identifier for the tone. See Cautions for more information on the tone ID.</td>
</tr>
<tr>
<td>freq1</td>
<td>specifies the first frequency (in Hz) for the tone</td>
</tr>
<tr>
<td>fq1dev</td>
<td>specifies the allowable deviation (in Hz) for the first frequency</td>
</tr>
<tr>
<td>freq2</td>
<td>specifies the second frequency (in Hz) for the tone</td>
</tr>
<tr>
<td>fq2dev</td>
<td>specifies the allowable deviation (in Hz) for the second frequency</td>
</tr>
<tr>
<td>ontime</td>
<td>specifies the length of time for which the cadence is on (in 10 msec units)</td>
</tr>
</tbody>
</table>
dx_blddtcad() — define a user-defined dual frequency cadenced tone

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ontdev</td>
<td>specifies the allowable deviation for on time (in 10 msec units)</td>
</tr>
<tr>
<td>offtime</td>
<td>specifies the length of time for which the cadence is off (in 10 msec units)</td>
</tr>
<tr>
<td>offtdev</td>
<td>specifies the allowable deviation for off time (in 10 msec units)</td>
</tr>
<tr>
<td>repcnt</td>
<td>specifies the number of repetitions for the cadence (that is, the number of times that an on/off signal is repeated)</td>
</tr>
</tbody>
</table>

#### Cautions

- Only one user-defined tone per process can be defined at any time. `dx_blddtcad()` will replace the previous user-defined tone definition.
- On Windows, do not use tone IDs 261, 262 and 263; they are reserved for library use.
- When using this function in a multi-threaded application, use critical sections or a semaphore around the function call to ensure a thread-safe application. Failure to do so will result in “Bad Tone Template ID” errors.

#### Errors

If this function returns -1 to indicate failure, call the Standard Runtime Library (SRL) Standard Attribute function `ATDV_LASTERR()` to obtain the error code, or use `ATDV_ERRMSGP()` to obtain a descriptive error message. For a list of error codes returned by `ATDV_LASTERR()`, see the Error Codes chapter.

#### Example

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

#define TID_2 102

main()
{
    int dxxxdev;

    /*
     * Open the Voice Channel Device and Enable a Handler
     */
    if ( ( dxxxdev = dx_open( "dxxxB1C1", 0 ) ) == -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\n" );
        printf( "Template\n" );
    }
}
```
/* Continue Processing */

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

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

See Also

- global tone detection topic in Voice API Programming Guide
- dx_bldst()
- dx_blddt() 
- dx_bldstcad()
- dx_addtone()
- dx_distone()
- dx_enbtone()
**dx_bldstcad( ) — define a user-defined single-frequency cadenced tone**

#### dx_bldstcad( )

**Name:** int dx_bldstcad(tid, freq, fqdev, ontime, ontdev, offtime, offsetdev, 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 10 msec
- unsigned int ontdev: • on time deviation in 10 msec
- unsigned int offtime: • tone off time in 10 msec
- unsigned int offsetdev: • off time deviation in 10 msec
- unsigned int repcnt: • repetitions if cadence

**Returns:**
- 0 if success
- -1 if failure

**Includes:**
- srllib.h
- dxxxlib.h

**Category:** Global Tone Detection

**Mode:** synchronous

---

**Description**

The **dx_bldstcad( )** function defines a user-defined, single-frequency, cadenced 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. You must use **dx_addtone( )** to add the tone to the channel and enable its detection.

For more information about global tone detection, see the *Voice API Programming Guide*.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tid</td>
<td>specifies a unique identifier for the tone. See Cautions for more information about the tone ID.</td>
</tr>
<tr>
<td>freq</td>
<td>specifies the frequency (in Hz) for the tone</td>
</tr>
<tr>
<td>fqdev</td>
<td>specifies the allowable deviation (in Hz) for the frequency</td>
</tr>
<tr>
<td>ontime</td>
<td>specifies the length of time for which the cadence is on (in 10 msec units)</td>
</tr>
<tr>
<td>ontdev</td>
<td>specifies the allowable deviation for on time (in 10 msec units)</td>
</tr>
<tr>
<td>offtime</td>
<td>specifies the length of time for which the cadence is off (in 10 msec units)</td>
</tr>
</tbody>
</table>
**define a user-defined single-frequency cadenced tone — dx_bldstcad()**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>offtdev</td>
<td>specifies the allowable deviation for off time (in 10 msec units)</td>
</tr>
<tr>
<td>repcnt</td>
<td>specifies the number of repetitions for the cadence (i.e., the number of times that an on/off signal is repeated)</td>
</tr>
</tbody>
</table>

**Cautions**

- Only one tone per application may be defined at any time. `dx_bldstcad()` will replace the previous user-defined tone definition.
- On Windows, do not use tone IDs 261, 262 and 263; they are reserved for library use.
- When using this function in a multi-threaded application, use critical sections or a semaphore around the function call to ensure a thread-safe application. Failure to do so will result in “Bad Tone Template ID” errors.

**Errors**

If this function returns -1 to indicate failure, call the Standard Runtime Library (SRL) Standard Attribute function `ATDV_LASTERR()` to obtain the error code, or use `ATDV_ERRMSGP()` to obtain a descriptive error message. For a list of error codes returned by `ATDV_LASTERR()`, see the Error Codes chapter.

**Example**

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

#define TID_4   104

main()
{
    int  dxxxdev;

    /*
    * Open the Voice Channel Device and Enable a Handler
    */
    if ( ( dxxxdev = dx_open( "dxxxB1C1", 0 ) ) == -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
    */
    .
    .
    .
    */
```
dx_bldstcad() — define a user-defined single-frequency cadenced tone

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

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

See Also

- global tone detection topic in *Voice API Programming Guide*
- dx_blddtcad()
- dx_blddt()
- dx_bldst()
- dx_addtone()
- dx_distone()
- dx_enbtone()
define a user-defined single-frequency tone — dx_bldst( )

**dx_bldst( )**

**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:** synchronous

---

### Description

The `dx_bldst( )` function defines a user-defined 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. You must use `dx_addtone( )` to add the tone to the channel and enable its detection.

For more information about global tone detection, see the Voice API Programming Guide.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tid</td>
<td>specifies a unique identifier for the tone. See Cautions for more information about the tone ID.</td>
</tr>
<tr>
<td>freq</td>
<td>specifies the frequency (in Hz) for the tone</td>
</tr>
<tr>
<td>fqdev</td>
<td>specifies the allowable deviation (in Hz) for the frequency</td>
</tr>
<tr>
<td>mode</td>
<td>specifies whether detection is on the leading or trailing edge of the tone. Set to one of the following:</td>
</tr>
<tr>
<td></td>
<td>• TN_LEADING</td>
</tr>
<tr>
<td></td>
<td>• TN_TRAILING</td>
</tr>
</tbody>
</table>

### Cautions

- Only one tone per application may be defined at any time. `dx_bldst( )` will replace the previous user-defined tone definition.
- On Windows, do not use tone IDs 261, 262 and 263; they are reserved for library use.
dx_bldst() — define a user-defined single-frequency tone

- When using this function in a multi-threaded application, use critical sections or a semaphore around the function call to ensure a thread-safe application. Failure to do so will result in “Bad Tone Template ID” errors.

**Errors**

If this function returns -1 to indicate failure, call the Standard Runtime Library (SRL) Standard Attribute function ATDV_LASTERR() to obtain the error code, or use ATDV_ERRMSGP() to obtain a descriptive error message. For a list of error codes returned by ATDV_LASTERR(), see the Error Codes chapter.

**Example**

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

#define TID_3 103

main()
{
  int dxxxdev;

  /*
   * Open the Voice Channel Device and Enable a Handler
   */
  if ( ( dxxxdev = dx_open( "dxxxBlCl", 0 ) ) == -1 ) {
    perror( "dxxxBlCl" );
    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 );
}
```

**See Also**

- global tone detection topic in *Voice API Programming Guide*
- dx_blddtcad()
define a user-defined single-frequency tone — **dx_bldst()**

- **dx_blddt()**
- **dx_bldstcad()**
- **dx_addtone()**
- **dx_distone()**
- **dx_enbtone()**
**dx_bldtngen( ) — define a tone for generation**

**dx_bldtngen( )**

**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 msec units

**Returns:**  
none

**Includes:**  
srllib.h  
dxxxlib.h

**Category:** Global Tone Generation

**Mode:** synchronous

**Description**

The **dx_bldtngen( )** function is a convenience function that defines a tone for generation by setting up the tone generation template (TN_GEN) and 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.

For more information about Global Tone Generation, see the Voice API Programming Guide.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tngenp</td>
<td>points to the TN_GEN data structure where the tone generation template is output</td>
</tr>
<tr>
<td>freq1</td>
<td>specifies the frequency of tone 1 in Hz. Valid range is 200 to 3000 Hz.</td>
</tr>
<tr>
<td>freq2</td>
<td>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.</td>
</tr>
<tr>
<td>ampl1</td>
<td>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.</td>
</tr>
<tr>
<td>ampl2</td>
<td>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.</td>
</tr>
<tr>
<td>duration</td>
<td>specifies the duration of the tone in 10 msec units. A value of -1 specifies infinite duration (the tone will only terminate upon an external terminating condition).</td>
</tr>
</tbody>
</table>

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.
Cautions

None.

Errors

None.

Example

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

main()
{
  TN_GEN   tngen;
  int      dxxxdev;

  /*
   * Open the Voice Channel Device and Enable a Handler
   */
  if ( ( dxxxdev = dx_open( "dxxxBlCl", 0 ) ) == -1 ) {
    perror( "dxxxBlCl" );
    exit( 1 );
  }

  /*
   * Build a Tone Generation Template.
   * This template has Frequency1 = 1140,
   * Frequency2 = 1020, amplitude 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 );
}
```

See Also

- TN_GEN structure
- dx_playtone()
- global tone generation topic in Voice API Programming Guide
dx_close( ) — close a channel or board device handle

**dx_close( )**

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

### Description

The `dx_close( )` function closes a channel device handle or board device handle that was previously opened using `dx_open( )`.

This function does not affect any action occurring on a device. It does not affect the hook state or any of the parameters that have been set for the device. It releases the handle and breaks the link between the calling process and the device, regardless of whether the device is busy or idle.

**Note:** The `dx_close( )` function disables the generation of all events.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dev</strong></td>
<td>specifies the valid device handle obtained when a board or channel was opened using <code>dx_open( )</code></td>
</tr>
</tbody>
</table>

### Cautions

- Once a device is closed, a process can no longer act 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.
- Do not use the operating system `close( )` command to close a voice device; unpredictable results will occur.
- The `dx_close( )` function discards any outstanding events on that handle.
- If you close a device via `dx_close( )` after modifying volume table values using `dx_setsvmt( )`, the `dx_getcursv( )` function may return incorrect volume settings for the device. This is because the next `dx_open( )` resets the volume tables to their default values.

### Errors

In Windows, if this function returns -1 to indicate failure, a system error has occurred; use `dx_fileerrno( )` to obtain the system error value. Refer to the `dx_fileerrno( )` function for a list of the possible system error values.
In Linux, if this function returns -1 to indicate failure, check `errno` for one of the following reasons:

EBADF
Invalid file descriptor

EINTR
A signal was caught

EINVAL
Invalid argument

**Example**

This example illustrates how to close a channel device handle.

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

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

**See Also**

- `dx_open()`
**dx_CloseStreamBuffer( ) — delete a circular stream buffer**

**dx_CloseStreamBuffer( )**

- **Name:** int dx_CloseStreamBuffer(hBuffer)
- **Inputs:** int hBuffer • stream buffer handle
- **Returns:** 0 if successful
- -1 if failure
- **Includes:** srllib.h
dxxlib.h
- **Category:** streaming to board
- **Mode:** synchronous

### Description

The `dx_CloseStreamBuffer( )` function deletes the circular stream buffer identified by the stream buffer handle. If the stream buffer is currently in use (playing), this function returns -1 as an error.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hBuffer</td>
<td>specifies the stream buffer handle obtained from <code>dx_OpenStreamBuffer( )</code></td>
</tr>
</tbody>
</table>

### Cautions

You cannot delete a circular stream buffer while it is in use by a play operation. If you try to delete the buffer in this situation, the `dx_CloseStreamBuffer( )` function will return -1 as an error.

### Errors

This function returns -1 on error. The error can occur if you passed the wrong buffer handle to the function call or if the buffer is in use by an active play.

To see if the buffer is in use by an active play, call `dx_GetStreamInfo( )` and check the item “currentState” in the `DX_STREAMSTAT` structure. A value of `ASSIGNED_STREAM_BUFFER` for this item means that the buffer is currently in use in a play. A value of `UNASSIGNED_STREAM_BUFFER` means that the buffer is not being used currently in any play.

Unlike other voice API library functions, the streaming to board functions do not use SRL device handles. Therefore, `ATDV_LASTERR( )` and `ATDV_ERRMSGP( )` cannot be used to retrieve error codes and error descriptions.
Example

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

main()
{
    int nBuffSize = 32768, vDev = 0;
    int hBuffer = -1;
    char pData[1024];
    DX_IOTT iott;
    DV_TPT ptpt;

    if ((hBuffer = dx_OpenStreamBuffer(nBuffSize)) < 0)
    {
        printf("Error opening stream buffer \n");
        exit(1);
    }
    if ((vDev = dx_open("dxxxB1C1", 0)) < 0)
    {
        printf("Error opening voice device\n");
        exit(2);
    }

    iott.io_type = IO_STREAM|IO_EOT;
    iott.io_bufp = 0;
    iott.io_offset = 0;
    iott.io_length = -1;  /* play until STREAM_EOD */
    iott.io_fhandle = hBuffer;

    dx_clrpt(&ptpt,1);
    ptpt.tp_type = IO_EOT;
    ptpt.tp_termno = DX_MAXDTMF;
    ptpt.tp_length = 1;
    ptpt.tp_flags = TF_MAXDTMF;

    if (dx_play(vDev, &iott, &ptpt, EV_ASYNC) < 0)
    {
        printf("Error in dx_play() %d\n", ATDV_LASTERR(vDev));
    }
    /* Repeat the following until all data is streamed */

    if (dx_PutStreamData(hBuffer, pData, 1024, STREAM_CONT) < 0)
    {
        printf("Error in dx_PutStreamData \n");
        exit(3);
    }
    /* Wait for TDX_PLAY event and other events as appropriate */

    if (dx_CloseStreamBuffer(hBuffer) < 0)
    {
        printf("Error closing stream buffer \n");
    }
}
```

See Also

- dx_OpenStreamBuffer()
- dx_GetStreamInfo()
dx_clrcap( )

Name: void dx_clrcap(capp)

Inputs: DX_CAP *capp • pointer to call progress analysis parameter data structure

Returns: none

Includes: srllib.h
dxxxlib.h

Category: Structure Clearance

Mode: synchronous

■ Description

The dx_clrcap( ) function clears all 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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>capp</td>
<td>pointer to call progress analysis parameter data structure, DX_CAP. For more information on this structure, see DX_CAP, on page 285.</td>
</tr>
</tbody>
</table>

■ Cautions

Clear the DX_CAP structure 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.

■ Errors

None.

■ Example

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

main()
{
  DX_CAP cap;
  int chdev;

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

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

82 Voice API for HMP Library Reference — September 2004
clear all fields in a DX_CAP structure — dx_clrcap()

   /* continue with call progress analysis */
   .
   .

See Also

- dx_dial()
- DX_CAP data structure
- call progress analysis topic in the Voice API Programming Guide
**dx_clrdigbuf()**

**Name:** int dx_clrdigbuf(chdev)

**Inputs:**
- int chdev  
  - valid channel device handle

**Returns:**
- 0 if success
- -1 if failure

**Includes:**
- srllib.h
- dxxxlib.h

**Category:** Configuration

**Mode:** synchronous

---

### Description

The `dx_clrdigbuf()` function clears all digits in the firmware digit buffer of the channel specified by `chdev`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <code>dx_open()</code></td>
</tr>
</tbody>
</table>

### Cautions

The function will fail and return -1 if the channel device handle is invalid or the channel is busy.

### Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function `ATDV_LASTERR()` to obtain the error code or use `ATDV_ERRMSGP()` to obtain a descriptive error message. One of the following error codes may be returned:

- `EDX_BADPARM`
  - Invalid parameter
- `EDX_SYSTEM`
  - Error from operating system

### Example

See the Example code in the function descriptions for `dx_getdig()`, `dx_play()`, and `dx_rec()` for more examples of how to use `dx_clrdigbuf()`.

```c
#include <srllib.h>
#include <dxxxlib.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 Also

None.
dx_clrsvcond( ) — clear all volume adjustment conditions

**dx_clrsvcond( )**

- **Name:** int dx_clrsvcond(chdev)
- **Inputs:** int chdev • valid channel device handle
- **Returns:** 0 if success
  -1 if failure
- **Includes:** srllib.h
dxxxlib.h
- **Category:** Volume
- **Mode:** synchronous

### Description

The `dx_clrsvcond( )` function clears all volume adjustment conditions that have been previously set using `dx_setsvcond( )` or the convenience function `dx_addvoldig( )`. On HMP, speed adjustment is not supported.

Before resetting an adjustment condition, you must first clear all current conditions by using this function, and then reset conditions using `dx_setsvcond( )` or `dx_addvoldig( )`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <code>dx_open( )</code></td>
</tr>
</tbody>
</table>

### Cautions

None.

### Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function `ATDV_LASTERR( )` to obtain the error code or use `ATDV_ERRMSGP( )` to obtain a descriptive error message. One of the following error codes may be returned:

- **EDX_BADPARM**
  - Invalid parameter
- **EDX_BADPROD**
  - Function not supported on this board
- **EDX_SYSTEM**
  - Error from operating system
clear all volume adjustment conditions — dx_clrsvcond()

Example

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

int dxxxdev;

/*
 * Open the Voice Channel Device and Enable a Handler
 */
if ( ( dxxxdev = dx_open( "dxxxB1C1", 0 ) ) == -1 ) {
    perror( "dxxxB1C1" );
    exit( 1 );
}

/*
 * Clear all Volume Conditions
 */
if ( dx_clrsvcond( dxxxdev ) == -1 ) {
    printf( "Unable to Clear the Volume\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 */
exiut( 0 );
```

See Also

- dx_setsvmt()
- dx_addvoldig()
- volume modification tables in Voice API Programming Guide
- DX_SVCB data structure
dx_clrtpt() — clear all fields in a DV_TPT structure

**dx_clrtpt()**

**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

**Mode:** synchronous

---

#### Description

The dx_clrtpt() function clears all fields except tp_type and tp_nextp in the specified number of DV_TPT structures. This function is provided as a convenient way of clearing a DV_TPT structure, before reinitializing it for a new set of termination conditions.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tptp</td>
<td>points to the first DV_TPT structure to be cleared</td>
</tr>
<tr>
<td>size</td>
<td>indicates the number of DV_TPT structures to clear. If size is set to 0, the function will return a 0 to indicate success. For more information on this structure, see DV_TPT, on page 279.</td>
</tr>
</tbody>
</table>

**Notes:**
1. The DV_TPT is defined in srllib.h rather than dxxxlib.h since it can be used by other non-voice devices.
2. Before 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

#### Cautions

If tp_type in the DV_TPT structure is set to IO_LINK, you must set tp_nextp to point to the next DV_TPT in the chain. The last DV_TPT in the chain must have its tp_type field set to IO_EOT. 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.

To reinitialize DV_TPT structures with a new set of conditions, call dx_clrtpt() only after the links have been set up properly, as illustrated in the Example.
clear all fields in a DV_TPT structure — dx_clrtpt()

- **Errors**

  The function will fail and return -1 if IO_EOT is encountered in the tp_type field before the number of DV_TPT structures specified in `size` have been cleared.

- **Example**

  ```c
  #include <srllib.h>
  #include <dxxxlib.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 */
      .
      .
  }
  ```

- **See Also**

  - DV_TPT data structure
dx_createtone( ) — create a new tone definition for a specific call progress tone

**dx_createtone( )**

**Name:** int dx_createtone(brdhdl, toneid, *tonedata, mode)

**Inputs:**
- int brdhdl — a valid board device handle
- int toneid — tone ID of the call progress tone
- TONE_DATA *tonedata — pointer to the TONE_DATA structure
- unsigned short mode — mode

**Returns:**
- 0 if successful
- -1 if failure

**Includes:** srllib.h
dxxmplib.h

**Category:** Call Progress Analysis

**Mode:** Asynchronous or synchronous

---

**Description**

The **dx_createtone( )** function creates a new tone definition for a specific call progress tone. On successful completion of the function, the TONE_DATA structure is used to create a tone definition for the specified call progress tone.

Prior to creating a new tone definition with **dx_createtone( )**, use **dx_querytone( )** to get tone information for that tone, then use **dx_deletetone( )** to delete that tone.
create a new tone definition for a specific call progress tone — `dx_createtone()`

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>brdhdl</code></td>
<td>specifies a valid board device handle (not a virtual board device) of the format <code>brdBn</code> obtained by a call to <code>dx_open()</code></td>
</tr>
<tr>
<td><code>toneid</code></td>
<td>specifies the tone ID of the call progress tone whose definition needs to be created. Valid values are:</td>
</tr>
<tr>
<td></td>
<td>- <code>TID_DIAL_LCL</code></td>
</tr>
<tr>
<td></td>
<td>- <code>TID_DIAL_INTL</code></td>
</tr>
<tr>
<td></td>
<td>- <code>TID_BUSY1</code></td>
</tr>
<tr>
<td></td>
<td>- <code>TID_RNGBK1</code></td>
</tr>
<tr>
<td></td>
<td>- <code>TID_BUSY2</code></td>
</tr>
<tr>
<td></td>
<td>- <code>TID_RNGBK2</code></td>
</tr>
<tr>
<td></td>
<td>- <code>TID_DISCONNECT</code></td>
</tr>
<tr>
<td></td>
<td>- <code>TID_FAX1</code></td>
</tr>
<tr>
<td></td>
<td>- <code>TID_FAX2</code></td>
</tr>
<tr>
<td></td>
<td>- <code>TID_SIT_NC</code> (no circuit found)</td>
</tr>
<tr>
<td></td>
<td>- <code>TID_SIT_IC</code> (operator intercept)</td>
</tr>
<tr>
<td></td>
<td>- <code>TID_SIT_VC</code> (vacant circuit)</td>
</tr>
<tr>
<td></td>
<td>- <code>TID_SIT_RO</code> (reorder)</td>
</tr>
<tr>
<td><code>tonedata</code></td>
<td>specifies a pointer to the TONE_DATA data structure which contains the tone information to be created for the call progress tone identified by <code>toneid</code></td>
</tr>
<tr>
<td><code>mode</code></td>
<td>specifies how the function should be executed, either <code>EV_ASYNC</code> (asynchronous) or <code>EV_SYNC</code> (synchronous)</td>
</tr>
</tbody>
</table>

When running asynchronously, the function returns 0 to indicate that it initiated successfully and generates the TDX_CREATETONE event to indicate completion or the TDX_CREATETONE_FAIL event to indicate failure.

By default, this function runs synchronously and returns 0 to indicate completion.

**Cautions**

- Only the default call progress tones as listed in the `toneid` parameter description are supported for this function.
- If you call `dx_createtone()` prior to calling `dx_deletetone()`, then `dx_createtone()` will fail with an error `EDX_TNQUERYDELETE`.
- To modify a default tone definition, use the three functions `dx_querytone()`, `dx_deletetone()`, and `dx_createtone()` in this order, for one tone at a time.
- When `dx_createtone()` is issued on a board device in asynchronous mode, and the function is immediately followed by another similar call prior to completion of the previous call on the same device, the subsequent call will fail with device busy.
dx_createtone() — create a new tone definition for a specific call progress tone

 Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function ATDV_LASTERR() to obtain the error code or use ATDV_ERRMSGP() to obtain a descriptive error message. One of the following error codes may be returned:

EDX_BADPARM
invalid parameter

EDX_SYSTEM
error from operating system

EDX_TNPARM
invalid tone template parameter

EDX_TNQUERYDELETE
tone not queried or deleted prior to create

 Example

```c
#include "srllib.h"
#include "dxxxlib.h"

main()
{
  int brdhdl; /* board handle */
  
  /* Opening board */
  if ((brdhdl = dx_open("brdB1",0)) == -1) {
    printf("Cannot open board\n");
    /* Perform system error processing */
    exit(1);
  }

  /* Get the Tone Information for the TID_BUSY1 tone*/
  int result;
  TONE_DATA tonedata;
  if ((result = dx_querytone(brdhdl, TID_BUSY1, &tonedata, EV_ASYNC)) == -1) {
    printf("Cannot obtain tone information for TID_BUSY1\n");
    /* Perform system error processing */
    exit(1);
  }

  /* Delete the current TID_BUSY1 call progress tone before creating a new definition*/
  if ((result = dx_deletetone(brdhdl, TID_BUSY1, EV_ASYNC)) == -1) {
    printf("Cannot delete the TID_BUSY1 tone\n");
    /* Perform system error processing */
    exit(1);
  }

  /* Change call progress default Busy tone */
  tonedata.numofseg = 1; /* Single segment tone */
  toneinfo.toneseg[0].tn1_min = 0; /* Min. Frequency for Tone 1 (in Hz) */
  toneinfo.toneseg[0].tn1_max = 450; /* Max. Frequency for Tone 1 (in Hz) */
  toneinfo.toneseg[0].tn2_min = 0; /* Min. Frequency for Tone 2 (in Hz) */
  toneinfo.toneseg[0].tn2_max = 150; /* Max. Frequency for Tone 2 (in Hz) */
  toneinfo.toneseg[0].tnon_min = 400; /* Debounce Min. ON Time */
  toneinfo.toneseg[0].tnon_max = 550; /* Debounce Max. ON Time */
}```
create a new tone definition for a specific call progress tone — dx_createtone()

```c

toneinfo.toneseg[0].tnoff_min = 400;    /* Debounce Min. OFF Time */
toneinfo.toneseg[0].tnoff_max = 550;    /* Debounce Max. OFF Time */

tonedata.toneseg[0].tn_rep_cnt = 4;

if ((result = dx_createtone(brdhdl, TID_BUSY1, &tonedata, EV_SYNC)) == -1) {
    printf("create tone for TID_BUSY1 failed\n");
    /* Perform system error processing */
    exit(1);
}
```

See Also

- dx_deletetone()
- dx_querytone()
dx_deletetone( ) — delete a specific call progress tone

dx_deletetone( )

Name: int dx_deletetone(brdhdl, toneid, mode)

Inputs: int brdhdl
        • a valid board device handle
        int toneid
        • tone ID of the call progress tone
        unsigned short mode
        • mode

Returns: 0 if successful
                -1 if failure

Includes: srllib.h
                dxxxlib.h

Category: Call Progress Analysis

Mode: Asynchronous or synchronous

Description

The dx_deletetone( ) function deletes the specified call progress tone.

Prior to creating a new tone definition with dx_createtone( ), use dx_querytone( ) to get tone information for that tone, then use dx_deletetone( ) to delete that tone.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>brdhdl</td>
<td>specifies a valid board device handle (not a virtual board device) of the format brdBn obtained by a call to dx_open( )</td>
</tr>
<tr>
<td>toneid</td>
<td>specifies the tone ID of the call progress tone. Valid values are:</td>
</tr>
<tr>
<td></td>
<td>• TID_DIAL_LCL</td>
</tr>
<tr>
<td></td>
<td>• TID_DIAL_INTL</td>
</tr>
<tr>
<td></td>
<td>• TID_BUSY1</td>
</tr>
<tr>
<td></td>
<td>• TID_RNGBK1</td>
</tr>
<tr>
<td></td>
<td>• TID_BUSY2</td>
</tr>
<tr>
<td></td>
<td>• TID_RNGBK2</td>
</tr>
<tr>
<td></td>
<td>• TID_DISCONNECT</td>
</tr>
<tr>
<td></td>
<td>• TID_FAX1</td>
</tr>
<tr>
<td></td>
<td>• TID_FAX2</td>
</tr>
<tr>
<td></td>
<td>• TID_SIT_NC (no circuit found)</td>
</tr>
<tr>
<td></td>
<td>• TID_SIT_IC (operator intercept)</td>
</tr>
<tr>
<td></td>
<td>• TID_SIT_VC (vacant circuit)</td>
</tr>
<tr>
<td></td>
<td>• TID_SIT_RO (reorder)</td>
</tr>
<tr>
<td>mode</td>
<td>specifies how the function should be executed, either EVASYNC (asynchronous) or EV_SYNC (synchronous)</td>
</tr>
</tbody>
</table>
delete a specific call progress tone — dx_deletetone( )

When running asynchronously, the function returns 0 to indicate that it initiated successfully and generates the TDX_DELETETONE event to indicate completion or the TDX_DELETETONE_FAIL event to indicate failure. The TONE_DATA structure should remain in scope until the application receives these events.

By default, this function runs synchronously and returns 0 to indicate completion.

■ Cautions

- Only the default call progress tones as listed in the toneid parameter description are supported for this function.
- When dx_deletetone( ) is issued on a board device in asynchronous mode, and the function is immediately followed by another similar call prior to completion of the previous call on the same device, the subsequent call will fail with device busy.

■ Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. One of the following error codes may be returned:

EDX_BADPARM
  invalid parameter

EDX_SYSTEM
  error from operating system

EDX_TONEID
  bad tone template ID

■ Example

```c
#include "srllib.h"
#include "dxxxlib.h"

int main()
{
  int brdhdl; /* board handle */
  ...
  /* Open board */
  if ((brdhdl = dx_open("brdB1",0)) == -1)
  {
    printf("Cannot open board\n");
    /* Perform system error processing */
    exit(1);
  }
  /* Delete the current TID_BUSY1 call progress tone*/
  int result;
  if ((result = dx_deletetone(brdhdl, TID_BUSY1, &tonedata, EV_SYNC)) == -1)
  {
    printf("Cannot delete the TID_BUSY1 tone \n");
    /* Perform system error processing */
    exit(1);
  }
}
```

dx_deletetone( ) — delete a specific call progress tone

- See Also
  - dx_createtone( )
  - dx_querytone( )
**dx_deltones( )**

**Name:** int dx_deltones(chdev)

**Inputs:**
- int chdev • valid channel device handle

**Returns:**
- 0 if successful
- -1 if error

**Includes:** srllib.h
dxxxlib.h

**Category:** Global Tone Detection

**Mode:** synchronous

---

**Description**

The `dx_deltones( )` function deletes 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 set by `dx_bldtt( )`, `dx_bldst( )`, `dx_bldstcad( )`, or `dx_blddtcad( )`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <code>dx_open( )</code></td>
</tr>
</tbody>
</table>

**Cautions**

When using this function in a multi-threaded application, use critical sections or a semaphore around the function call to ensure a thread-safe application. Failure to do so will result in “Bad Tone Template ID” errors.

**Errors**

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function `ATDV_LASTERR( )` to obtain the error code or use `ATDV_ERRMSGP( )` to obtain a descriptive error message. One of the following error codes may be returned:

- **EDX_BADPARM**
  - Invalid parameter

- **EDX_BADPROD**
  - Function not supported on this board

- **EDX_SYSTEM**
  - Error from operating system
dx_deltones() — delete all user-defined tones

Example

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

main()
{
    int  dxxxdev;

    /*
     * Open the Voice Channel Device and Enable a Handler
     */
    if ( ( dxxxdev = dx_open( "dxxxB1C1", 0 ) ) == -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 );
}
```

See Also

Adding and Enabling User-defined Tones:
- `dx_addtone()`
- `dx_enbtone()`

Building Tones:
- `dx_blddt()`
- `dx_bldst()`
- `dx_bldstcad()`
- `dx_blddtcad()`
dx_dial( )

**Name:** int dx_dial(chdev, dialstrp, capp, mode)

**Inputs:**
- int chdev: valid channel device handle
- char *dialstrp: pointer to the ASCIIZ dial string
- DX_CAP *capp: pointer to call progress analysis parameter structure
- unsigned short mode: asynchronous/synchronous setting and call progress analysis flag

**Returns:**
- 0 to indicate successful initiation (asynchronous)
- ≥0 to indicate call progress analysis result if successful (synchronous)
- -1 if failure

**Includes:** srllib.h
- dxxxlib.h

**Category:** I/O

**Mode:** asynchronous or synchronous

### Description

The `dx_dial( )` function dials an ASCIIZ string on an open, idle channel and optionally enables call progress analysis to provide information about the call. For detailed information on call progress analysis, see the Voice API Programming Guide. See also the Global Call API Programming Guide for information on call progress analysis.

To determine the state of the channel during a dial and/or call progress analysis, use ATDX_STATE( ).

**Notes:**
1. `dx_dial( )` doesn’t affect the hook state.
2. `dx_dial( )` doesn’t wait for dial tone before dialing.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <code>dx_open( )</code></td>
</tr>
<tr>
<td>dialstrp</td>
<td>points to the ASCII dial string. <code>dialstrp</code> must contain a null-terminated string of ASCII characters. For a list of valid dialing and control characters, see Table 1.</td>
</tr>
</tbody>
</table>
dx_dial( ) — dial an ASCII string

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>capp</td>
<td>points to the call progress analysis parameter structure, DX_CAP. To use the default call progress analysis parameters, specify NULL in capp and DX_CALLP in mode.</td>
</tr>
</tbody>
</table>
| mode      | specifies whether the ASCII string will be dialed with or without call progress analysis enabled, and whether the function will run asynchronously or synchronously. This parameter is a bit mask that can be set to a combination of the following values:  
  • DX_CALLP – enables call progress analysis  
  • EV_ASYNC – runs dx_dial() asynchronously  
  • EV_SYNC – runs dx_dial() synchronously (default)  
  If dx_dial() with call progress analysis is performed on a channel that is onhook, the function will only dial digits. Call progress analysis will not occur. |

- **Asynchronous Operation**

For asynchronous operation, set the mode field to EV_ASYNC, using a bitwise OR. The function returns 0 to indicate it has initiated successfully, and generates one of the following termination events to indicate completion:

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDX_CALLP</td>
<td>termination of dialing (with call progress analysis)</td>
</tr>
<tr>
<td>TDX_DIAL</td>
<td>termination of dialing (without call progress analysis)</td>
</tr>
</tbody>
</table>

Use SRL Event Management functions to handle the termination event.

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.

- **Synchronous Operation**

By default, this function runs synchronously, and returns a 0 to indicate that it has completed successfully.

When synchronous dialing terminates, the function returns the call progress result (if call progress analysis is enabled) or 0 to indicate success (if call progress analysis isn’t enabled).

- **Valid Dial String Characters**

The following is a list of valid dialing and control characters.
When using `dx_dial()` , be aware of the following considerations:

- Dial string characters are case-sensitive.
- The default dialing mode is “T” (DTMF tone dialing).
- When you change the dialing mode by specifying the M or T control characters, the dialing mode remains in effect for that `dx_dial()` invocation only. The dialing mode is reset to the default of T (DTMF) for the next invocation, unless you specify otherwise.
- The `dx_dial()` function does not support dial tone detection.
- Dialing parameter default values can be set or retrieved using `dx_getparm()` and `dx_setparm()` ; see board and channel parameter defines in these function descriptions.
- Invalid characters that are part of a dial string are ignored and an error will not be generated. For instance, a dial string of “(123) 456-7890” is equivalent to “1234567890”.

## Cautions

- If you attempt to dial a channel in MF mode and do not have MF capabilities on that channel, DTMF tone dialing is used.
- Issuing a `dx_stopch()` on a channel that is dialing with call progress analysis disabled has no effect on the dial, and will return 0. The digits specified in the `dialstrp` parameter will still be dialed.
- Issuing a `dx_stopch()` on a channel that is dialing with call progress analysis enabled will cause the dialing to complete, but call progress analysis will not be executed. The digits

### Table 1. Valid Dial String Characters

<table>
<thead>
<tr>
<th>Characters</th>
<th>Description</th>
<th>Valid in Dial Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>On Keypad</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8 9</td>
<td>digits</td>
<td>Yes</td>
</tr>
<tr>
<td>*</td>
<td>asterisk or star</td>
<td>Yes</td>
</tr>
<tr>
<td>#</td>
<td>pound, hash, number, or octothorpe</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Not on Keypad</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>c</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>d</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Special Control</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>pause for 2.5 seconds (comma)</td>
<td>Yes</td>
</tr>
<tr>
<td>T</td>
<td>Dial Mode: Tone (DTMF) (default)</td>
<td>Yes</td>
</tr>
<tr>
<td>M</td>
<td>Dial Mode: MF</td>
<td>Yes</td>
</tr>
</tbody>
</table>
dx_dial( ) — dial an ASCII string

specified in the dialstrp parameter will be dialed. Any call progress analysis information collected prior to the stop will be returned by extended attribute functions.

- Issue this function when the channel is idle.
- Clear the DX_CAP structure 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.

errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP() to obtain a descriptive error message. One of the following error codes may be returned:

EDX_BADPARM
Invalid parameter

EDX_BUSY
Channel is busy

EDX_SYSTEM
Error from operating system

example

This example demonstrates how to use dx_dial( ) and call progress analysis (synchronous mode) on Springware boards. On HMP, dx_dial( ) supports call progress analysis directly; you do not use dx_initcallp( ) to initialize call progress analysis.

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.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 call progress analysis default local dial tone
     */
    if (dx_chgfreq( TID_DIAL_LCL, 425, 150, 0, 0 ) < 0) {
        /* handle error */
    }
```
/**
  * Change call progress analysis default busy cadence
  */
if (dx_chgdur( TID_BUSY1, 550, 400, 550, 400 ) < 0) {
  /* handle error */
} if (dx_chgrepnt( TID_BUSY1, 4 ) < 0) {
  /* handle error */
}

/*
 * Now enable call progress analysis 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", 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 );

### See Also

- `dx_stopch()`  
- event management functions in the *Standard Runtime Library API Library Reference*  
- `ATDX_CPTERM()` (to retrieve termination reason and events for `dx_dial()` with call progress analysis)  
- `ATDX_TERMMSK()` (to retrieve termination reason for `dx_dial()` without call progress analysis)
dx_dial() — dial an ASCIIZ string

- DX_CAP data structure
- call progress analysis topic in the Voice API Programming Guide
- ATDX_CONNTYPE()
dx_distone( )

**Name:** int dx_distone(chdev, toneid, evt_mask)

**Inputs:**
- int chdev • valid channel device handle
- int toneid • tone template identification
- int evt_mask • event mask

**Returns:**
- 0 if success
- -1 if error

**Includes:** srlib.h
dxxxlib.h

**Category:** Global Tone Detection

**Mode:** synchronous

---

### Description

The `dx_distone()` function disables detection of a user-defined tone on a channel, as well as the tone-on and tone-off events for that tone. Detection capability for user-defined tones is enabled on a channel by default when `dx_addtone()` is called.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <code>dx_open()</code></td>
</tr>
<tr>
<td>toneid</td>
<td>specifies the user-defined tone identifier for which detection is being disabled</td>
</tr>
<tr>
<td></td>
<td>To disable detection of all user-defined tones on the channel, set <code>toneid</code> to TONEALL.</td>
</tr>
<tr>
<td>evt_mask</td>
<td>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 (</td>
</tr>
<tr>
<td></td>
<td>• DM_TONEON – disable TONE ON detection</td>
</tr>
<tr>
<td></td>
<td>• DM_TONEOFF – disable TONE OFF detection</td>
</tr>
<tr>
<td></td>
<td><code>evt_mask</code> affects the enabled/disabled status of the tone template and remains in effect until <code>dx_distone()</code> or <code>dx_enbtone()</code> is called again to reset it.</td>
</tr>
</tbody>
</table>

### Cautions

When using this function in a multi-threaded application, use critical sections or a semaphore around the function call to ensure a thread-safe application. Failure to do so will result in “Bad Tone Template ID” errors.
### Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function `ATDV_LASTERR( )` to obtain the error code or use `ATDV_ERRMSGP( )` to obtain a descriptive error message. One of the following error codes may be returned:

- **EDX_BADPARM**
  - Invalid parameter
- **EDX_BADPROD**
  - Function not supported on this board
- **EDX_SYSTEM**
  - Error from operating system
- **EDX_TNMSGSTATUS**
  - Invalid message status setting
- **EDX_TONEID**
  - Bad tone ID

### Example

```c
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#define TID_1   101

main()
{
  int  dxxxdev;

  /*
   * Open the Voice Channel Device and Enable a Handler
   */
  if ( ( dxxxdev = dx_open( "dxxxBlCl", 0 ) ) == -1 ) {
      perror( "dxxxBlCl" );
      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 );
  }
}```
disable detection of a user-defined tone — *dx_distone(*)

```c
/**
 * 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 );
```

- **See Also**
  - *dx_addtone(*)
  - *dx_bldtt(),* *dx_bldst(),* *dx_bldtcad(),* *dx_bldstcad(*)
  - *dx_enbtone(*)
  - global tone detection topic in the *Voice API Programming Guide*
  - *dx_getev(*)
  - *DX_CST* data structure
  - *sr_getevidatap(* in the *Standard Runtime Library API Library Reference*
**dx_enbtone( )** — enable detection of a user-defined tone

### Description

The `dx_enbtone( )` function enables detection of a user-defined tone on a channel, including the tone-on and tone-off events for that tone. Detection capability for tones is enabled on a channel by default when `dx_addtone( )` is called.

See the `dx_addtone( )` function description for information about retrieving call status transition (CST) tone-on and tone-off events.

Use `dx_enbtone( )` to enable a tone that was previously disabled using `dx_distone( )`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <code>dx_open( )</code></td>
</tr>
<tr>
<td>toneid</td>
<td>specifies the user-defined tone identifier for which detection is being enabled To enable detection of all user-defined tones on the channel, set <code>toneid</code> to TONEALL.</td>
</tr>
<tr>
<td>evt_mask</td>
<td>specifies whether to enable detection of the user-defined tone going on or going off. Set to one or both of the following using a bitwise-OR (</td>
</tr>
</tbody>
</table>

### Cautions

When using this function in a multi-threaded application, use critical sections or a semaphore around the function call to ensure a thread-safe application. Failure to do so will result in “Bad Tone Template ID” errors.

**Name:** int `dx_enbtone(chdev, toneid, evt_mask)`  
**Inputs:**  
`int chdev` • valid channel device handle  
`int toneid` • tone template identification  
`int evt_mask` • event mask  
**Returns:**  
0 if success  
-1 if failure  
**Includes:** srllib.h  
dxxxlib.h  
**Category:** Global Tone Detection  
**Mode:** synchronous
enable detection of a user-defined tone — dx_enbtone( )

Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. One of the following error codes may be returned:

EDX_BADPARM
Invalid parameter

EDX_BADPROD
Function not supported on this board

EDX_SYSTEM
Error from operating system

EDX_TONEID
Bad tone ID

EDX_TNMSGSTATUS
Invalid message status setting

Example

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

#define TID_1 101

main()
{
  int dxxxdev;

  /*
   * Open the Voice Channel Device and Enable a Handler
   */
  if ( ( dxxxdev = dx_open( "dxxxB1C1", 0 ) ) == -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 );
  }
```

```c
/* 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 );
```

See Also

- `dx_addtone( )`
- `dx_bldt( ), dx_bldst( ), dx_bldtcad( ), dx_bldstcad( )`
- `dx_distone( )`
- Global tone detection in Voice API Programming Guide
- `dx_getevt( )`
- DX_CST data structure
- `sr_getevtdata( )` in Standard Runtime Library API Library Reference
dx_fileclose( )

**Name:** int dx_fileclose(handle)

**Inputs:**
- int handle
  - handle returned from dx_fileopen()

**Returns:**
- 0 if success
- -1 if failure

**Includes:**
- srllib.h
- dxxxlib.h

**Category:** File Manipulation

**Mode:** synchronous

---

**Description**

Supported on Windows only. The dx_fileclose() function closes a file associated with the device handle returned by the dx_fileopen() function. See the _close function in the Microsoft Visual C++ Run-Time Library Reference for more information.

Use dx_fileclose() instead of _close to ensure the compatibility of applications with the libraries across various versions of Visual C++.

---

**Cautions**

None.

---

**Errors**

If this function returns -1 to indicate failure, a system error has occurred.

---

**Example**

```c
/*
 * 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 */
}

See Also

- dx_fileopen()
- dx_filesok()
- dx_fileread()
- dx_filewrite()
dx_fileerrno( )

**Name:** int dx_fileerrno(void)

**Inputs:** none

**Returns:** system error value

**Includes:** srllib.h
dxxlib.h

**Category:** File Manipulation

**Mode:** synchronous

---

**Description**

Supported on Windows only. The *dx_fileerrno*() function returns the global system error value from the operating system.

Call *dx_fileerrno*() to obtain the correct system error value, which provides the reason for the error. For example, if *dx_fileopen*() fails, the error supplied by the operating system can only be obtained by calling *dx_fileerrno*().

**Note:** Unpredictable results can occur if you use the global variable *errno* directly to obtain the system error value. Earlier versions of Visual C++ use different Visual C++ runtime library names. The application and Intel® Dialogic® libraries may then be using separate C++ runtime libraries with separate errno values for each.

See the *Microsoft Visual C++ Run-Time Library Reference* or MSDN documentation for more information on system error values and their meanings. All error values, which are defined as manifest constants in *errno.h*, are UNIX-compatible. The values valid for 32-bit Windows applications are a subset of these UNIX values.

Table 2 lists the system error values that may be returned by *dx_fileerrno*().

**Table 2. System Error Values**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E2BIG</td>
<td>Argument list too long.</td>
</tr>
<tr>
<td>EACCES</td>
<td>Permission denied; indicates a locking or sharing violation. The file’s permission setting or sharing mode does not allow the specified access.</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>No more processes. An attempt to create a new process failed because there are no more process slots, or there is not enough memory, or the maximum nesting level has been reached.</td>
</tr>
</tbody>
</table>
Table 2. System Error Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>Bad file number; invalid file descriptor (file is not opened for writing). Possible causes: 1) The specified file handle is not a valid file-handle value or does not refer to an open file. 2) An attempt was made to write to a file or device opened for read-only access or a locked file.</td>
</tr>
<tr>
<td>EDOM</td>
<td>Math argument.</td>
</tr>
<tr>
<td>EEXIST</td>
<td>Files exist. An attempt has been made to create a file that already exists. For example, the _O_CREAT and _O_EXCL flags are specified in an _open call, but the named file already exists.</td>
</tr>
<tr>
<td>EINTR</td>
<td>A signal was caught.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Invalid argument. An invalid value was given for one of the arguments to a function. For example, the value given for the origin or the position specified by offset when positioning a file pointer (by means of a call to fseek) is before the beginning of the file. Other possibilities are as follows: The dev/evt/handler triplet was not registered or has already been registered. Invalid timeout value. Invalid flags or pmode argument.</td>
</tr>
<tr>
<td>EIO</td>
<td>Error during a Windows open.</td>
</tr>
<tr>
<td>EMFILE</td>
<td>Too many open files. No more file handles are available, so no more files can be opened.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>No such file or directory; invalid device name; file or path not found. The specified file or directory does not exist or cannot be found. This message can occur whenever a specified file does not exist or a component of a path does not specify an existing directory.</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>Not enough memory. Not enough memory is available for the attempted operation. The library has run out of space when allocating memory for internal data structures.</td>
</tr>
<tr>
<td>ENOSPC</td>
<td>Not enough space left on the device for the operation. No more space for writing is available on the device (for example, when the disk is full).</td>
</tr>
<tr>
<td>ERANGE</td>
<td>Result too large. An argument to a math function is too large, resulting in partial or total loss of significance in the result. This error can also occur in other functions when an argument is larger than expected.</td>
</tr>
<tr>
<td>ESR_TMOUT</td>
<td>Timed out waiting for event.</td>
</tr>
<tr>
<td>EXDEV</td>
<td>Cross-device link. An attempt was made to move a file to a different device (using the rename function).</td>
</tr>
</tbody>
</table>

- **Cautions**
  
  None.

- **Errors**
  
  None.

- **Example**

  ```c
  rc=dx_fileopen(FileName, O_RDONLY);
  if (rc == -1) {
      printf("Error opening %s, system error: %d\n", FileName, dx_fileerrno( ) );
  }
  ```
return the system error value — dx_fileerrno()

See Also

None.
dx_fileopen() — open a file

dx_fileopen()

- **Name:** int dx_fileopen(filep, flags, pmode)
- **Inputs:**
  - const char *filep
  - int flags
  - int pmode
- **Returns:**
  - file handle if success
  - -1 if failure
- **Includes:**
  - srllib.h
  - dxxxlib.h
- **Category:** File Manipulation
- **Mode:** synchronous

**Description**

Supported on Windows only. The `dx_fileopen()` function opens a 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.

Use `dx_fileopen()` instead of `_open` to ensure the compatibility of applications with the libraries across various versions of Visual C++.

**Cautions**

When using `dx_reciottdata()` to record WAVE files, you cannot use the O_APPEND mode with `dx_fileopen()`, because for each record, a WAVE file header will be created.

**Errors**

If this function returns -1 to indicate failure, a system error has occurred.

**Example**

```c
/* 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>

tmain()
{
  int chdev;
  DX_IOTT iott;
  DV_TPT tpt;
  DV_DIGIT dig;
  .
  .
```
open a file — dx_fileopen()

/* 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_EOL;
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_EOL; /* 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 */
}

See Also

- dx_fileclose()
- dx_fileseek()
- dx_fileread()
- dx_filewrite()
dx_fileread( ) — read data from a file

**dx_fileread( )**

- **Name:** int dx_fileread(handle, buffer, count)
- **Inputs:**
  - int handle
  - void *buffer
  - unsigned int count
- **Returns:** number of bytes if success
  - -1 if failure
- **Includes:** srllib.h
dxxxlib.h
- **Category:** File Manipulation
- **Mode:** synchronous

## Description

Supported on Windows only. The **dx_fileread( )** function reads data from a file associated with the file handle. 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.

Use **dx_fileread( )** instead of **_read** to ensure the compatibility of applications with the libraries across various versions of Visual C++.

## Cautions

None.

## Errors

If this function returns -1 to indicate failure, a system error has occurred.

## Example

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

main()
{
    int cd; /* channel device 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(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;
  iottp->io_offset = 20001;
  iottp->io_length = 20000;
dx_fileread( ) — read data from a file

/*This block uses standard I/O functions */
iottpp++
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_wstring(devhandle,1,DX_OFFHOOK,EV_SYNC);
dx_clrdbgbuf;
  if(dx_rec(devhandle,iott,(DX_TPT*)NULL,RM_TONE|EV_SYNC) == -1) {
      perror("*");
      exit(1);
  }
dx_clrdbgbuf(devhandle);
  if(dx_play(devhandle,iott,(DX_TPT*)EV_SYNC) == -1 {         perror("*");
      exit(1);
  }
dx_close(devhandle);

See Also

- dx_fileopen( )
- dx_fileclose( )
- dx_fileseek( )
- dx_filewrite( )
dx_fileseek()

**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

**Includes:**
- srllib.h
- dxxxlib.h

**Category:** File Manipulation

**Mode:** synchronous

### Description

Supported on Windows only. The `dx_fileseek()` function moves a file pointer associated with the file 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.

Use `dx_fileseek()` instead of `_lseek` to ensure the compatibility of applications with the libraries across various versions of Visual C++.

### Cautions

Do not use `dx_fileseek()` against files that utilize encoding formats with headers (such as GSM). The `dx_fileseek()` function is not designed to make adjustments for the various header sizes that some encoding formats use.

### Errors

If this function returns -1 to indicate failure, a system error has occurred.

### Example

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

main()
{
  int cd; /* channel device descriptor */
  DX_UIO myio; /* user definable I/O structure */
```
dx_fileseek() — move a file pointer

/*
 * 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(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;
    iottp->io_offset = 20001;
    iottp->io_length = 20000;
/*This block uses standard I/O functions */

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_U10|IO_EOT
iott->io_fhandle = vodat_fd;
iott->io_offset = 10003;
iott->io_length = 20000;
devhandle = dx_open("dxxxB1Cl", 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);

■ See Also

- dx_fileopen()
- dx_fileclose()
- dx_fileread()
- dx_filewrite()
dx_filewrite( ) — write data from a buffer into a file

**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

**Includes:** srllib.h
dxxxlib.h

**Category:** File Manipulation

**Mode:** synchronous

---

**Description**

Supported on Windows only. The dx_filewrite( ) function writes data from a buffer into a file associated with file 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.

Use dx_filewrite( ) instead of _write to ensure the compatibility of applications with the libraries across various versions of Visual C++.

**Cautions**

None.

**Errors**

If this function returns -1 to indicate failure, a system error has occurred.

**Example**

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

main()
{
    int cd;               /* channel device descriptor */
    DX_U10 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(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 */
iopp++;
iopp->io_type = IO_DEV|IO_UIO|IO_CONT
iopp->io_fhandle = vodat_fd;
iopp->io_offset = 20001;
iopp->io_length = 20000;
/*This block uses standard I/O functions */
  iottpp++
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_wstring(devhandle,i,DX_OFFHOOK,EV_SYNC);
dx_clrdbgbuf;
  if(dx_rec(devhandle,iott,(DX_TPT*)NULL,RM_TONE|EV_SYNC) == -1) {
    perror(""");
    exit(1);
  }
dx_clrdbgbuf(devhandle);
  if(dx_play(devhandle,iott,(DX_TPT*)EV_SYNC) == -1) {
    perror(""");
    exit(1);
  }
dx_close(devhandle);

- See Also
  - dx_fileopen()
  - dx_fileclose()
  - dx_fileseek()
  - dx_fileread()
dx_getctinfo()

**Name:** int dx_getctinfo(chdev, ct_devinfop)

**Inputs:**
- int chdev • valid channel device handle
- CT_DEVINFO *ct_devinfop • pointer to device information structure

**Returns:**
- 0 on success
- -1 on error

**Includes:**
- srllib.h
- dxxxlib.h

**Category:** TDM Routing

**Mode:** synchronous

---

**Description**

The `dx_getctinfo()` function returns information about a voice channel of a voice device. The information includes the device family, device mode, type of network interface, bus architecture, and PCM encoding. The information is returned in the CT_DEVINFO structure.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid voice channel handle obtained when the channel was opened using <code>dx_open()</code></td>
</tr>
<tr>
<td>ct_devinfop</td>
<td>specifies a pointer to the CT_DEVINFO structure that will contain the voice channel device information</td>
</tr>
</tbody>
</table>

**Cautions**

This function will fail if an invalid voice channel handle is specified.

**Errors**

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function `ATDV_LASTERR()` to obtain the error code or use `ATDV_ERRMSGP()` to obtain a descriptive error message. One of the following error codes may be returned:

- EDX_BADPARM Parameter error
- EDX_SH_BADEXTTS TDM bus time slot is not supported at current clock rate
- EDX_SH_BADINDX Invalid Switch Handler index number
- EDX_SH_BADTYPE Invalid local time slot channel type (voice, analog, etc.)
dx_getctinfo() — get information about a voice device

EDX_SH_CMDBLOCK
  Blocking command is in progress

EDX_SH_LIBBSY
  Switch Handler library is busy

EDX_SH_LIBNOTINIT
  Switch Handler library is uninitialized

EDX_SH_MISSING
  Switch Handler is not present

EDX_SH_NOCLK
  Switch Handler clock fallback failed

EDX_SYSTEM
  Error from operating system

### Example

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

main()
{
  int chdev; /* Channel device handle */
  CT_DEVINFO ct_devinfo; /* Device information structure */

  /* Open board 1 channel 1 devices */
  if ((chdev = dx_open("dxxxB1C1", 0)) == -1) {
    /* process error */
  }

  /* Get Device Information */
  if (dx_getctinfo(chdev, &ct_devinfo) == -1) {
    printf("Error message = %s", ATDV_ERRMSGP(chdev));
    exit(1);
  }

  printf("%s Product Id = 0x%x, Family = %d, Mode = %d, Network = %d, Bus ...mode = %d, Encoding = %d", ATDV_NAMEP(chdev), ct_devinfo.ct_prodid,
  ...ct_devinfo.ct_devfamily, ct_devinfo.ct_devmode, ct_devinfo.ct_nettype,
  ...ct_devinfo.ct_busmode, ct_devinfo.ct_busencoding);
}
```

### See Also
- gc_GetCTInfo() in the Global Call API Library Reference
- ipm_GetCTInfo() in the IP Media Library API Library Reference
**dx_getcursv()**

**Name:** int dx_getcursv(chdev, curvolp, curspeedp)

**Inputs:**
- int chdev • valid channel device handle
- int * curvolp • pointer to current absolute volume setting
- int * curspeedp • this parameter is not supported in HMP

**Returns:**
- 0 if success
- -1 if failure

**Includes:**
- srllib.h
- dxxxlib.h

**Category:** Volume

**Mode:** synchronous

---

**Description**

The `dx_getcursv()` function returns the specified current volume settings on a channel. For example, use `dx_getcursv()` to determine the volume level set interactively by a listener using DTMF digits during a play. DTMF digits are set as play adjustment conditions using `dx_setsvcond()` or `dx_addvoldig()`. On HMP, speed adjustment is not supported.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <code>dx_open()</code></td>
</tr>
<tr>
<td>curvolp</td>
<td>points to an integer that represents the current absolute volume setting for the channel. This value will be between -30 dB and +10 dB.</td>
</tr>
<tr>
<td>curspeedp</td>
<td>this parameter is not supported in HMP</td>
</tr>
</tbody>
</table>

**Cautions**

If you close a device via `dx_close()` after modifying volume table values using `dx_setsvmt()`, the `dx_getcursv()` function may return incorrect volume settings for the device. This is because the next `dx_open()` resets the volume tables to their default values.

**Errors**

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function `ATDV_LASTERR()` to obtain the error code or use `ATDV_ERRMSGP()` to obtain a descriptive error message. One of the following error codes may be returned:

- EDX_BADPARM
  - Invalid parameter

- EDX_BADPROD
  - Function not supported on this board
**dx_getcursv( ) — return the specified current volume settings**

EDX_SYSTEM

Error from operating system

### Example

Note that speed control is not supported in HMP.

```c
#include <stdio.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", 0 ) ) == -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 );
}
```

### See Also

- `dx_adjsv( )`
return the specified current volume settings — dx_getcursv( )

- dx_advdig( )
- dx_setsvmt( )
- dx_getsvmt( )
- dx_setsvcond( )
- dx_clrsvcond( )
- volume modification tables in the Voice API Programming Guide
- DX_SVMT data structure
**dx_getdig( )**

**Name:** int dx_getdig(chdev, tptp, digitp, mode)

**Inputs:**
- int chdev • valid 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 terminating null character) if successful (synchronous)
- -1 if failure

**Includes:** srllib.h
dxxxlib.h

**Category:** I/O

**Mode:** asynchronous or synchronous

---

**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).

**Note:** The channel must be idle, or the function will return an EDX_BUSY error.

**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. For a list of possible termination conditions, see **DV_TPT**, on page 279.
digitp | points to the User Digit Buffer structure, DV_DIGIT, where collected digits and their types are stored in arrays. For a list of digit types, see **DV_DIGIT**, on page 278.
mode | specifies whether to run **dx_getdig( )** asynchronously or synchronously. Specify one of the following:
  - EV_ASYNC – run asynchronously
  - EV_SYNC – run synchronously (default)
collect digits from a channel digit buffer — dx_getdig( )

The channel’s digit buffer contains 31 or more 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. The DG_MAXDIGS define in dxxxlib.h specifies the maximum number of digits that can be returned by a single call to dx_getdig( ).

Notes: 1. The maximum size of the digit buffer varies with the board type and technology. Multiple calls to dx_getdig( ) may be required to retrieve all digits in the digit buffer.
2. By default, after the maximum number of digits is received, all subsequent digits will be discarded.
3. Instead of getting digits from the DV_DIGIT structure using dx_getdig( ), an alternative method is to enable the DE_DIGITS call status transition event using dx_setevtmsk( ) and get them from the DX_EBLK event queue data (ev_data) using dx_getevt( ) or from the DX_CST call status transition data (cst_data) using sr_getevtdatap( ).

Asynchronous Operation

To run this function asynchronously, set the mode parameter to EV_ASYNC. In asynchronous mode, this function returns 0 to indicate success, and generates a TDX_GETDIG termination event to indicate completion. Use the Standard Runtime Library (SRL) Event Management functions to handle the termination event. For more information, see the Standard Runtime Library API Library Reference.

When operating asynchronously, ensure that the digit buffer stays in scope for the duration of the function.

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 character). Use ATDX_TERMMSK( ) to determine the reason for termination.

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 character terminator.

Cautions

- Global DPD is not supported (DG_DPD_ASCII is not available).
- Some MF digits use approximately the same frequencies as DTMF digits (see Section 6.1, “DTMF and MF Tone Specifications”, on page 317). 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.
- 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 both to adjust play and to terminate play, then the play adjustment will take priority.
dx_getdig() — collect digits from a channel digit buffer

- The dx_getdig() does not support terminating on a user-defined tone (GTD). Specifying DX_TONE in the DV_TPT tp_termno field has no effect on dx_getdig() termination and will be ignored.
- In a TDM bus configuration, when a caller on one voice board is routed in a conversation on an analog line with a caller on another voice board (analog inbound/outbound configuration) and either caller sends a DTMF digit, both voice channels will detect the DTMF digit if the corresponding voice channels are listening. This occurs because the network functionality of the voice board cannot be separated from the voice functionality in an analog connection between two callers. In this situation, you are not able to determine which caller sent the DTMF digit.

### Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function ATDV_LASTERR() to obtain the error code or use ATDV_ERRMSGP() to obtain a descriptive error message. One of the following error codes may be returned:

- EDX_BADPARM
  - Invalid parameter
- EDX_BADTPT
  - Invalid DV_TPT entry
- EDX_BUSY
  - Channel busy
- EDX_SYSTEM
  - Error from operating system

### Example 1

This example illustrates how to use dx_getdig() in synchronous mode.

```c
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.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 */
```
collect digits from a channel digit buffer — dx_getdig()

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 msec (10 msec resolution timer) */
tpt[1].tp_flags = TF_LCOFF|TF_10MS; /* level triggered, clear history, */
                        /* 10 msec resolution */

tpt[2].tp_type = IO_EOT;
tpt[2].tp_termno = DX_MAXTIME;       /* Function Time */
tpt[2].tp_length = 100;              /* 10 seconds (100 msec resolution timer) */
tpt[2].tp_flags = TF_MAXTIME;        /* Edge-triggered */

/* clear previously entered digits */
if (dx_clrdigbuf(chdev) == -1) {    /* process error */
    /* go to next state */
    .
    .
}

for (cnt=0; cnt < numdigs; cnt++) {
    printf("Digit received = %c, digit type = %d",
            digp.dg_value[cnt], digp.dg_type[cnt]);
}

Example 2

This example illustrates how to use \texttt{dx\_getdig()} in asynchronous mode.

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.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() */
```
dx_getdig( ) — collect digits from a channel digit buffer

* 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_clrpt(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 msec (10 msec resolution timer) */
tpt[1].tp_flags = TF_LCOFF|TF_10MS;  /* level triggered, clear
    * history, 10 msec resolution */

tpt[2].tp_type = IO_EOT;
tpt[2].tp_termno = DX_MAXTIME;     /* Function Time */
tpt[2].tp_length = 100;            /* 10 seconds (100 msec 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("Digit 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;
}

See Also

- dx_addtone( )
collect digits from a channel digit buffer — dx_getdig()
dx_GetDllVersion( ) — retrieve the voice DLL version number

**dx_GetDllVersion( )**

**Name:** dx_GetDllVersion (dwfileverp, dwprodverp)  

**Inputs:** LPDWORD dwfileverp  

• voice DLL version number  

LPDWORD dwprodverp  

• product version of this release  

**Returns:**  

0 if success  

-1 if failure  

**Includes:** srllib.h  

dxxxlib.h  

**Category:** Configuration  

**Mode:** synchronous

---

**Description**

Supported on Windows only. The dx_GetDllVersion( ) function returns the voice DLL version number for the file and product.

DLL Version Number functions return the file version number and product version number. The file version number specifies the version of the DLL. The product version number specifies the version of the software release that includes the DLL. Each function returns both version numbers in hexadecimal format. For example, if the DLL version is 4.13, the function returns it as 0x0004000D. If the product version is 11.3, the function returns it as 0x000bB0003. In each case, the high word represents the major number, and the low word represents the minor number.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dwfileverp</td>
<td>pointer to where to return file version information</td>
</tr>
<tr>
<td>dwprodverp</td>
<td>pointer to where to return product version information</td>
</tr>
</tbody>
</table>

**Cautions**

None.

**Errors**

None.

**Example**

*/$ dx_GetDllVersion( ) example $*/

#include <windows.h>
#include <srllib.h>
#include <dxxxlib.h>
int InitDevices( )
{
    DWORD dwfilever, dwprodver;
    /*****************************************************************************************/
* Initialize all the DLLs required. This will cause the DLLs to be
* loaded and entry points to be resolved. Entry points not resolved
* are set up to point to a default not implemented function in the
* 'C' library. If the DLL is not found all functions are resolved
* to not implemented.
************************************************************************/
if (sr_libinit(DLGC_MT) == -1) {
    /* Must be already loaded, only reason if sr_libinit() was already called */
}/
/* Call technology specific dx_libinit() functions to load voice DLL */
if (dx_libinit(DLGC_MT) == -1) {
    /* Must be already loaded, only reason if dx_libinit() was already called */
}/
*********************************************************************************
* Voice library initialized so all other voice functions may be called
* as normal. Display the version number of the DLL
*********************************************************************************/
dx_GetDllVersion(&dwfilever, &dwprodver);
printf("File Version for voice DLL is %d.%02d\n",
    HIWORD(dwfilever), LOWORD(dwfilever));
printf("Product Version for voice DLL is %d.%02d\n",
    HIWORD(dwprodver), LOWORD(dwprodver));
/* Now open all the voice devices */

■ See Also

• fx_GetDllVersion() in the Fax Software Reference
• sr_GetDllVersion() in the Standard Runtime Library API Library Reference
**dx_getevt( )**

**Name:** int dx_getevt(chdev, eblkp, timeout)

**Inputs:**
- int chdev • valid 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

**Mode:** synchronous

---

**Description**

The `dx_getevt()` function monitors channel events synchronously for possible call status transition events in conjunction with `dx_setevtmsk()` . The `dx_getevt()` function 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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <code>dx_open()</code></td>
</tr>
<tr>
<td>eblkp</td>
<td>points to the Event Block structure DX_EBLK, which contains the event that ended the blocking</td>
</tr>
<tr>
<td>timeout</td>
<td>specifies the maximum amount of time in seconds to wait for an event to occur. <code>timeout</code> can have one of the following values: • number of seconds – maximum length of time <code>dx_getevt()</code> will wait for an event. When the time specified has elapsed, the function will terminate and return an error. • -1 – <code>dx_getevt()</code> will block until an event occurs; it will not time out. • 0 – The function will return -1 immediately if no event is present.</td>
</tr>
</tbody>
</table>

**Notes:**

1. When the time specified in `timeout` expires, `dx_getevt()` will terminate and return an error. Use the Standard Attribute function `ATDV_LASTERR()` to determine the cause of the error, which in this case is EDX_TIMEOUT.

2. On Linux, an application can stop the `dx_getevt()` function from within a process or from another process.

   From within a process, a signal handler may issue a `dx_stopch()` with the handle for the device waiting in `dx_getevt()` . The `mode` parameter to `dx_stopch()` should be OR’ed with the EV_STOPGETEVT flag to stop `dx_getevt()` . In this case `dx_getevt()` will successfully return with the event DE_STOPGETEVT. The EV_STOPGETEVT flag influences `dx_getevt()` only.
It does not affect the existing functionality of \texttt{dx\_stopch}(). Specifically, if a different function besides \texttt{dx\_getevt}() is in progress when \texttt{dx\_stopch}() is called with the EV\_STOPGETEVT mode, that function will be stopped as usual. EV\_STOPGETEVT will be ignored if \texttt{dx\_getevt}() is not in progress.

From another process, the \texttt{dx\_getevt}() function may be stopped using the Inter-Process Event Communication mechanism. A process can receive an event from another process on the handle for the device waiting in \texttt{dx\_getevt}(). The event-sending process needs to open the same device and call the new function \texttt{dx\_sendevt}() with its device handle. The \texttt{dx\_getevt}() function in this case will return with the event specified in \texttt{dx\_sendevt}().

\section*{Cautions}

It is recommended that you enable only one process per channel. The event that \texttt{dx\_getevt}() is waiting for may change if another process sets a different event for that channel. See \texttt{dx\_setevtmstk}() for more information.

\section*{Errors}

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function \texttt{ATDV\_LASTERR}() to obtain the error code or use \texttt{ATDV\_ERRMSGP}() to obtain a descriptive error message. One of the following error codes may be returned:

- \texttt{EDX\_BADPARM}
  - Invalid parameter
- \texttt{EDX\_SYSTEM}
  - Error from operating system
- \texttt{EDX\_TIMEOUT}
  - Timeout time limit is reached

\section*{Example}

\begin{verbatim}
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.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_setevtmstk(chdev,DM\_RINGS|DM\_WINK) == -1) { /* process error */
  }

  ...
}
\end{verbatim}
dx_getevt( ) — monitor channel events synchronously

/* 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;
}

See Also

- dx_setevtsnk()
- DX_EBLK data structure
dx_getfeaturelist()  

**Name:** int dx_getfeaturelist(dev, feature_tablep)

**Inputs:**
- int dev • valid board or channel device handle
- FEATURE_TABLE *feature_tablep • pointer to features information structure

**Returns:**
- 0 on success
- -1 on error

**Includes:**
- srllib.h
- dxxxlib.h

**Category:** Configuration

**Mode:** synchronous

---

### Description

The `dx_getfeaturelist()` function returns information about the features supported on the device. This information is contained in the `FEATURE_TABLE` data structure.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| dev       | specifies the valid device handle obtained when a board (in the format dxxxBn) or channel (dxxxBnCm) was opened using `dx_open()`.

*Note:* Retrieving information for a channel device can be time-consuming as each channel is opened one by one. You can retrieve information for the board device instead. All channel devices belonging to the specific board device have the same features as the parent board.

| feature_tablep | specifies a pointer to the `FEATURE_TABLE` data structure which contains the bitmasks of various features supported such as data format for play/record, fax features, and more. For more information on this structure, see `FEATURE_TABLE`, on page 303.

### Cautions

- This function fails if an invalid device handle is specified.

### Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function `ATDV_LASTERR()` to obtain the error code or use `ATDV_ERRMSGP()` to obtain a descriptive error message. One of the following error codes may be returned:

- `EDX_BADPARM` Parameter error
- `EDX_SH_BADEXTTS` TDM bus time slot is not supported at current clock rate
dx_getfeaturelist( ) — retrieve feature support information for the device

EDX_SH_BADINDX
Invalid Switch Handler index number

EDX_SH_BADTYPE
Invalid local time slot channel type (voice, analog, etc.)

EDX_SH_CMDBLOCK
Blocking command is in progress

EDX_SH_LIBBSY
Switch Handler library is busy

EDX_SH_LIBNOTINIT
Switch Handler library is uninitialized

EDX_SH_MISSING
Switch Handler is not present

EDX_SH_NOCLK
Switch Handler clock fallback failed

EDX_SYSTEM
Error from operating system

Example

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

void main(int argc, char ** argv)
{
    char   chname[32] = "dxxxB1C1";
    int    dev;
    FEATURE_TABLE feature_table;

    if ((dev = dx_open(chname, 0)) == -1) {
        printf("Error opening ":", chname);  
        exit(1);
    }

    if (dx_getfeaturelist(dev, &feature_table) == -1) {
        printf("Error %d getting featurelist
", feature_table.ft_play & FT_ADPCM, chname);
        exit(2);
    }

    printf("ADPCM ");
    if (feature_table.ft_play & FT_PCM) {
        printf("PCM ");
    }

    printf("ALAW ");
    if (feature_table.ft_play & FT_ULAW) {
        printf("ULAW ");
    }
}
```
retrieve feature support information for the device — dx_getfeaturelist(

```c
if (feature_table.ft_play & FT_LINEAR) {
    printf("LINEAR ");
}

if (feature_table.ft_play & FT_ADSI) {
    printf("ADSI ");
}

if (feature_table.ft_play & FT_DRT6KHZ) {
    printf("DRT6KHZ ");
}

if (feature_table.ft_play & FT_DRT8KHZ) {
    printf("DRT8KHZ ");
}

if (feature_table.ft_play & FT_DRT11KHZ) {
    printf("DRT11KHZ");
}

printf("\n\n%5s: Record Features:-\n", chname);
if (feature_table.ft_record & FT_ADPCM) {
    printf("ADPCM ");
}

if (feature_table.ft_record & FT_PCM) {
    printf("PCM ");
}

if (feature_table.ft_record & FT_ALAW) {
    printf("ALAW ");
}

if (feature_table.ft_record & FT_ULAW) {
    printf("ULAW ");
}

if (feature_table.ft_record & FT_LINEAR) {
    printf("LINEAR ");
}

if (feature_table.ft_record & FT_ADSI) {
    printf("ADSI ");
}

if (feature_table.ft_record & FT_DRT6KHZ) {
    printf("DRT6KHZ ");
}

if (feature_table.ft_record & FT_DRT8KHZ) {
    printf("DRT8KHZ ");
}

if (feature_table.ft_record & FT_DRT11KHZ) {
    printf("DRT11KHZ");
}

printf("\n\n%5s: Tone Features:-\n", chname);
if (feature_table.ft_tone & FT_GTGENABLED) {
    printf("GTGENABLED ");
}

if (feature_table.ft_tone & FT_GTDENABLED) {
    printf("GTDENABLED ");
}
```
dx_getfeaturelist( ) — retrieve feature support information for the device

```c
if (feature_table.ft_tone & FT_CADENCE_TONE) {
    printf("CADENCE_TONE\n");
}

printf("\n\nE2P Board Configuration Features:-\n", chname);
if (feature_table.ft_e2p_brd_cfg & FT_DPD) {
    printf("DPD ");
}
if (feature_table.ft_e2p_brd_cfg & FT_SYNTELLECT) {
    printf("SYNTELLECT\n");
}

printf("\n\nFAX Features:-\n", chname);
if (feature_table.ft_fax & FT_FAX) {
    printf("FAX ");
}
if (feature_table.ft_fax & FT_VFX40) {
    printf("VFX40 ");
}
if (feature_table.ft_fax & FT_VFX40E) {
    printf("VFX40E ");
}
if (feature_table.ft_fax & FT_VFX40E_PLUS) {
    printf("VFX40E_PLUS\n");
}
if( (feature_table.ft_fax & FT_FAX_EXT_TBL)
    && (feature_table.ft_send & FT_SENDFAX_TXFILE_ASCII) )
    printf("SOFTFAX \n");

printf("\n\nFrontEnd Features:-\n", chname);
if (feature_table.ft_front_end & FT_ANALOG) {
    printf("ANALOG ");
}
if (feature_table.ft_front_end & FT_EARTH_RECALL) {
    printf("EARTH_RECALL\n");
}

printf("Miscellaneous Features:-\n", chname);
if (feature_table.ft_misc & FT_CALLERID) {
    printf("CALLERID\n");
}

printf("\n");
dx_close(dev);
```

### See Also

- `dx_getctinfo()`
**dx_getparm( )**

<table>
<thead>
<tr>
<th>Name:</th>
<th>int dx_getparm(dev, parm, valuep)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs:</td>
<td>int dev               • valid channel or board device handle</td>
</tr>
<tr>
<td></td>
<td>unsigned long parm    • parameter type to get value of</td>
</tr>
<tr>
<td></td>
<td>void *valuep          • pointer to variable for returning parameter value</td>
</tr>
<tr>
<td>Returns:</td>
<td>0 if success</td>
</tr>
<tr>
<td></td>
<td>-1 if failure</td>
</tr>
<tr>
<td>Includes:</td>
<td>srllib.h</td>
</tr>
<tr>
<td></td>
<td>dxxxlib.h</td>
</tr>
<tr>
<td>Category:</td>
<td>Configuration</td>
</tr>
<tr>
<td>Mode:</td>
<td>synchronous</td>
</tr>
</tbody>
</table>

### Description

The `dx_getparm( )` function returns the current parameter settings for an open device. This function returns the value of one parameter at a time.

A different set of parameters is available for board and channel devices. Board parameters affect all channels on the board. Channel parameters affects the specified channel only.

The channel must be idle (that is, no I/O function running) when calling `dx_getparm( )`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dev</td>
<td>specifies the valid device handle obtained when a board or channel was opened using <code>dx_open( )</code></td>
</tr>
</tbody>
</table>
**dx_getparm( ) — get the current parameter settings**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parm</td>
<td>Specifies the define for the parameter type whose value is to be returned in the variable pointed to by <code>valuep</code>. The voice device parameters allow you to query and control device-level information and settings related to the voice functionality. These parameters are described in the <code>dx_setparm( )</code> function description. Board parameter defines are described in Table 5, “Voice Board Parameters”, on page 238 and channel parameter defines are described in Table 6, “Voice Channel Parameters”, on page 238.</td>
</tr>
<tr>
<td>valuep</td>
<td>Points to the variable where the value of the parameter specified in <code>parm</code> should be returned. <strong>Note:</strong> You must use a <code>void*</code> cast on the returned parameter value, as demonstrated in the Example section code for this function. <strong>Note:</strong> <code>valuep</code> should point to a variable large enough to hold the value of the parameter. The size of a parameter is encoded in the define for the parameter. The defines for parameter sizes are <code>PM_SHORT</code>, <code>PM_BYTE</code>, <code>PM_INT</code>, <code>PM_LONG</code>, <code>PM_FLSTR</code> (fixed length string), and <code>PM_VLSTR</code> (variable length string). Most parameters are of type short.</td>
</tr>
</tbody>
</table>

### Cautions

Clear the variable in which the parameter value is returned prior to calling `dx_getparm( )`, as illustrated in the Example section. The variable whose address is passed to should be of a size sufficient to hold the value of the parameter.

### Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function `ATDV_LASTERR( )` to obtain the error code or use `ATDV_ERRMSGP( )` to obtain a descriptive error message. One of the following error codes may be returned:

- **EDX_BADPARM**
  - Invalid parameter
- **EDX_BUSY**
  - Channel is busy (when channel device handle is specified) or first channel is busy (when board device handle is specified)
- **EDX_SYSTEM**
  - Error from operating system

### Example

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

main()
{
    int bddev;
    unsigned short parmval;
```
get the current parameter settings — dx_getparm()

/* 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);
.
.
.

■ See Also

• dx_setparm()
**dx_GetStreamInfo( ) — retrieve information about the circular stream buffer**

**dx_GetStreamInfo( )**

**Name:** int dx_GetStreamInfo(hBuffer, &StreamStatStruct)

**Inputs:**
- int hBuffer
- DX_STREAMSTAT StreamStatStruct

**Returns:**
- 0 if successful
- -1 if failure

**Includes:** srllib.h
dxxxlib.h

**Category:** streaming to board

**Mode:** synchronous

---

**Description**

The `dx_GetStreamInfo( )` function populates the stream status structure with the current status information about the circular stream buffer handle passed into it. The data returned is a snapshot of the status at the time `dx_GetStreamInfo( )` is called.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hBuffer</td>
<td>specifies the circular stream buffer handle</td>
</tr>
<tr>
<td>StreamStatStruct</td>
<td>specifies a pointer to the DX_STREAMSTAT data structure. For more information on this structure, see DX_STREAMSTAT, on page 293.</td>
</tr>
</tbody>
</table>

**Cautions**

None.

**Errors**

Unlike other voice API library functions, the streaming to board functions do not use SRL device handles. Therefore, `ATDV_LASTERR( )` and `ATDV_ERRMSGP( )` cannot be used to retrieve error codes and error descriptions.

**Example**

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

main()
{
    int nBuffSize = 32768;
    int hBuffer = -1;
    DX_STREAMSTAT streamStat;
    
    if ((hBuffer = dx_OpenStreamBuffer(nBuffSize)) < 0)
    {
```
retrieve information about the circular stream buffer — dx_GetStreamInfo( )

```c
printf("Error opening stream buffer \n" );
} if (dx_GetStreamInfo(hBuffer, &streamStat) < 0)
{ printf("Error getting stream buffer info \n");
} else 
{ printf("version=%d, 
bytesIn=%d, 
bytesOut=%d, 
headPointer=%d, 
tailPointer=%d, 
currentState=%d, 
numberOfBufferUnderruns=%d, 
numberOfBufferOverruns=%d, 
BufferSize=%d, 
spaceAvailable=%d, 
highWaterMark=%d, 
lowWaterMark=%d \n", 
streamStat.tailPointer,streamStat.currentState,streamStat.numberOfBufferUnderruns, 
streamStat.highWaterMark,streamStat.lowWaterMark);
} if (dx_CloseStreamBuffer(hBuffer) < 0)
{ printf("Error closing stream buffer \n");
}
}

■ See Also

- dx_OpenStreamBuffer( )
```
**dx_getsvmt( ) — return the current volume modification table**

### dx_getsvmt( )

**Name:** int dx_getsvmt(chdev, tabletype, svmtp )

**Inputs:**
- int chdev • valid channel device handle
- unsigned short tabletype • type of table to retrieve (volume)
- DX_SVMT * svmtp • pointer to volume modification table structure to retrieve

**Returns:**
- 0 if success
- -1 if failure

**Includes:** srllib.h
dxxxlib.h

**Category:** Volume

**Mode:** synchronous

---

#### Description

The `dx_getsvmt( )` function returns the current volume modification table to the DX_SVMT structure. On HMP, speed adjustment is not supported.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <code>dx_open( )</code></td>
</tr>
<tr>
<td>tabletype</td>
<td>specifies the volume modification table:</td>
</tr>
<tr>
<td></td>
<td>• SV_VOLUMETBL – retrieve the volume modification table values</td>
</tr>
<tr>
<td>svmtp</td>
<td>points to the DX_SVMT structure that contains the volume modification table entries</td>
</tr>
</tbody>
</table>

#### Cautions

None.

#### Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function `ATDV_LASTERR( )` to obtain the error code or use `ATDV_ERRMSGP( )` to obtain a descriptive error message. One of the following error codes may be returned:

- EDX_BADPARM
  - Invalid parameter
- EDX_BADPROD
  - Function not supported on this board
- EDX_SPDVOL
  - Must specify either SV_SPEEDTBL or SV_VOLUMETBL
EDX_SYSTEM

Error from operating system

Example

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

/*
 * Global Variables
 */

main()
{
    DX_SVMT svmt;
    int dxxxdev, index;

    /*
     * Open the Voice Channel Device and Enable a Handler
     */
    if ( ( dxxxdev = dx_open( "dxxxB1C1", 0 ) ) == -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("Lasterror = %d  Err Msg = %s
",
                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);
}
```
dx_getsvmt() — return the current volume modification table

■ See Also

- dx_addvoldig()
- dx_adjsv()
- dx_clrsvcond()
- dx_getcursv()
- dx_setsvcond()
- dx_setsvmt()
- volume modification tables in Voice API Programming Guide
- DX_SVMT data structure
**dx_getxmitslot()**

**Name:** int dx_getxmitslot(chdev, sc_tsinfop)

**Inputs:**
- int chdev • valid channel device handle
- SC_TSINFO *sc_tsinfop • pointer to TDM bus time slot information structure

**Returns:**
- 0 on success
- -1 on error

**Includes:**
- srllib.h
- dxxxlilb.h

**Category:** TDM routing

**Mode:** synchronous

---

### Description

The **dx_getxmitslot()** function returns the time division multiplexing (TDM) bus time slot number of the voice transmit channel. The TDM bus time slot information is contained in an SC_TSINFO structure that includes the number of the TDM bus time slot connected to the voice transmit channel. For more information on this structure, see **SC_TSINFO**, on page 306.

**Note:**
TDM bus convenience function **nr_scroute()** includes **dx_getxmitslot()** functionality.

**Parameter** | **Description**
--- | ---
chdev | specifies the voice channel device handle obtained when the channel was opened using **dx_open()**
sc_tsinfop | specifies a pointer to the data structure **SC_TSINFO**

A voice channel on a TDM bus-based board can transmit on only one TDM bus time slot.

### Cautions

- This function fails when an invalid channel device handle is specified.

### Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function **ATDV_LASTERR()** to obtain the error code or use **ATDV_ERRMSGP()** to obtain a descriptive error message. One of the following error codes may be returned:

- **EDX_BADPARM**
  
  Parameter error

- **EDX_SH_BADCMD**
  
  Command is not supported in current bus configuration

- **EDX_SH_BADINDX**
  
  Invalid Switch Handler index number
dx_getxmitslot( ) — get TDM bus time slot number of voice transmit channel

EDX_SH_BADLCLTS
Invalid channel number

EDX_SH_BADMODE
Function is not supported in current bus configuration

EDX_SH_BADTYPE
Invalid channel type (voice, analog, etc.)

EDX_SH_CMDBLOCK
Blocking command is in progress

EDX_SH_LCLDSCNCT
Channel is already disconnected from TDM bus

EDX_SH_LIBBSY
Switch Handler library is busy

EDX_SH_LIBNOTINIT
Switch Handler library is uninitialized

EDX_SH_MISSING
Switch Handler is not present

EDX_SH_NOCLK
Switch Handler clock fallback failed

EDX_SYSTEM
Error from operating system

Example

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

main()
{
    int      chdev;       /* Channel device handle */
    SC_TSINFO sc_tsinfo;  /* Time slot information structure */
    long     scts;        /* TDM bus time slot */

    /* Open board 1 channel 1 devices */
    if ((chdev = dx_open("dxxxB1C1", 0)) == -1) {
        /* process error */
    }

    /* Fill in the TDM bus time slot information */
    sc_tsinfo.sc_nums = 1;
    sc_tsinfo.sc_tarray = &scts;

    /* Get TDM bus time slot connected to transmit of voice channel 1 on board ...1 */
    if (dx_getxmitslot(chdev, &sc_tsinfo) == -1) {
        printf("Error message = %s", ATDV_ERRMSGP(chdev));
        exit(1);
    }
    printf("%s transmitting on TDM bus time slot %d", ATDV_NAMEP(chdev), scts);
    return(0);
}
```

See Also

- dx_listen()
get TDM bus time slot number of voice transmit channel — dx_getxmitslot()

- `fx_listen()` in the *Fax Software Reference*
**dx_libinit( )**

**Name:** dx_libinit( flags )

**Inputs:** unsigned short flags

* specifies the programming model

**Returns:**

0 if success

-1 if failure

**Includes:** srllib.h
dxxxlib.h

**Category:** Configuration

**Mode:** synchronous

---

## Description

Supported on Windows only. The `dx_libinit( )` function initializes the voice library DLL by loading and resolving all entry points in `LIBDXXMT.DLL`.

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| flags     | This flag has two possible values:  
  - DLGC_MT – specify if using a multi-threaded or window callback model  
  - DLGC_ST – specify if using the single-threaded model |

## Cautions

The `sr_libinit( )` function must be called prior to using the `dx_libinit( )` function.

## Errors

The `dx_libinit( )` function fails if the library has already been initialized. For example, if you try to make a second call to `sr_libinit( )`, it fails.

## Example

```c
/*$ dx_libinit( ) example $*/

#include <windows.h>
#include <srllib.h>
#include <dxxxlib.h>
int InitDevices( )
{
    DWORD dwfilever, dwprodver;
    /**************************************************************************
    * Initialize all the DLLs required. This will cause the DLLs to be
    * loaded and entry points to be resolved. Entry points not resolved
    * are set up to point to a default not implemented function in the
    * 'C' library. If the DLL is not found all functions are resolved
    * to not implemented.
    ***************************************************************************/
```
if (sr_libinit(DLGC_MT) == -1) {  
    /* Must be already loaded, only reason if sr_libinit( ) was already called */
}

/* Call technology specific dx_libinit( ) functions to load voice DLL */
if (dx_libinit(DLGC_MT) == -1) {
    /* Must be already loaded, only reason if dx_libinit( ) was already called */
}

/*********************************************************************************
* Voice library initialized so all other voice functions may be called
* as normal. Display the version number of the DLL
**********************************************************************************/
dx_GetDllVersion(&dwfilever, &dwprodver);
printf("File Version for voice DLL is %d.%02d\n",
    HIWORD(dwfilever), LOWORD(dwfilever));
printf("Product Version for voice DLL is %d.%02d\n",
    HIWORD(dwprodver), LOWORD(dwprodver));
/* Now open all the voice devices */

See Also

- fx_libinit() in the Fax Software Reference
- sr_libinit() in the Standard Runtime Library API Library Reference
**dx_listen( )**

*Description*

The `dx_listen()` function connects a voice listen channel to a TDM bus time slot. This function uses the information stored in the `SC_TSINFO` data structure to connect the receive voice (listen) channel to a TDM bus time slot. This function sets up a half-duplex connection. For a full-duplex connection, the receive (listen) channel of the other device must be connected to the voice transmit channel.

*Note:* TDM bus convenience function `nr_scroute()` includes `dx_listen()` functionality.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>chdev</code></td>
<td>specifies the voice channel device handle obtained when the channel was opened using <code>dx_open()</code></td>
</tr>
<tr>
<td><code>sc_tsinfop</code></td>
<td>specifies a pointer to the <code>SC_TSINFO</code> structure</td>
</tr>
</tbody>
</table>

Upon return from the `dx_listen()` function, the voice receive channel will be connected to the TDM bus time slot.

Although multiple voice channels may listen (be connected) to the same TDM bus time slot, the receive of a voice channel can connect to only one TDM bus time slot.

*Cautions*

- This function fails when an invalid channel device handle is specified or when an invalid TDM bus time slot number is specified.
Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function ATDV_LASTERR() to obtain the error code or use ATDV_ERRMSGP() to obtain a descriptive error message. One of the following error codes may be returned:

**EDX_BADPARM**
Parameter error

**EDX_SH_BADCMD**
Command is not supported in current bus configuration

**EDX_SSH_BADEXTTS**
TDM bus time slot is not supported at current clock rate

**EDX_SH_BADINDX**
Invalid Switch Handler index number

**EDX_SH_BADLCLTS**
Invalid channel number

**EDX_SH_BADMODE**
Function not supported in current bus configuration

**EDX_SH_CMDBLOCK**
Blocking command is in progress

**EDX_SH_LCLTSCNCT**
Channel is already connected to TDM bus

**EDX_SH_LIBBSY**
Switch Handler library busy

**EDX_SH_LIBNOTINIT**
Switch Handler library uninitialized

**EDX_SH_MISSING**
Switch Handler is not present

**EDX_SH_NOCLK**
Switch Handler clock fallback failed

**EDX_SYSTEM**
Error from operating system

Example

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

main()
{
    int        chdev;                 /* Channel device handle */
    SC_TSINFO  sc_tsinfo;             /* Time slot information structure */
    long       scts;                  /* TDM bus time slot */

    /* Open board 1 channel 1 device */
    if ((chdev = dx_open("dxxxB1C1", 0)) == -1) {
        /* process error */
    }

    /* Connect channel to TDM bus */
    dx_listen(chdev, scts, sc_tsinfo);
}
```
dx_listen( ) — connect a voice listen channel to TDM bus time slot

/* Fill in the TDM bus time slot information */
sc_tsinfo.sc_numts = 1;
sc_tsinfo.sc_tsarrayp = &scts;

/* Get TDM bus time slot connected to transmit of analog channel 1 on board 1 */
if (ag_getxmitslot(chdev, &sc_tsinfo) == -1) {
   printf("Error message = %s", ATDV_ERRMSGP(chdev));
   exit(1);
}

/* Connect the receive of voice channel 1 on board 1 to TDM bus time slot */
if (dx_listen(chdev, &sc_tsinfo) == -1) {
   printf("Error message = %s", ATDV_ERRMSGP(chdev));
   exit(1);
}

See Also

- fx_getxmislot( ) in the Fax Software Reference
- dx_unlisten( )
dx_open( )

**Name:** int dx_open(namep, oflags)

**Inputs:**
- char *namep  • pointer to device name to open

**Returns:**
- >0 to indicate valid device handle if successful
- -1 if failure

**Includes:**
- srllib.h
- dxxxlib.h

**Category:** Device Management

**Mode:** synchronous

---

### Description

The **dx_open( )** function opens a voice board device or channel device, and returns a unique device handle to identify the device. All subsequent references to the opened device must be made using the handle until the device is closed.

The device handle returned by this function is defined by Intel. It is not a standard operating system file descriptor. Any attempts to use operating system commands such as `read()`, `write()`, or `ioctl()` will produce unexpected results.

On Windows, by default, the maximum number of times you can simultaneously open the same channel in your application is set to 30 in the Windows Registry.

Use Standard Runtime Library device mapper functions to return information about the structure of the system. This device information is used as input in the **dx_open( )** function. For more information on these functions, see the *Standard Runtime Library API Library Reference*.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>namep</td>
<td>points to an ASCIIZ string that contains the name of the valid device. These valid devices can be either boards or channels. The standard board device naming convention for voice devices is: dxxxB1, dxxxB2, and so on. The standard channel device naming convention for voice devices is: dxxxB1C1, dxxxB1C2, and so on.</td>
</tr>
<tr>
<td>oflags</td>
<td>reserved for future use. Set this parameter to 0.</td>
</tr>
</tbody>
</table>

### Cautions

- Do not use the operating system `open()` function to open a voice device. Unpredictable results will occur.
dx_open( ) — open a voice device and return a unique device handle

- In applications that spawn child processes from a parent process, the device handle is not inheritable by the child process. Make sure devices are opened in the child process.
- Two processes cannot open and access the same device.
- In Linux, if STDOUT has been closed and an Intel® Dialogic® device is then opened, the device may get the same handle as STDOUT. Subsequent calls to printf( ) (which goes to STDOUT) may cause a kernel panic.
- On Springware boards in Linux, when developing an application for a large system (more than 350 devices), the application should open all the voice devices (board and/or channel) first, and then open all other devices.

### Errors

In Windows, if this function returns -1 to indicate failure, a system error has occurred; use dx_fileerrno( ) to obtain the system error value. Refer to the dx_fileerrno( ) function for a list of the possible system error values.

In Linux, if this function returns -1 to indicate failure, check errno for one of the following reasons:

- EBADF  
  Invalid file descriptor
- EINTR  
  A signal was caught
- EINVAL  
  Invalid argument
- EIO  
  Error during a Linux STREAMS open

This function will fail and return -1 if:
- The device name is invalid.
- A hardware error on the board or channel is discovered.

### Example

This example illustrates how to open a channel device.

```c
#include "srllib.h"
#include "dxxxlib.h"

main()
{
    int chdev; /* channel descriptor */
    
    /* Open Channel */
    if ((chdev = dx_open("dxxxB1C1",0)) == -1) {
        /* process error */
    }
    
}```
open a voice device and return a unique device handle — dx_open()

- See Also
  - dx_close()
dx_OpenStreamBuffer( ) — create and initialize a circular stream buffer

**dx_OpenStreamBuffer( )**

- **Name:** int dx_OpenStreamBuffer(BuffSize)
- **Inputs:** int BuffSize  
  - size in bytes of circular stream buffer
- **Returns:** stream buffer handle if successful  
  -1 if failure
- **Includes:** srllib.h  
  dxxxlib.h
- **Category:** streaming to board
- **Mode:** synchronous

## Description

The **dx_OpenStreamBuffer( )** function allocates and initializes a circular stream buffer for streaming to a voice device.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BuffSize</td>
<td>specifies the size in bytes of the circular stream buffer to allocate</td>
</tr>
</tbody>
</table>

You can create as many stream buffers as needed on a channel; however, you are limited by the amount of memory on the system. You can use more than one stream buffer per play via the DX_IOTT structure. In this case, specify that the data ends in one buffer using the STREAM_EOD flag so that the play can process the next DX_IOTT structure in the chain. For more information about using the streaming to board feature, see the *Voice API Programming Guide*.

This function initializes the circular stream buffer to the same initial state as **dx_ResetStreamBuffer( )**.

## Cautions

The buffer identified by the circular stream buffer handle cannot be used by multiple channels for the play operation.

## Errors

This function fails with -1 error if there is not enough system memory available to process this request.

Unlike other voice API library functions, the streaming to board functions do not use SRL device handles. Therefore, **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** cannot be used to retrieve error codes and error descriptions.
create and initialize a circular stream buffer — dx_OpenStreamBuffer( )

**Example**

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

main()
{
    int nBuffSize = 32768, vDev = 0;
    int hBuffer = -1;
    char pData[1024];
    DX_IOTT iott;
    DV_TPT ptpt;

    if ((hBuffer = dx_OpenStreamBuffer(nBuffSize)) < 0)
    {
        printf("Error opening stream buffer \n");
        exit(1);
    }

    if ((vDev = dx_open("dxxxB1C1", 0)) < 0)
    {
        printf("Error opening voice device\n");
        exit(2);
    }

    iott.io_type = IO_STREAM|IO_EOT;
    iott.io_bufp = 0;
    iott.io_offset = 0;
    iott.io_length = -1; /* play until STREAM_EOD */
    iott.io_fhandle = hBuffer;

    dx_clrtpt(&ptpt,1);
    tpt.tp_type = IO_EOT;
    tpt.tp_termno = DX_MAXDTMF;
    tpt.tp_length = 1;
    tpt.tp_flags = TF_MAXDTMF;

    if (dx_play(vDev, &iott, &tpt, EV_ASYNC) < 0)
    {
        printf("Error in dx_play() %d", ATDV_LASTERR(vDev));
    }

    /* Repeat the following until all data is streamed */

    if ((dx_PutStreamData(hBuffer, pData, 1024, STREAM_CONT) < 0)
    {
        printf("Error in dx_PutStreamData \n");
        exit(3);
    }

    /* Wait for TDX_PLAY event and other events as appropriate */

    if (dx_CloseStreamBuffer(hBuffer) < 0)
    {
        printf("Error closing stream buffer \n");
    }
}
```

**See Also**

- `dx_CloseStreamBuffer( )`
- `dx_SetWaterMark( )`
dx_play( ) — play recorded voice data

The dx_play( ) function plays recorded voice data, which may come from any combination of data files, memory, or custom devices.

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.

To specify format information about the data to be played, including file format, data encoding, sampling rate, and bits per sample, use dx_playiottdata( ).

Parameter | Description |
--- | --- |
chdev | Specifies the valid channel device handle obtained when the channel was opened using dx_open( ). |
riottp | Points to the I/O Transfer Table Structure, DX_IOTT, which specifies the order of playback and the location of voice data. See DX_IOTT, on page 290, for information about the data structure. |
tptp | Points to the Termination Parameter Table structure, DV_TPT, which specifies termination conditions for playing. For more information on this structure, see DV_TPT, on page 279. |

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.
**play recorded voice data — dx_play()**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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 3). Choose one only:  
  - EV_ASYNC – run asynchronously  
  - EV_SYNC – run synchronously (default)  
Choose one or more of the following:  
  - 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  
  - PM_ALAW – play using A-law  
  - PM_SR6 – play using 6 kHz sampling rate (6000 samples per second)  
  - PM_SR8 – play using 8 kHz sampling rate (8000 samples per second)  
  - 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. |

Notes:  
1. The rate specified in the last play function applies 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 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 6 kHz.
3. Make sure data is played using the same encoding algorithm and sampling rate used when the data was recorded.

Table 3 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 braces, `{}`, are default settings and do not have to be specified.

**Table 3. Play Mode Selections**

<table>
<thead>
<tr>
<th>Feature(s)</th>
<th>ADPCM (24 kbps)</th>
<th>ADPCM (32 kbps)</th>
<th>PCM (48 kbps)</th>
<th>PCM (64 kbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tone</td>
<td>PM_TONE</td>
<td>PM_TONE</td>
<td>PM_TONE</td>
<td>PM_TONE</td>
</tr>
<tr>
<td></td>
<td>PM_SR6</td>
<td>PM_SR8</td>
<td>PM_ALAW*</td>
<td>PM_ALAW*</td>
</tr>
<tr>
<td></td>
<td>(MD_ADPCM)</td>
<td>(MD_ADPCM)</td>
<td>PM_SR6</td>
<td>PM_SR8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MD_PCM</td>
<td>MD_PCM</td>
</tr>
</tbody>
</table>
dx_play( ) — play recorded voice data

**Table 3. Play Mode Selections**

<table>
<thead>
<tr>
<th>Feature(s)</th>
<th>ADPCM (24 kbps)</th>
<th>ADPCM (32 kbps)</th>
<th>PCM (48 kbps)</th>
<th>PCM (64 kbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Tone</td>
<td>PM_SR6 (MD_ADPCM)</td>
<td>PM_SR8 (MD_ADPCM)</td>
<td>PM_SR6 MD_PCM</td>
<td>PM_SR8 MD_PCM</td>
</tr>
</tbody>
</table>

{} = Default modes.
* = Select if file was encoded using A-law

### Asynchronous Operation

To run this function asynchronously, set the **mode** field to EV_ASYNC. When running asynchronously, this function returns 0 to indicate it has initiated successfully, and generates a TDX_PLAY termination event 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.

Termination of asynchronous play is indicated by a TDX_PLAY event. Use the Standard Runtime Library (SRL) Event Management functions to handle the termination event.

After **dx_play( )** terminates, the current channel’s status information, including the reason for termination, can be accessed using extended attribute functions. Use the **ATDX_TERMMSK( )** function to determine the reason for termination.

**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 returns 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.

### Cautions

- Whenever **dx_play( )** is called, its volume is based on the most recent adjustment made using **dx_adjsv( )** or **dx_setsvcond( )**.
- If A-law encoding is selected (RM_ALAW), the A-law parameter must be passed each time the play function is called or the setting will return to mu-law (the default).
play recorded voice data — *dx_play*( )

Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function
*ATDV_LASTERR*( ) to obtain the error code or use *ATDV_ERRMSGP*( ) to obtain a descriptive
error message. One of the following error codes may be returned:

EDX_BADPARM
Invalid parameter

EDX_BADIOTT
Invalid DX_IOTT entry

EDX_BADTPT
Invalid DV_TPT entry

EDX_BUSY
Busy executing I/O function

EDX_SYSTEM
Error from operating system

Example 1

This example illustrates how to use *dx_play*( ) in synchronous mode.

```c
/* Play a voice file. Terminate on receiving 4 digits or at end of file */
#include <fcntl.h>
#include <srllib.h>
#include <dxxxlib.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 */
```
dx_play( ) — play recorded voice data

/* 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

This example illustrates how to use dx_play( ) in asynchronous mode.

#include <stdio.h>
#include <srllib.h>
#include <dxlib.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 */
    
    /* For Windows applications: 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 Linux applications, open the vox file to play; the file descriptor will be used
     * by all channels.
     */
    if ((voxfd = open("prompt.vox", O_RDONLY)) == -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., dxxxBlC1, dxxxBlC2,... */ /* 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; /* Two 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].io_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_getevtdiag()); 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: %s\n", term); } /* Kick off next function in the state machine model. */ /* return 0; */ /* See Also */ /* dx_play() */
dx_play() — play recorded voice data

- dx_playiottdata()
- dx_playvox()
- dx_setparm(), dx_getparm()
- dx_adjsv()
- dx_setsvcond()
- DX_IOTT data structure (to identify source or destination of the voice data)
- event management functions in *Standard Runtime Library API Library Reference*
- ATDX TERMMSK()
- DV_TPT data structure (to specify a termination condition)
- dx_setuio()
dx_playf( )

**Name:** int dx_playf(chdev, fnamep, tptp, mode)

**Inputs:**
- int chdev • valid 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:** I/O Convenience

**Mode:** synchronous

---

**Description**

dx_playf( ) is a convenience function that synchronously plays voice data from a single file.

Calling dx_playf( ) is the same as calling dx_play( ) and specifying a single file entry in the DX_IOTT structure. Using dx_playf( ) is more convenient for single file playback, because you do not have to set up a DX_IOTT structure for one file, and the application does not need to open the file. The dx_playf( ) function opens and closes the file specified by fnamep.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using dx_open( )</td>
</tr>
<tr>
<td>fnamep</td>
<td>points to name of file from which voice data will be played</td>
</tr>
<tr>
<td>tptp</td>
<td>points to the Termination Parameter Table structure, DV_TPT, which specifies termination conditions for playing. For more information on this structure, see DV_TPT, on page 279.</td>
</tr>
<tr>
<td>mode</td>
<td>specifies the mode. This function supports EV_SYNC (synchronous mode) only.</td>
</tr>
</tbody>
</table>

**Cautions**

None.
Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function `ATDV_LASTERR( )` to obtain the error code or use `ATDV_ERRMSGP( )` to obtain a descriptive error message. One of the following error codes may be returned:

- **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**
  - Error from operating system

Source Code

```c
/***************************************************************************
* 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 = open(filep,O_RDONLY)) == -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;
```
synchronously play voice data — dx_playf( )

```c
rval = dx_play(devd, &iott, tptp, mode);
if (close(iott.io_fhandle) == -1) {
    return -1;
}
return rval;
```

**Example**

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

int 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 */
    }
}
```

**See Also**

- dx_play( )
- dx_playiottdata( )
- dx_playvox( )
- dx_setparm( ), dx_getparm( )
- dx_adjsv() (for volume control)
- dx_setsvcond() (for volume control)
- ATDX_TERMMSK()
- DV_TPT data structure (to specify a termination condition)
**dx_playiottdata( )**

**Name:** short $dx_{\_}\text{playiottdata}(\text{chdev, iottp, tptp, xpbp, mode})$

**Inputs:**
- `int chdev`: valid channel device handle
- `DX_{\_}\text{IOTT} *iottp`: pointer to I/O Transfer Table
- `DV_{\_}\text{TPT} *tptp`: pointer to Termination Parameter Block
- `DX_{\_}\text{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

**Mode:** asynchronous or synchronous

---

**Description**

The `dx_playiottdata( )` function plays back recorded voice data, which may come from any combination of data files, memory, or custom devices.

The file format for the files to be played is specified in the `wFileFormat` field of the `DX_{\_}\text{XPB}`. Other fields in the `DX_{\_}\text{XPB}` describe the data format. For files that include data format information (for example, WAV files), these other fields are ignored.

The `dx_playiottdata( )` function is similar to `dx_{\_}\text{play}( )`, but takes an extra parameter, `xpbp`, which allows you to specify format information about the data to be played. This includes file format, data encoding, sampling rate, and bits per sample.
play back recorded voice data from multiple sources — dx_playiottdata()

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>Specifies the valid channel device handle obtained when the channel was opened using <code>dx_open()</code></td>
</tr>
<tr>
<td>iottp</td>
<td>Points to the I/O Transfer Table structure, DX_IOTT, which specifies the order of playback and the location of voice data. See DX_IOTT, on page 290, for information about the data structure. The order of playback and the location of the voice data is specified in an array of DX_IOTT structures pointed to by <code>iottp</code></td>
</tr>
<tr>
<td>tptp</td>
<td>Points to the Termination Parameter Table structure, DV_TPT, which specifies termination conditions for this function. For more information on termination conditions, see DV_TPT, on page 279.</td>
</tr>
<tr>
<td>xpbp</td>
<td>Points to the I/O Transfer Parameter Block, DX_XPB. The file format for the files to be played is specified in the <code>wFileFormat</code> field of the DX_XPB. Other fields in the DX_XPB describe the data format. For more information about this structure, see the description for DX_XPB, on page 301. For information about supported data formats, see the Voice API Programming Guide.</td>
</tr>
<tr>
<td>mode</td>
<td>Specifies the play mode and synchronous/asynchronous mode. For a list of all valid values, see the <code>dx_play()</code> function description. - <code>PM_TONE</code> – play 200 msec audible tone - <code>EV_SYNC</code> – synchronous mode - <code>EV_ASYNC</code> – asynchronous mode</td>
</tr>
</tbody>
</table>

■ Cautions

- 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.
- All files specified in the DX_IOTT table must contain data of the type described in DX_XPB.
- When playing or recording VOX files, the data format is specified in DX_XPB rather than through the mode argument of this function.
- The DX_IOTT data area must remain in scope for the duration of the function if running asynchronously.
- The DX_XPB data area must remain in scope for the duration of the function if running asynchronously.
- Playing an empty WAVE file results in an invalid offset error. To play a silent WAVE file successfully, ensure that there is at least one byte of silence data (0xFF) in the payload.
- When set to play WAVE files, all other fields in the DX_XPB are ignored.
- When set to play WAVE files, this function will fail if an unsupported data format is attempted to be played. For information about supported data formats, see the description for DX_XPB, on page 301 and the Voice API Programming Guide.

■ Errors

In asynchronous mode, the function returns immediately and a TDX_PLAY event is queued upon completion. Check `ATDX_TERMMSK()` for the termination reason. If a failure occurs during
playback, then a TDX_ERROR event will be queued. Use ATDV_LASTERR() to determine the reason for the error. In some limited cases such as when invalid arguments are passed to the library, the function may fail before starting the play. In such cases, the function returns -1 immediately to indicate failure and no event is queued.

In synchronous mode, if this function returns -1 to indicate failure, use the Standard Runtime Library (SRL) Standard Attribute function ATDV_LASTERR() to obtain the error code or use ATDV_ERRMSGP() to obtain a descriptive error message. One of the following error codes may be returned:

- **EDX_BADIOTT**
  - Invalid DX_IOTT setting
- **EDX_BADWAVFILE**
  - Invalid WAVE file
- **EDX_BUSY**
  - Channel is busy
- **EDX_SH_BADCMD**
  - Unsupported command or WAVE file format
- **EDX_SYSTEM**
  - Error from operating system
- **EDX_XPBPARM**
  - Invalid DX_XPB setting

### Example

This example illustrates how to play back a VOX file in synchronous mode.

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

main()
{
  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");
    return(1);
  }
  tpt.tp_type   = IO_EOT;
  tpt.tp_termno = DX_MAXDTMF;
  tpt.tp_length = 1;
  tpt.tp_flags  = TF_MAXDTMF;
  dx_playiottdata(chdev, fd, iott, tpt, xpb);

  .
  .
  .

  /* 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;
}
```
play back recorded voice data from multiple sources — dx_playiottdata()

/* For Windows applications: open VOX file to play */
if ((fd = dx_fileopen("HELLO.VOX",O_RDONLY|O_BINARY)) == -1) {
    printf("File open error\n");
    exit(2);
}

/* For Linux applications: Open VOX file to play */
if ((fd = open("HELLO.VOX",O_RDONLY)) == -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_type = 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.wBitsPerSample = 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);
}

See Also

- dx_play()
- dx_playf()
- dx_playwav()
- dx_playvox()
- dx_setuio()
dx_playtone() — play tone defined by TN_GEN structure

dx_playtone()

**Name:** int dx_playtone(chdev, tngenp, tptp, mode)

**Inputs:**
- int chdev • valid channel device handle
- TN_GEN *tngenp • pointer to the Tone Generation template structure
- DV_TPT *tptp • pointer to a Termination Parameter Table structure
- int mode • asynchronous/synchronous

**Returns:**
- 0 if success
- -1 if failure

**Includes:** srllib.h
dxxxlib.h

**Category:** Global Tone Generation

**Mode:** asynchronous or synchronous

---

**Description**

The `dx_playtone()` function plays tones defined by the TN_GEN structure, which defines the frequency, amplitude, and duration of a single- or dual-frequency tone to be played.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <code>dx_open()</code></td>
</tr>
<tr>
<td>tngenp</td>
<td>points to the TN_GEN structure, which defines the frequency, amplitude, and duration of a single- or dual-frequency tone. For more information, see TN_GEN, on page 307. You can use the <code>dx_bldtngen()</code> function to set up the structure.</td>
</tr>
<tr>
<td>tptp</td>
<td>points to the DV_TPT data structure, which specifies a terminating condition for this function. For more information, see DV_TPT, on page 279.</td>
</tr>
</tbody>
</table>
| 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) |

**Asynchronous Operation**

To run this function asynchronously, set the `mode` parameter to EV_ASYNC. This function returns 0 to indicate it has initiated successfully, and generates a TDX_PLAYTONE termination event to indicate completion. Use the Standard Runtime Library (SRL) Event Management functions to handle the termination event; see the Standard Runtime Library API Library Reference for more information.
play tone defined by TN_GEN structure — dx_playtone()

Set termination conditions using a DV_TPT structure, which is pointed to by the tptp parameter. After dx_playtone() terminates, use the ATDX_TERMMSK() function to determine the reason for termination.

- **Synchronous Operation**

  By default, this function runs synchronously, and returns a 0 to indicate that it has completed successfully.

  Set termination conditions using a DV_TPT structure, which is pointed to by the tptp parameter. After dx_playtone() terminates, use the ATDX_TERMMSK() function to determine the reason for termination.

- **Cautions**

  - The channel must be idle when calling this function.
  - 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.
  - The DX_MAXTIME termination condition is not supported by tone generation functions, which include dx_playtone().

- **Errors**

  If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function ATDV_LASTERR() to obtain the error code or use ATDV_ERRMSGP() to obtain a descriptive error message. One of the following error codes may be returned:

  EDX_AMPLGEN
  Invalid amplitude value in TN_GEN structure

  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_FLAGGEN
  Invalid tn_dflag field in TN_GEN structure

  EDX_FREQGEN
  Invalid frequency component in TN_GEN structure

  EDX_SYSTEM
  Error from operating system
dx_playtone( ) — play tone defined by TN_GEN structure

Example

```c
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.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", 0 ) ) == -1 ) {
     * printf( "Unable to open the Voice Channel Device
     *
     * 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, 0, 25, TN_LEADING ) == -1 ) { 
     * printf( "Unable to build a Dual Tone Template
     * 
     * 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, amplitude 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 = TP_TONE;
```
play tone defined by TN_GEN structure — dx_playtone()

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_RDT;
tpt[2].tp_termno = DX_MAXTIME; /* On HMP, DX_MAXTIME not supported */
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 );

■ See Also

- dx_bidtngen()
- TN_GEN data structure
- global tone generation topic in Voice API Programming Guide
- event management functions in Standard Runtime Library API Library Reference
- DV_TPT data structure (to specify a termination condition)
- ATDX_TERMMSK()
dx_playtoneEx() — play the cadenced tone defined by TN_GENCAD

**dx_playtoneEx()**

**Name:** int dx_playtoneEx(chdev, tngencadp, tptp, mode)

**Inputs:**
- int chdev  
  • valid channel device handle
- TN_GENCAD *tngencadp  
  • pointer to the Cadenced Tone Generation template structure
- DV_TPT *tptp  
  • pointer to a Termination Parameter Table structure
- int mode  
  • asynchronous/synchronous

**Returns:**
- 0 if success
- -1 if failure

**Includes:**
- srllib.h
- dxxxlib.h

**Category:** Global Tone Generation

**Mode:** asynchronous or synchronous

---

**Description**

The **dx_playtoneEx()** function plays the cadenced tone defined by TN_GENCAD, which describes a signal by specifying the repeating elements of the signal (the cycle) and the number of desired repetitions. The cycle can contain up to four segments, each with its own tone definition and on/off duration, which creates the signal pattern or cadence. Each segment consists of a TN_GEN single- or dual-tone definition (frequency, amplitude and duration) followed by a corresponding off-time (silence duration) that is optional. The **dx_bldtngen()** function can be used to set up the TN_GEN components of the TN_GENCAD structure. The segments are seamlessly concatenated in ascending order to generate the signal cycle.

This function returns the same errors, return codes, and termination events as the **dx_playtone()** function. Also, the TN_GEN array in the TN_GENCAD data structure has the same requirements as the TN_GEN used by the **dx_playtone()** function.

Set termination conditions using the DV_TPT structure. This structure is pointed to by the tptp parameter. After **dx_playtoneEx()** terminates, use the **ATDX_TERMMSK()** function to determine the termination reason.

For signals that specify an infinite repetition of the signal cycle (cycles = 255) or an infinite duration of a tone (tg_dur = -1), you must specify the appropriate termination conditions in the DV_TPT structure used by **dx_playtoneEx()**. Valid values are for the cycles field of TN_GENCAD is 1 to 40 cycles.
play the cadenced tone defined by TN_GENCAD — dx_playtoneEx()

Parameter | Description
---|---
chdev | specifies the valid channel device handle obtained when the channel was opened using `dx_open()`
tngencadp | points to a TN_GENCAD structure (which defines a signal by specifying a cycle and its number of repetitions)
tptp | points to the DV_TPT data structure, which specifies one or more terminating conditions for this function. For more information on this structure, see `DV_TPT`, on page 279.
mode | specifies whether to run this function asynchronously or synchronously. Set to one of the following:
  - `EV_ASYNC` – run the function asynchronously
  - `EV_SYNC` – run the function synchronously (default)

To run this function asynchronously, set the `mode` parameter to `EV_ASYNC`. When running asynchronously, this function will return 0 to indicate that it has initiated successfully, and will generate a TDX_PLAYTONE termination event to indicate successful termination.

By default, this function will run synchronously, and will return a 0 to indicate successful termination of synchronous play.

### Cautions

- The channel must be idle when calling this function.
- If a `TN_GEN` tone generation template contains an invalid `tg_dflag`, or the specified amplitude or frequency is outside the valid range, `dx_playtoneEx()` will generate a TDX_ERROR event if asynchronous, or -1 if synchronous.
- The DX_MAXTIME termination condition is not supported by tone generation functions, which include `dx_playtoneEx()`.

### Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function `ATDV_LASTERR()` to obtain the error code or use `ATDV_ERRMSGP()` to obtain a descriptive error message. One of the following error codes may be returned:

- `EDX_AMPLGEN`
  - Invalid amplitude value in `TN_GEN` structure
- `EDX_BAD Parm`
  - Invalid parameter
- `EDX_BADPROD`
  - Function not supported on this board
- `EDX_BADTPT`
  - Invalid `DV_TPT` entry
dx_playtoneEx( ) — play the cadenced tone defined by TN_GENCAD

**EDX_BUSY**
Busy executing I/O function

**EDX_FLAGGEN**
Invalid tg_dflag field in TN_GEN structure

**EDX_FREQGEN**
Invalid frequency component in TN_GEN structure

**EDX_SYSTEM**
Error from operating system

---

**Example**

```c
/*$ dx_playtoneEx( ) example $*/

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

main()
{
    TN_GEN       tngen;
    TN_GENCAD    tngencad;
    DV_TPT       tpt[ 2 ];
    int          dxxxdev;
    long         term;

    /*
    * Open the Voice Channel Device and Enable a Handler
    */
    if ( ( dxxxdev = dx_open( "dxxxB1C1", 0 ) ) == -1 ) {
        perror( "dxxxB1C1" );
        exit( 1 );
    }

    /*
    * Set up the Terminating Conditions.
    * (Play until a digit is pressed or until time-out at 45 seconds.)
    */
    tpt[0].tp_type = IO_CONT;
    tpt[0].tp_termno = DX_MAXDTMF;
    tpt[0].tp_length = 1;
    tpt[0].tp_flags = TF_MAXDTMF;

    tpt[1].tp_type = IO_END;
    tpt[1].tp_termno = DX_MAXTIME; /* On HMP, DX_MAXTIME not supported */
    tpt[1].tp_length = 450;
    tpt[1].tp_flags = TF_MAXTIME;

    /*
    * Build a custom cadence dial tone to indicate that a priority message is waiting.
    * Signal cycle has 4 segments & repeats forever (cycles=255) until tpt termination:
    * Note that cycles = 255 is not supported on HMP.
    * 1) 350 + 440 Hz at -17dB ON for 125 * 10 msec and OFF for 10 *10 msec
    * 2) 350 + 440 Hz at -17dB ON for 10 *10 msec and OFF for 10 *10 msec
    * 3) 350 + 440 Hz at -17dB ON for 10 *10 msec and OFF for 10 *10 msec
    * 4) 350 + 440 Hz at -17dB ON for 10 *10 msec and OFF for 10 *10 msec
    */
```
play the cadenced tone defined by TN_GENCAD — dx_playtoneEx()

tngencad.cycles = 255;
tngencad.numsegs = 4;
tngencad.offtime[0] = 10;
tngencad.offtime[1] = 10;
tngencad.offtime[2] = 10;
tngencad.offtime[3] = 10;

dx_bldtngen( &tngencad.tone[0], 350, 440, -17, -17, 125 );
dx_bldtngen( &tngencad.tone[1], 350, 440, -17, -17, 10 );
dx_bldtngen( &tngencad.tone[2], 350, 440, -17, -17, 10 );
dx_bldtngen( &tngencad.tone[3], 350, 440, -17, -17, 10 );

/*
 * Play the custom dial tone.
 */
if ( dx_playtoneEx( dxxxdev, &tngencad, tpt, EV_SYNC ) == -1 ) {
    printf( "Unable to Play the Cadenced Tone\n" );
    printf( "Lasterror = %d  Err Msg = %s\n",
            ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
    dx_close( dxxxdev );
    exit( 1 );
}

/*
/* Examine termination reason in bitmap.
/* If time-out caused termination, play reorder tone.
 */
if((term = ATDX_TERMMSK(dxxxdev)) == AT_FAILURE) {
    /* Process error */
}

if(term & TM_MAXTIME) {
    /*
    * Play the standard Reorder Tone (fast busy) using the predefined tone
    * from the set of standard call progress signals.
    */
    if ( dx_playtoneEx( dxxxdev, CP_REORDER, tpt, EV_SYNC ) == -1 ) {
        printf( "Unable to Play the Cadenced Tone\n" );
        printf( "Lasterror = %d  Err Msg = %s\n",
                ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
        dx_close( dxxxdev );
        exit( 1 );
    }
}

/* Terminate the Program */
dx_close( dxxxdev );
exi( 0 );

See Also

- dx_playtone()
- dx_bldtngen()
- TN_GEN data structure
- TN_GENCAD data structure
dx_playvox( ) — play voice data stored in a single VOX file

**dx_playvox( )**

**Name:** int dx_playvox(chdev, filenamep, tptp, xpbp, mode)

**Inputs:**
- int chdev • valid channel device handle
- char *filenamep • pointer to name of file to play
- DV_TPT *tptp • pointer to Termination Parameter Table structure
- DX_XPB *xpbp • pointer to I/O Transfer parameter block structure
- unsigned short mode • play mode

**Returns:**
- 0 if successful
- -1 if failure

**Includes:** srlib.h
dxxxlib.h

**Category:** I/O Convenience

**Mode:** synchronous

---

**Description**

The **dx_playvox()** convenience function plays voice data stored in a single VOX file. This function calls **dx_playiottdata()**.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <strong>dx_open()</strong>.</td>
</tr>
<tr>
<td>filenamep</td>
<td>points to name of VOX file to play</td>
</tr>
<tr>
<td>tptp</td>
<td>points to the Termination Parameter Table structure, DV_TPT, which specifies termination conditions for this function. For more information on termination conditions, see <strong>DV_TPT</strong>, on page 279.</td>
</tr>
<tr>
<td>xpbp</td>
<td>points to the I/O Transfer Parameter Block structure, which specifies the file format, data format, sampling rate, and resolution of the voice data. For more information, see <strong>DX_XPB</strong>, on page 301. If xpbp is set to NULL, this function interprets the data as 6 kHz linear ADPCM.</td>
</tr>
<tr>
<td>mode</td>
<td>specifies the play mode. The following two values must be ORed together: • <strong>PM_TONE</strong> – play 200 msec audible tone • <strong>EV_SYNC</strong> – synchronous operation (must be specified)</td>
</tr>
</tbody>
</table>

**Cautions**

When playing or recording VOX files, the data format is specified in **DX_XPB** rather than through the mode parameter of **dx_playvox()**.
Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function \texttt{ATDV\_LASTERR( )} to obtain the error code or use \texttt{ATDV\_ERRMSGP( )} to obtain a descriptive error message. One of the following error codes may be returned:

- **EDX\_BADIOTT**
  - Invalid DX\_IOTT setting

- **EDX\_BADWAVEFILE**
  - Invalid WAVE file

- **EDX\_BUSY**
  - Channel is busy

- **EDX\_SH\_BADCMD**
  - Unsupported command or WAVE file format

- **EDX\_SYSTEM**
  - Error from operating system

- **EDX\_XPBPARM**
  - Invalid DX\_XPB setting

Example

```c
#include "srllib.h"
#include "dxxxlib.h"

main()
{
    int chdev;        /* channel descriptor */
    DV\_TPT tpt;        /* termination parameter table */.
    .
    .
    /* Open channel */
    if ((chdev = dx\_open("dxxxB1C1",0)) == -1) {
        printf("Cannot open channel\n");
        /* Perform system error processing */
        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, "HELLO\_VOX", \&tpt, NULL, EV\_SYNC) = = -1) {
        printf("Error playing file - %s\n", ATDV\_ERRMSGP(chdev));
        exit(4);
    }
}
```
dx_playvox( ) — play voice data stored in a single VOX file

See Also

- dx_play( )
- dx_playf( )
- dx_playiottdata( )
- dx_playwav( )
dx_playwav( )

**Name:** int dx_playwav(chdev, filenamep, tptp, mode)

**Inputs:**
- int chdev • valid channel device handle
- char *filenamep • pointer to name of file to play
- DV_TPT *tptp • pointer to Termination Parameter Table structure
- unsigned short mode • play mode

**Returns:**
- 0 if successful
- -1 if failure

**Includes:** srllib.h
dxxxlib.h

**Category:** I/O Convenience

**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 structure because the WAVE file contains the necessary format information.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <code>dx_open()</code></td>
</tr>
<tr>
<td>tptp</td>
<td>points to the Termination Parameter Table structure, DV_TPT, which specifies termination conditions for playing. For more information on this function, see <code>DV_TPT</code>, on page 279.</td>
</tr>
<tr>
<td>filenamep</td>
<td>points to the name of the file to play</td>
</tr>
</tbody>
</table>
| mode      | specifies the play mode. The following two symbolic values can be used individually or ORed together:  
  - PM_TONE – play 200 msec audible tone  
  - EV_SYNC – synchronous operation (must be specified) |

**Cautions**

This function fails when an unsupported WAVE file format is attempted to be played. For information on supported data formats, see the description for DX_XPB, on page 301 and the Voice API Programming Guide.
dx_playwav( ) — play voice data stored in a single WAVE file

Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. One of the following error codes may be returned:

EDX_BADIOTT
  Invalid DX_IOTT setting

EDX_BADWAVFILE
  Invalid WAVE file

EDX_BUSY
  Channel is busy

EDX_SH_BADCMD
  Unsupported command or WAVE file format

EDX_SYSTEM
  Error from operating system

EDX_XPBPARM
  Invalid DX_XPB setting

Example

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

main()
{
  int chdev;        /* channel descriptor */
  DV_TPT tpt;        /* termination parameter table */
  .
  .
  /* Open channel */
  if ((chdev = dx_open("dxxxB1C1",0)) == -1) {
    printf("Cannot open channel\n");
    /* Perform system error processing */
    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,"HELLO.WAV",&tpt,EV_SYNC) == -1) {
    printf("Error playing file - %s\n", ATDV_ERRMSGP(chdev));
    exit(4);
  }
}
play voice data stored in a single WAVE file — dx_playwav()
dx_PutStreamData( ) — place data into a circular stream buffer

**dx_PutStreamData( )**

**Name:** int dx_PutStreamData(hBuffer, pNewData, BuffSize, flag)

**Inputs:**
- int hBuffer
- char* pNewData
- int BuffSize
- int flag

**Returns:**
- 0 if successful
- -1 if failure

**Includes:** srllib.h
dxxxlib.h

**Category:** streaming to board

**Mode:** synchronous

---

**Description**

The **dx_PutStreamData( )** function puts data into the specified circular stream buffer. If there is not enough room in the buffer (an overrun condition), an error of -1 is returned and none of the data will be placed in the stream buffer. Writing 0 bytes of data to the buffer is not considered an error. The flag field is used to indicate that this is the last block of data. Set this flag to STREAM_CONT (0) for all buffers except the last one, which should be set to STREAM_EOD (1). This function can be called at any time between the opening and closing of the stream buffer.

**Parameter** | **Description**
--- | ---
**hBuffer** | specifies the circular stream buffer handle obtained from **dx_OpenStreamBuffer( )**
**pNewData** | a pointer to the user buffer containing data to be placed in the circular stream buffer
**BuffSize** | specifies the number of bytes in the user buffer
**flag** | a flag indicating whether this is the last block of data in the user buffer. Valid values are:
- STREAM_CONT – for all buffers except the last one
- STREAM_EOD – for the last buffer

**Cautions**

None.

**Errors**

If there is not enough room in the buffer (an overrun condition), this function returns an error of -1.
Unlike other voice API library functions, the streaming to board functions do not use SRL device handles. Therefore, ATDV_LASTERR() and ATDV_ERRMSGP() cannot be used to retrieve error codes and error descriptions.

### Example

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

main()
{
    int nBuffSize = 32768, vDev = 0;
    int hBuffer = -1;
    char pData[1024];
    DX_IOCTL iott;
    DV_TPT tpt;

    if ((hBuffer = dx_OpenStreamBuffer(nBuffSize)) < 0)
    {
        printf("Error opening stream buffer \n");
        exit(1);
    }
    if ((vDev = dx_open("dxxxB1C1", 0)) < 0)
    {
        printf("Error opening voice device\n");
        exit(2);
    }

    iott.io_type = IO_STREAM|IO_EOT;
    iott.io_bufp = 0;
    iott.io_offset = 0;
    iott.io_length = -1; /* play until STREAM_EOD */
    iott.io_fhandle = hBuffer;

    if (dx_paint(vDev, &iott, &tpt) < 0)
    {
        printf("Error in dx_paint() \n", ATDV_LASTERR(vDev));
    }
    /* Repeat the following until all data is streamed */
    if (dx_PutStreamData(hBuffer, pData, 1024, STREAM_CONT) < 0)
    {
        printf("Error in dx_PutStreamData \n");
        exit(3);
    }
    /* Wait for TDX_PLAY event and other events as appropriate */

    if (dx_CloseStreamBuffer(hBuffer) < 0)
    {
        printf("Error closing stream buffer \n");
    }
}
```

### See Also

- `dx_OpenStreamBuffer()`
dx_querytone( ) — get tone information for a specific call progress tone

The `dx_querytone( )` function returns tone information for a call progress tone currently available on the board device. On successful completion of the function, the TONE_DATA structure contains the relevant tone information.

Prior to creating a new tone definition with `dx_createtone( )`, use `dx_querytone( )` to get tone information for that tone, then use `dx_deletetone( )` to delete that tone.
get tone information for a specific call progress tone — dx_querytone()

When running asynchronously, the function returns 0 to indicate that it initiated successfully and generates the TDX_QUERYTONE event to indicate completion or TDX_QUERYTONE_FAIL to indicate failure. The TONE_DATA structure should remain in scope until the application receives these events.

By default, this function runs synchronously and returns 0 to indicate completion.

**Cautions**

- Only the default call progress tones as listed in the **toneid** parameter description are supported for this function.
- To modify a default tone definition, use the three functions **dx_querytone()**, **dx_deletetone()**, and **dx_createtone()** in this order, for one tone at a time.
- When **dx_querytone()** is issued on a board device in asynchronous mode, and the function is immediately followed by another similar call prior to completion of the previous call on the same device, the subsequent call will fail with device busy.

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>brdhdl</strong></td>
<td>specifies a valid board device handle (not a virtual board device) of the format <strong>brdBn</strong> obtained by a call to <strong>dx_open()</strong></td>
</tr>
<tr>
<td><strong>toneid</strong></td>
<td>specifies the tone ID of the call progress tone. Valid values are:</td>
</tr>
<tr>
<td></td>
<td>• TID_DIAL_LCL</td>
</tr>
<tr>
<td></td>
<td>• TID_DIAL_INTL</td>
</tr>
<tr>
<td></td>
<td>• TID_BUSY1</td>
</tr>
<tr>
<td></td>
<td>• TID_RNGBK1</td>
</tr>
<tr>
<td></td>
<td>• TID_BUSY2</td>
</tr>
<tr>
<td></td>
<td>• TID_RNGBK2</td>
</tr>
<tr>
<td></td>
<td>• TID_DISCONNECT</td>
</tr>
<tr>
<td></td>
<td>• TID_FAX1</td>
</tr>
<tr>
<td></td>
<td>• TID_FAX2</td>
</tr>
<tr>
<td></td>
<td>• TID_SIT_NC (no circuit found)</td>
</tr>
<tr>
<td></td>
<td>• TID_SIT_IC (operator intercept)</td>
</tr>
<tr>
<td></td>
<td>• TID_SIT_VC (vacant circuit)</td>
</tr>
<tr>
<td></td>
<td>• TID_SIT_RO (reorder)</td>
</tr>
<tr>
<td><strong>tonedata</strong></td>
<td>specifies a pointer to the TONE_DATA data structure that contains the tone information for the call progress tone identified by <strong>toneid</strong></td>
</tr>
<tr>
<td><strong>mode</strong></td>
<td>specifies how the function should be executed, either EV_ASYNC (asynchronous) or EV_SYNC (synchronous)</td>
</tr>
</tbody>
</table>
dx_querytone( ) — get tone information for a specific call progress tone

### Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. One of the following error codes may be returned:

- **EDX_BADPARM**
  - invalid parameter
- **EDX_SYSTEM**
  - error from operating system
- **EDX_TONEID**
  - bad tone template ID

### Example

```c
#include "srllib.h"
#include "dxxxlib.h"

main()
{
    int brdhdl; /* board handle */
    .
    .
    /* Open board */
    if ((brdhdl = dx_open("brdB1",0)) == -1)
    {
        printf("Cannot open board\n");
        /* Perform system error processing */
        exit(1);
    }

    /* Get the tone information for the TID_BUSY1 Tone*/
    int result;
    TONE_DATA tonedata;
    if ((result = dx_querytone(brdhdl, TID_BUSY1, &tonedata, EV_SYNC)) == -1)
    {
        printf("Cannot obtain tone information for TID_BUSY1 \n");
        /* Perform system error processing */
        exit(1);
    }
}
```

### See Also

- **dx_deletetone( )**
- **dx_createtone( )**
dx_rec( )

**Name:** int dx_rec(chdev, iottp, tptp, mode)

**Inputs:**
- int chdev • valid 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 setting and recording mode bit mask

**Returns:**
- 0 if successful
- -1 if failure

**Includes:**
- srllib.h
- dxxxlib.h

**Category:** I/O

**Mode:** asynchronous or synchronous

---

**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.

After `dx_rec( )` is called, recording continues until `dx_stopch( )` is called, until 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. Use the ATDX_TERMMSK( ) function to determine the reason for termination.

**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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <code>dx_open( )</code></td>
</tr>
<tr>
<td>iottp</td>
<td>points to the I/O Transfer Table Structure, DX_IOTT, which specifies the order of recording and the location of voice data. This structure must remain in scope for the duration of the function if using asynchronously. See DX_IOTT, on page 290, for more information on this data structure.</td>
</tr>
</tbody>
</table>
dx_rec() — record voice data from a single channel

Table 4 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).

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tptp</td>
<td>points to the Termination Parameter Table Structure, DV_TPT, which specifies termination conditions for recording. For more information on this structure, see DV_TPT, on page 279.</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>mode</th>
<th>defines the recording mode. One or more of the values listed below may be selected in the bit mask using bitwise OR (see Table 4 for record mode combinations).</th>
</tr>
</thead>
<tbody>
<tr>
<td>EV_ASYNC</td>
<td>run asynchronously</td>
</tr>
<tr>
<td>EV_SYNC</td>
<td>run synchronously (default)</td>
</tr>
</tbody>
</table>

#### Choose one only:

- MD_ADPCM – record using Adaptive Differential Pulse Code Modulation encoding algorithm (4 bits per sample). Recording with ADPCM is the default setting.
- MD_GAIN – record with Automatic Gain Control (AGC). Recording with AGC is the default setting.
- MD_NOGAIN – record without AGC
- MD_PCM – record using Pulse Code Modulation encoding algorithm (8 bits per sample)
- 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 6 kHz sampling rate (6000 samples per second). This is the default setting.
- RM_SR8 – record using 8 kHz sampling rate (8000 samples per second)

**Notes:**

1. 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.

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 6 kHz.

3. 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().

4. When using the RM_TONE bit for tone-initiated record, each time slot must be “listening” to the transmit time slot of the recording channel because the alert tone can only be transmitted on the recording channel transmit time slot.
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 braces, `{ }`, are default settings and do not have to be specified.

### Table 4. Record Mode Selections

<table>
<thead>
<tr>
<th>Feature</th>
<th>ADPCM (24 kbps)</th>
<th>ADPCM (32 kbps)</th>
<th>PCM (48 kbps)</th>
<th>PCM (64 kbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AGC</strong></td>
<td>RM_SR6</td>
<td>RM_SR8</td>
<td>RM_SR6</td>
<td>RM_SR8</td>
</tr>
<tr>
<td><strong>No Tone</strong></td>
<td>(MD_ADPCM)</td>
<td>(MD_ADPCM)</td>
<td>(MD_PCM)</td>
<td>(MD_PCM)</td>
</tr>
<tr>
<td></td>
<td>(MD_GAIN)</td>
<td>(MD_GAIN)</td>
<td>(MD_GAIN)</td>
<td>(MD_GAIN)</td>
</tr>
<tr>
<td><strong>No AGC</strong></td>
<td>MD_NOGAIN</td>
<td>MD_NOGAIN</td>
<td>MD_NOGAIN</td>
<td>MD_NOGAIN</td>
</tr>
<tr>
<td><strong>No Tone</strong></td>
<td>RM_SR6</td>
<td>RM_SR8</td>
<td>RM_SR6</td>
<td>RM_SR8</td>
</tr>
<tr>
<td></td>
<td>(MD_ADPCM)</td>
<td>(MD_ADPCM)</td>
<td>(MD_PCM)</td>
<td>(MD_PCM)</td>
</tr>
<tr>
<td><strong>AGC</strong></td>
<td>RM_TONE</td>
<td>RM_TONE</td>
<td>RM_TONE</td>
<td>RM_TONE</td>
</tr>
<tr>
<td><strong>Tone</strong></td>
<td>RM_SR6</td>
<td>RM_SR8</td>
<td>RM_SR6</td>
<td>RM_SR8</td>
</tr>
<tr>
<td></td>
<td>(MD_ADPCM)</td>
<td>(MD_ADPCM)</td>
<td>(MD_PCM)</td>
<td>(MD_PCM)</td>
</tr>
<tr>
<td><strong>No AGC</strong></td>
<td>MD_NOGAIN</td>
<td>MD_NOGAIN</td>
<td>MD_NOGAIN</td>
<td>MD_NOGAIN</td>
</tr>
<tr>
<td><strong>Tone</strong></td>
<td>RM_TONE</td>
<td>RM_TONE</td>
<td>RM_TONE</td>
<td>RM_TONE</td>
</tr>
<tr>
<td></td>
<td>RM_SR6</td>
<td>RM_SR8</td>
<td>RM_SR6</td>
<td>RM_SR8</td>
</tr>
<tr>
<td></td>
<td>(MD_ADPCM)</td>
<td>(MD_ADPCM)</td>
<td>(MD_PCM)</td>
<td>(MD_PCM)</td>
</tr>
</tbody>
</table>

{ } = Default modes.
* = Select if A-law encoding is required

### Asynchronous Operation

To run this function asynchronously, set the `mode` parameter to EV_ASYNC. When running asynchronously, this function returns 0 to indicate it has initiated successfully, and generates a TDX_RECORD termination event to indicate completion.

Set termination conditions using the DV_TPT structure, which is pointed to by the `tptp` parameter.

Termination of asynchronous recording is indicated by a TDX_RECORD event. Use the Standard Runtime Library (SRL) event management functions to handle the termination event.

After `dx_rec()` terminates, use the `ATDX_TERMMSK()` function to determine the reason for termination.

**Note:** The DX_IOTT data area must remain in scope for the duration of the function if running asynchronously.

### Synchronous Operation

By default, this function runs synchronously, and returns a 0 to indicate that it has completed successfully.
dx_rec() — record voice data from a single channel

Set termination conditions using the DV_TPT structure, which is pointed to by the tptp parameter. After dx_rec() terminates, use the ATDX_TERMMSK( ) function to determine the reason for termination.

■ Cautions

- If A-law data encoding is selected (RM_ALAW), the A-law parameters must be passed each time the record function is called or the setting will return to mu-law (the default).
- Voice channels must be listening to a TDM bus time slot in order for any voice streaming functions, such as dx_rec(), to work. In other words, you must issue a dx_listen() function call on the device handle before calling any voice streaming function for that device handle. Furthermore, the dx_listen() function must be called within the same process as the voice streaming functions. The actual recording operation will start only after the voice channel is listening to the proper external time slot.
- The io_fhandle member of the DX_IOTT is normally set to the value of the descriptor obtained when opening the file used for recording. That file cannot be opened in append mode since multiple recordings would corrupt the file during playback because of different coders used, header and other format-related issues. Consequently, when opening a file, the O_APPEND flag is not supported and will cause TDX_ERROR to be returned if used.

■ Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. One of the following error codes may be returned:

EDX_BADDEV
Invalid Device Descriptor

EDX_BADIOTT
Invalid DX_IOTT entry

EDX_BADParm
Invalid parameter

EDX_BADTPT
Invalid DX_TPT entry

EDX_BUSY
Busy executing I/O function

EDX_SYSTEM
Error from operating system

■ Example 1

This example illustrates how to using dx_rec() in synchronous mode.

```c
#include <fcntl.h>
#include <srllib.h>
#include <dxxxlib.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 */

    /* For Windows applications */
    if ((iott[1].io_fhandle = dx_fileopen("file.vox",
        O_RDWR|O_CREAT|O_TRUNC|O_BINARY, 0666)) == -1) {
        /* process error */
    }

    /* For Linux applications */
    if ((iott[1].io_fhandle = open("file.vox", O_RDWR|O_CREAT|O_TRUNC,
        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

This example illustrates how to use \texttt{dx\_rec()} in asynchronous mode.
#include <stdio.h>
#include <fcntl.h>
#include <srllib.h>
#include <dxxxlib.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_getevtdv());
    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;
}

See Also

- dx_recf()
- dx_reciottdata()
- dx_recvox()
- dx_setparm()
- dx_getparm()
- DX_IOTT data structure (to identify source or destination of the voice data)
- event management functions in Standard Runtime Library API Library Reference
- ATDX_TERMMSK()
- DV_TPT data structure (to specify a termination condition)
- dx_setuio()
**Description**

The `dx_recf()` function is a convenience function that records voice data from a channel to a single file.

Calling `dx_recf()` is the same as calling `dx_rec()` and specifying a single file entry in the `DX_IOTT` structure. Using `dx_recf()` is more convenient for recording to one file, because you do not have to set up a `DX_IOTT` structure for one file, and the application does not need to open the file. The `dx_recf()` function opens and closes the file specified by `fnamep`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <code>dx_open()</code></td>
</tr>
<tr>
<td>fnamep</td>
<td>points to the name of the file where voice data will be recorded</td>
</tr>
<tr>
<td>tptp</td>
<td>points to the Termination Parameter Table structure, <code>DV_TPT</code>, which specifies termination conditions for recording. For more information on this structure, see <code>DV_TPT</code>, on page 279.</td>
</tr>
<tr>
<td>mode</td>
<td>defines the recording mode. One or more of the values listed in the <strong>mode</strong> description of <code>dx_rec()</code> may be selected in the bitmask using bitwise OR (see Table 4, “Record Mode Selections”, on page 203 for record mode combinations).</td>
</tr>
</tbody>
</table>

**Cautions**

None.
 Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. One of the following error codes may be returned:

EDX_BADIOIT
Invalid DX_IOTT entry

EDX_BADPARM
Invalid parameter

EDX_BADTPT
Invalid DX_TPT entry

EDX_BUSY
Busy executing I/O function

EDX_SYSTEM
Error from operating system

 Source Code

```c
#include <stdio.h>

/*
 NAME: int dx_recf(devd,filep,tptp,mode)
 DESCRIPTION: Record data to a file
 INPUTS: devd - channel descriptor
 tptp - TPT pointer
 filep - ASCII 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     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 = open(filep,(O_WRONLY|O_CREAT|O_TRUNC),0666)) == -1) {
        return -1;
    }
}```
dx_recf() — record voice data to a single file

/* Use dx_rec() to do the record */

tott.io_type = IO_EOT | IO_DEV;
tott.io_offset = (long)0;
tott.io_length = -1;

rval = dx_rec(devd,&tott,tptp,mode);

if (close(tott.io_fhandle) == -1) {
    return -1;
}

return rval;
} 

Example

#include <srllib.h>
#include <dxxxlib.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 */
    }
    . . .
}

See Also

- dx_rec()
- dx_reciofdata( )
- dx_recox( )
- dx_setparm( )
- dx_getparm( )
- ATDX_TERMMSK( )
- DV_TPT data structure (to specify a termination condition)
**dx_reciottdata( )**

**Name:** int dx_reciottdata(chdev, iottp, tptp, xpbp, mode)

**Inputs:**
- int chdev • valid channel device handle
- DX_IOTT *iottp • pointer to I/O Transfer Table structure
- DV_TPT *tptp • pointer to Termination Parameter Table structure
- 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

**Mode:** asynchronous or synchronous

---

**Description**

The `dx_reciottdata()` function records voice data to multiple destinations, a combination of data files, memory, or custom devices.

`dx_reciottdata()` is similar to `dx_rec()`, but takes an extra parameter, `xpbp`, which allows the user to specify format information about the data to be recorded. This includes file format, data encoding, sampling rate, and bits per sample.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <code>dx_open()</code></td>
</tr>
<tr>
<td>iottp</td>
<td>points to the I/O Transfer Table Structure, DX_IOTT, which specifies the order of recording and the location of voice data. This structure must remain in scope for the duration of the function if using asynchronously. See DX_IOTT, on page 290, for more information on this data structure.</td>
</tr>
<tr>
<td>tptp</td>
<td>points to the Termination Parameter Table Structure, DV_TPT, which specifies termination conditions for recording. For more information on this structure, see DV_TPT, on page 279.</td>
</tr>
</tbody>
</table>
Voice API for HMP Library Reference — September 2004

record voice data to multiple destinations — dx_reciootdata()

Parameter Description

xpbp points to the I/O Transfer Parameter Block, DX_XPB, which specifies the file format, data format, sampling rate, and resolution for I/O data transfer. For more information on this structure, see Table, “DX_XPB”, on page 301.

mode specifies the recording mode. One or more of the values listed below may be selected in the bit mask using bitwise OR.

- EV_ASYNC – asynchronous mode
- EV_SYNC – synchronous mode
- PM_TONE – play 200 msec audible tone
- RM_TONE – transmit a tone before initiating record. If this mode is not selected, no tone will be transmitted (the default setting). For a list of all valid values, see dx_rec().

Cautions

- Voice channels must be listening to a TDM bus time slot in order for any voice streaming functions, such as dx_rec(), to work. In other words, you must issue a dx_listen() function call on the device handle before calling any voice streaming function for that device handle. Furthermore, the dx_listen() function must be called within the same process as the voice streaming functions. The actual recording operation will start only after the voice channel is listening to the proper external time slot.
- All files specified in the DX_IOTT structure will be of the file format described in DX_XPB.
- All files recorded to will have the data encoding and sampling rate as described in DX_XPB.
- When playing or recording VOX files, the data format is specified in DX_XPB rather than through the dx_setparm() function.
- The DX_IOTT data area must remain in scope for the duration of the function if running asynchronously.
- The DX_XPB data area must remain in scope for the duration of the function if running asynchronously.
- The io_fhandle member of the DX_IOTT is normally set to the value of the descriptor obtained when opening the file used for recording. That file cannot be opened in append mode since multiple recordings would corrupt the file during playback because of different coders used, header and other format-related issues. Consequently, when opening a file, the O_APPEND flag is not supported and will cause TDX_ERROR to be returned if used.

Errors

In asynchronous mode, the function returns immediately and a TDX_RECORD event is queued upon completion. Check ATDX_TERMMSK() for the termination reason. If a failure occurs during recording, then a TDX_ERROR event will be queued. Use ATDV_LASTERR() to determine the reason for error. In some limited cases such as when invalid arguments are passed to the library, the function may fail before starting the record. In such cases, the function returns -1 immediately to indicate failure and no event is queued.
dx_reciottdata() — record voice data to multiple destinations

In synchronous mode, if this function returns -1 to indicate failure, use the Standard Runtime Library (SRL) Standard Attribute function ATDV_LASTERR() to obtain the error code or use ATDV_ERRMSGP() to obtain a descriptive error message. One of the following error codes may be returned:

EDX_BADIOIT
    Invalid DX_IOTT setting

EDX_BADWAVFILE
    Invalid WAVE file

EDX_BUSY
    Channel is busy

EDX_SYSTEM
    Error from operating system

EDX_XPB PARM
    Invalid DX_XPB setting

EDX_SH_BADCMD
    Unsupported command or WAVE file format

Example

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

main()
{
    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("dxxxBlCl",0)) == -1) {
        printf("Cannot open channel\n");
        /* Perform system error processing */
        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;
    
    /* For Windows applications: open file */
    if ((fd = dx_fileopen("MESSAGE.VOX",O_RDWR|O_BINARY)) == -1) {
        printf("File open error\n");
        exit(2);
    }
}
```
record voice data to multiple destinations — dx_reciottdata()

/* For Linux applications: open file */
if ((fd = open("MESSAGE.VOX", O_RDWR)) == -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_type = 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.wBitsPerSample = 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_playvox(chdev, "HELLO.VOX", &tpt, &xpb, 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);
}

See Also

- dx_rec()
- dx_recf()
- dx_recvox()
- dx_recwav()
- dx_setuio()
dx_recvox() — record voice data to a single VOX file

dx_recvox()

Name: int dx_recvox(chdev, filenamep, tptp, xpbp, mode)

Inputs:
- int chdev • valid channel device handle
- char *filenamep • pointer to name of file to record to
- DV_TPT *tptp • pointer to Termination Parameter Table structure
- DX_XPB *xpbp • pointer to I/O Transfer Parameter Block structure
- unsigned short mode • record mode

Returns:
- 0 if successful
- -1 if failure

Includes:
- srllib.h
- dxxxlib.h

Category: I/O Convenience

Mode: synchronous

Description

The dx_recvox() function records voice data from a channel to a single VOX file. This is a convenience function.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using dx_open()</td>
</tr>
<tr>
<td>filenamep</td>
<td>points to the name of the VOX file to record to</td>
</tr>
<tr>
<td>tptp</td>
<td>points to the Termination Parameter Table Structure, DV_TPT, which specifies termination conditions for recording. For more information on this structure, see DV_TPT, on page 279.</td>
</tr>
<tr>
<td>xpbp</td>
<td>points to the I/O Transfer Parameter Block structure, which specifies the file format, data format, sampling rate, and resolution of the voice data. For more information, see DX_XPB, on page 301.</td>
</tr>
</tbody>
</table>

Note: If xpbp is set to NULL, this function interprets the data as 6 kHz linear ADPCM.

<table>
<thead>
<tr>
<th>mode</th>
<th>specifies the record mode. The following two values must be ORed together:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• PM_TONE – play 200 msec audible tone</td>
</tr>
<tr>
<td></td>
<td>• EV_SYNC – synchronous operation (must be specified)</td>
</tr>
</tbody>
</table>

Cautions

- Voice channels must be listening to a TDM bus time slot in order for any voice streaming functions, such as dx_rec(), to work. In other words, you must issue a dx_listen() function call on the device handle before calling any voice streaming function for that device handle.
Furthermore, the `dx_listent( )` function must be called within the same process as the voice streaming functions. The actual recording operation will start only after the voice channel is listening to the proper external time slot.

- When playing or recording VOX files, the data format is specified in `DX_XPB` rather than through the mode parameter of `dx_recvox( )`.

### Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function `ATDV_LASTERR( )` to obtain the error code or use `ATDV_ERRMSGP( )` to obtain a descriptive error message. One of the following error codes may be returned:

- **EDX_BADIOTT**
  - Invalid `DX_IOTT` setting
- **EDX_BUSY**
  - Channel is busy
- **EDX_SH_BADCMD**
  - Unsupported command or VOX file format
- **EDX_SYSTEM**
  - Error from operating system
- **EDX_XPBPARM**
  - Invalid `DX_XPB` setting

### Example

```c
#include "srllib.h"
#include "dxxxlib.h"

main()
{
    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");
        /* Perform system error processing */
        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));
        ATDV_ERRMSGP(chdev);
        exit(3);
    }
}
```
dx_recvox( ) — record voice data to a single VOX file

/* Start prompt playback */
if (dx_playvox(chdev,"HELLO.VOX",&tpt,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,RM_TONE|EV_SYNC) == -1){
  printf("Error recording file - %s\n", ATDV_ERRMSGP(chdev));
  exit(4);
}

See Also

- dx_rec( )
- dx_recf( )
- dx_reciotdata( )
- dx_recwav( )
dx_recwav( )

Name: int dx_recwav(chdev, filenamep, tptp, xpbp, mode)

Inputs: int chdev • valid channel device handle
char *filenamep • pointer to name of file to record to
DV_TPT *tptp • pointer to Termination Parameter Table structure
DX_XPB *xpbp • pointer to I/O Transfer Parameter Block
unsigned short mode • record mode

Returns: 0 if successful
-1 if failure

Includes: srllib.h
dxxxlib.h

Category: I/O Convenience
Mode: synchronous

Description

The dx_recwav( ) convenience function records voice data to a single WAVE file. This function in turn calls dx_reciothdata( ).

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using dx_open( )</td>
</tr>
<tr>
<td>tptp</td>
<td>points to the Termination Parameter Table structure, DV_TPT, which specifies termination conditions for playing. For more information on this function, see DV_TPT, on page 279.</td>
</tr>
<tr>
<td>filenamep</td>
<td>points to the name of the file to record to</td>
</tr>
<tr>
<td>xpbp</td>
<td>points to the I/O Transfer Parameter Block, DX_XPB, which specifies the file format, data format, sampling rate, and resolution. For more information on this structure, see DX_XPB, on page 301.</td>
</tr>
<tr>
<td>mode</td>
<td>specifies the play mode. The following two symbolic values may be used individually or ORed together:</td>
</tr>
<tr>
<td></td>
<td>• EV_SYNC – synchronous operation (must be specified)</td>
</tr>
<tr>
<td></td>
<td>• PM_TONE – play 200 msec audible tone</td>
</tr>
</tbody>
</table>

Cautions

Voice channels must be listening to a TDM bus time slot in order for any voice streaming functions, such as dx_rec( ), to work. In other words, you must issue a dx_listen( ) function call on the device.
dx_recwav() — record voice data to a single WAVE file

handle before calling any voice streaming function for that device handle. Furthermore, the dx_listen() function must be called within the same process as the voice streaming functions. The actual recording operation will start only after the voice channel is listening to the proper external time slot.

## Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function ATDV_LASTERR() to obtain the error code or use ATDV_ERRMSGP() to obtain a descriptive error message. One of the following error codes may be returned:

- EDX_BADIOTT
  - Invalid DX_IOTT setting
- EDX_BADWAVFILE
  - Invalid WAVE file
- EDX_BUSY
  - Channel is busy
- EDX_SH_BADCMD
  - Unsupported command or WAVE file format
- EDX_SYSTEM
  - Error from operating system
- EDX_XPBPARM
  - Invalid DX_XPB setting

## Example

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

int chdev;             /* channel device handle */
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");
    /* Perform system error processing */
    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;
```

record voice data to a single WAVE file — `dx_recwav()`

```c
/* Wait forever for phone to ring and go offhook */
if (dx_wstring(chdev,1,DX_OFFHOOK,-1) == -1) {
    printf("Error waiting for ring - %s\n", ATDV_LASTERR(chdev));
    exit(3);
}

/* Start playback */
if (dx_playwav(chdev,"HELLO.WAV",&tpt,EV_SYNC) == -1) {
    printf("Error playing file - %s\n", ATDV_ERRMSGP(chdev));
    exit(4);
}

/* clear digit buffer */
dx_clrdigbuf(chdev);

/* Start 11 kHz 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);
}
```

- **See Also**
  - `dx_reciothdata()`
  - `dx_recvox()`
**dx_ResetStreamBuffer( )**

**Name:** int dx_ResetStreamBuffer(hBuffer)

**Inputs:** int hBuffer  
- stream buffer handle

**Returns:**  
0 if successful  
-1 if failure

**Includes:** srllib.h  
dxxxlib.h

**Category:** streaming to board

**Mode:** synchronous

---

### Description

The **dx_ResetStreamBuffer( )** function resets the internal data for a circular stream buffer, including zeroing out internal counters as well as the head and tail pointers. This allows a stream buffer to be reused without having to close and open the stream buffer. This function will report an error if the stream buffer is currently in use (playing).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hBuffer</td>
<td>specifies the circular stream buffer handle</td>
</tr>
</tbody>
</table>

### Cautions

You cannot reset or delete the buffer while it is in use by a play operation.

### Errors

This function returns -1 when the buffer is in use by a play operation.

Unlike other voice API library functions, the streaming to board functions do not use SRL device handles. Therefore, **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** cannot be used to retrieve error codes and error descriptions.

### Example

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

main()
{
    int nBuffSize = 32768;
    int hBuffer = -1;

    if ((hBuffer = dx_OpenStreamBuffer(nBuffSize)) < 0)
    {
        printf("Error opening stream buffer \n");
        exit(1);
    }
```
if (dx_ResetStreamBuffer(hBuffer) < 0)  
  {printf("Error resetting stream buffer \n");  
    exit (2);  
  }
if (dx_CloseStreamBuffer(hBuffer) < 0)  
  {printf("Error closing stream buffer \n");  
  }

■ See Also

• dx_OpenStreamBuffer( )
• dx_CloseStreamBuffer( )
dx_setdevuio( ) — install and retrieve user-defined I/O functions

**Name:** int dx_setdevuio(chdev, devuiop, retuiop)

**Inputs:**
- int chdev • valid channel device handle
- DX_UIO *devuiop • pointer to user I/O routines structure
- DX_UIO **retuiop • pointer to return pointer for user I/O routines structure

**Returns:**
- 0 if successful
- -1 error return code

**Includes:**
- srllib.h
- dxxxlib.h

**Category:** I/O

**Mode:** synchronous

---

### Description

Supported on Windows only. The **dx_setdevuio( )** function installs and retrieves user-defined I/O functions on a per channel device basis. These user I/O functions are used on all subsequent I/O operations performed on the channel even if the application installs global user I/O functions for all devices using the **dx_setuio( )** function. The user I/O functions are installed by installing a pointer to a DX_UIO structure which contains addresses of the user-defined I/O functions.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>the channel for which the user-defined I/O functions will be installed</td>
</tr>
<tr>
<td>devuiop</td>
<td>a pointer to an application-defined global DX_UIO structure which contains the addresses of the user-defined I/O functions. This pointer to the DX_UIO structure will be stored in the voice DLL for the specified chdev channel device. The application must not overwrite the DX_UIO structure until <strong>dx_setdevuio( )</strong> has been called again for this device with the pointer to another DX_UIO structure.</td>
</tr>
<tr>
<td>retuiop</td>
<td>the address of a pointer to a DX_UIO structure. Any previously installed I/O functions for the chdev device are returned to the application as a pointer to DX_UIO structure in <strong>retuiop</strong>. If this is the first time <strong>dx_setdevuio( )</strong> is called for a device, then <strong>retuiop</strong> will be filled with the pointer to the global DX_UIO structure which may contain addresses of the user-defined I/O function that apply to all devices. Either of <strong>devuiop</strong> or <strong>retuiop</strong> may be NULL, but not both at the same time. If <strong>retuiop</strong> is NULL, the <strong>dx_setdevuio( )</strong> function will only install the user I/O functions specified via the DX_UIO pointer in <strong>devuiop</strong> but will not return the address of the previously installed DX_UIO structure. If <strong>devuiop</strong> is NULL, then the previously installed DX_UIO structure pointer will be returned in <strong>retuiop</strong> but no new functions will be installed.</td>
</tr>
</tbody>
</table>

Either of **devuiop** or **retuiop** may be NULL, but not both at the same time. If **retuiop** is NULL, the **dx_setdevuio( )** function will only install the user I/O functions specified via the DX_UIO pointer in **devuiop** but will not return the address of the previously installed DX_UIO structure. If **devuiop** is NULL, then the previously installed DX_UIO structure pointer will be returned in **retuiop** but no new functions will be installed.
install and retrieve user-defined I/O functions — dx_setdevuio()

- **Cautions**
  - The DX_UIO structure pointed to by `devuiop` must not be altered until the next call to `dx_setdevuio()` with new values for user-defined I/O functions.
  - For proper operation, it is the application’s responsibility to properly define the three DX_UIO user routines: `u_read`, `u_write` and `u_seek`. NULL is not permitted for any function. Refer to `DX_UIO`, on page 300 for more information.
  - User-defined I/O functions installed by `dx_setdevuio()` are called in a different thread than the main application thread. If data is being shared among these threads, the application must carefully protect access to this data using appropriate synchronization mechanisms (such as mutex) to ensure data integrity.

- **Errors**

  If the function returns -1 to indicate an error, use the SRL Standard Attribute function `ATDV_LASTERR()` to obtain the error code or you can use `ATDV_ERRMSGP()` to obtain a descriptive error message. The error codes returned by `ATDV_LASTERR()` are:

  - **EDX_BADDEV**
    - Invalid device descriptor
  - **EDX_BADPARAM**
    - Invalid parameter

- **Example**

```c
#include "windows.h"
#include "srllib.h"
#include "dxxxlib.h"

int chdev;                       /* channel descriptor */
DX_UIO devio;                    /* User defined I/O functions */
DX_UIO *getiop;                  /* Retrieve I/O functions */

int appread(fd, ptr, cnt)
int            fd;
char          *ptr;
unsigned       cnt;
{
    printf("appread: Read request
");
    return(read(fd, ptr, cnt));
}

int appwrite(fd, ptr, cnt)
int               fd;
char             *ptr;
unsigned          cnt;
{
    printf("appwrite: Write request
");
    return(write(fd, ptr, cnt));
}

int appseek(fd, offset, whence)
int               fd;
long              offset;
int               whence;
{
    printf("appseek: Seek request
");
    return(lseek(fd, offset, whence));
}
```
main(argc, argv)
    int    argc;
    char   *argv[];
{
    /* Open channel */
    if ((chdev = dx_open("dxxxB1C1",0)) == -1) {
        printf("Cannot open channel\n");
        /* Perform system error processing */
        exit(1);
    }
    /* Other initialization */
    /* Initialize the device specific UIO structure */
    devio.u_read  = appread;
    devio.u_write = appwrite;
    devio.u_seek  = appseek;
    /* Install the applications I/O routines */
    if (dx_setdevuio(chdev, &devio, &getiop) == -1) {
        printf("error registering the UIO routines = %d\n", ATPV_LASTERR(chdev));
    }
}

■ See Also

- dx_setuio()
dx_setdigtyp( )

**Description**

The `dx_setdigtyp( )` function controls the types of digits the voice channel detects.

**Notes:**
1. This function only applies to the standard voice board digits; that is, DTMF, MF. To set user-defined digits, use the `dx_addtone( )` function.
2. `dx_setdigtyp( )` does not clear the previously detected digits in the digit buffer.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <code>dx_open( )</code></td>
</tr>
<tr>
<td>dmask</td>
<td>sets the type of digits the channel will detect. More than one type of digit detection can be enabled in a single function call, as shown in the function example. The following are valid values:</td>
</tr>
<tr>
<td></td>
<td>• DM_DTMF – enable DTMF digit detection</td>
</tr>
<tr>
<td></td>
<td>• DM_MF – enable MF digit detection</td>
</tr>
<tr>
<td></td>
<td>• NULL – disable digit detection</td>
</tr>
</tbody>
</table>

**Notes:**
1. MF detection can only be enabled on systems with MF capability.
2. The digit detection type specified in `dmask` will remain valid after the channel has been closed and reopened.
3. `dx_setdigtyp( )` overrides digit detection enabled in any previous use of `dx_setdigtyp( )`.

For any digit detected, you can determine the digit type by using the `DV_DIGIT` data structure in the application. When a `dx_getdig( )` call is performed, the digits are collected and transferred to the user’s digit buffer. The digits are stored as an array inside the `DV_DIGIT` structure. For more information on this structure, see `DV_DIGIT`, on page 278.
dx_setdigtyp( ) — control the types of digits detected by the voice channel

■ Cautions

Some MF digits use approximately the same frequencies as DTMF digits (see Chapter 6, “Supplementary Reference Information”). 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, do NOT enable DTMF and MF detection at the same time.

■ Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. One of the following error codes may be returned:

EDX_BADPARM
Invalid parameter

EDX_SYSTEM
Error from operating system

■ Example

On HMP, dial pulse detection (DPD) is not supported.

/*$ dx_setdigtyp( ) and dx_getdig( ) example for Global Dial Pulse Detection $*/

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

void main(int argc, char **argv)
{
  int       dev; /* device handle */
  DV_DIGIT  dig;
  DV_TPT    tpt;

  /* Open device, make or accept call */

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

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

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

  /* collect 3 digits from the user */
  if (dx_getdig(dev, &tpt, &dig, EV_SYNC) == -1) {
    /* error, display error message */
    printf("dx_getdig error %d, %s\n", ATDV_LASTERR(dev), ATDV_ERRMSGP(dev));
  } else {

}
control the types of digits detected by the voice channel — dx_setdigtyp()

/* display digits received and digit type */
printf("Received "s":\n", dig.dg_value);
printf("Digit type is ");

/*
 * digit types have 0x30 ORed with them strip it off
 * so that we can use the DG_xxx equates from the header files
 */
switch ((dig.dg_type[0] & 0x000f)) {
  case DG_DTMF:
    printf("DTMF\n");
    break;
  case DG_DPD:
    printf("DPD\n");
    break;
  default:
    printf("Unknown, %d", (dig.dg_type[0] & 0x000f));
}

/*
 * continue processing call
 */

■ See Also

- dx_addtone()
**dx_setevtmsk( )**

**Name:** int dx_setevtmsk(chdev, mask)

**Inputs:**
- int chdev          • valid channel device handle
- unsigned int mask  • event mask of events to enable

**Returns:**
- 0 if successful
- -1 if failure

**Includes:** srllib.h
dxxxlib.h

**Category:** Call Status Transition Event

**Mode:** synchronous

---

**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.

When you enable detection of a CST event and the event occurs, it will be placed on the event queue. You can collect the event by getting it or waiting for it with an event handling function, such as `sr_waitevt()`, `sr_waitevtEx()`, or `dx_getevt()`. For a list of call status transition events, see Section 3.4, “Call Status Transition (CST) Events”, on page 273.

**Note:** This function can enable detection for all CST events except user-defined tone detection. See `dx_addtone( )` and `dx_enbtone( )` for information.
enable detection of call status transition (CST) events — dx_setevtsmsk( )

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using dx_open( )</td>
</tr>
</tbody>
</table>
| mask      | specifies the events to enable. To poll for multiple events, perform an OR operation on the bit masks of the events you want to enable. The first enabled CST event to occur will be returned. If an event is not specified in the mask, the event will be disabled. If an event is enabled, it will remain enabled until it is disabled through another function call; exceptions are DM_DIGITS and DM_DIGOFF. One or more of the following bits can be set:  
  • DM_SILOF – wait for non-silence  
  • DM_SILON – wait for silence  
  • DM_DIGITS – enable digit reporting on the event queue (each detected digit is reported as a separate event on the event queue)  
  • DM_DIGOFF – disable digit reporting on the event queue (as enabled by DM_DIGITS). This is the only way to disable DM_DIGITS.  
  • DM_UNDERRUN – enables firmware underrun reporting (TDX_UNDERRUN event) for streaming to board feature. This mask works like a toggle key. If set once, the next call to the function will unset this mask.  
  • DM_VADEVTS – voice activity detector (VAD) event notification (used in conjunction with the continuous speech processing (CSP) API library only)  
  • DM_CONVERGED – echo cancellation convergence notification (used in conjunction with the continuous speech processing (CSP) API library only) |

If DM_DIGITS is specified, a digits flag is set that causes individual digit events to queue until this flag is turned off by DM_DIGOFF. 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 an event handling function such as sr_waitevt( ), sr_waitevtEx(), or dx_getevt(). The event queue is not affected by dx_getdig() calls.

To enable DM_DIGITS:

```c
/* Set event mask to collect digits */
if (dx_setevtsmsk(chdev, DM_DIGITS) == -1) { |
```

To disable DM_DIGITS (turn off the digits flag and stop queuing digits):

```c
dx_setevtsmsk(DM_DIGOFF);
dx_clrdigbuf(chdev); /*Clear out queue*/
```

The following outlines the synchronous or asynchronous handling of CST events:

**Synchronous Application**

Call `dx_setevtsmsk()` to enable CST events.

**Asynchronous Application**

Call `dx_setevtsmsk()` to enable CST events.
dx_setevtmsk( ) — enable detection of call status transition (CST) events

Call dx_getevt( ) to wait for CST events. Events are returned to the DX_EBLK structure.

Use Standard Runtime Library (SRL) to asynchronously wait for TDX_CST events.

Use sr_getevtdatap( ) to retrieve DX_CST structure.

■ Cautions

- If you call this function on a busy device, and specify DM_DIGITS as the mask argument, the function will fail.
- On Linux, events are preserved between dx_getevt( ) function calls. The event that was set remains the same until another call to dx_setevtmsk( ) changes it.
- On Linux, in a TDM bus configuration, when a voice resource is not listening to a network device, it may report spurious silence-off transitions and ring events if the events are enabled. To eliminate this problem:
  - Disable the ring and silence detection on unrouted/unlistened channels using the dx_setevtmsk( ) function.
  - When you need to change the resource currently connected to your network device, do a half duplex disconnect of the current resource to disconnect the transmit time slot of the current resource (since two resources cannot transmit on the same time slot, although they can both listen), and a full duplex connect on the new resource using the appropriate listen/unlisten functions or the convenience functions nr_scroute( ) and nr_scunroute( ).

■ Errors

This function will fail and return -1 if the channel device handle is invalid or if any of the masks set for that device are invalid.

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. One of the following error codes may be returned:

EDX_BADPARM
  Invalid parameter

EDX_SYSTEM
  Error from operating system

■ Example

This example illustrates how to use dx_setevtmsk( ) to handle call status transition events in an asynchronous application.

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

#define MAXCHAN 24

int cst_handler();
```

```c
Call dx_getevt( ) to wait for CST events. Events are returned to the DX_EBLK structure.

Use Standard Runtime Library (SRL) to asynchronously wait for TDX_CST events.

Use sr_getevtdatap( ) to retrieve DX_CST structure.

■ Cautions

- If you call this function on a busy device, and specify DM_DIGITS as the mask argument, the function will fail.
- On Linux, events are preserved between dx_getevt( ) function calls. The event that was set remains the same until another call to dx_setevtmsk( ) changes it.
- On Linux, in a TDM bus configuration, when a voice resource is not listening to a network device, it may report spurious silence-off transitions and ring events if the events are enabled. To eliminate this problem:
  - Disable the ring and silence detection on unrouted/unlistened channels using the dx_setevtmsk( ) function.
  - When you need to change the resource currently connected to your network device, do a half duplex disconnect of the current resource to disconnect the transmit time slot of the current resource (since two resources cannot transmit on the same time slot, although they can both listen), and a full duplex connect on the new resource using the appropriate listen/unlisten functions or the convenience functions nr_scroute( ) and nr_scunroute( ).

■ Errors

This function will fail and return -1 if the channel device handle is invalid or if any of the masks set for that device are invalid.

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. One of the following error codes may be returned:

EDX_BADPARM
  Invalid parameter

EDX_SYSTEM
  Error from operating system

■ Example

This example illustrates how to use dx_setevtmsk( ) to handle call status transition events in an asynchronous application.

```c
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.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;
        .
    }
    .
    .
}
dx_setevtmsk() — enable detection of call status transition (CST) events

break;
case DE_LCOFF: 
    . 
    . 
} /* Kick off next function in the state machine model. */ 
    . 
    return 0;
}

- See Also

- dx_getevt() (to handle call status transition events, synchronous operation)
- sr_getevtdatap() (to handle call status transition events, asynchronous operation)
- DX_CST data structure
- dx_addtone()
dx_setgtdamp()

Name: void dx_setgtdamp(gtd_minamp1, gtd_maxamp1, gtd_minamp2, gtd_maxamp2)

Inputs:
- short int gtd_minamp1: minimum amplitude of the first frequency
- short int gtd_maxamp1: maximum amplitude of the first frequency
- short int gtd_minamp2: minimum amplitude of the second frequency
- short int gtd_maxamp2: maximum amplitude of the second frequency

Returns: void

Includes: srllib.h
dxxxlib.h

Category: Global Tone Detection

Mode: synchronous

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_blddt(), dx_blddtcad(), dx_bldst(), or dx_bldstcad() followed by dx_addtone(). Once called, the values set will take effect for all dx_blddt(), dx_blddtcad(), dx_bldst(), and dx_bldstcad() function calls.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gtd_minamp1</td>
<td>specifies the minimum amplitude of tone 1, in dB</td>
</tr>
<tr>
<td>gtd_maxamp1</td>
<td>specifies the maximum amplitude of tone 1, in dB</td>
</tr>
<tr>
<td>gtd_minamp2</td>
<td>specifies the minimum amplitude of tone 2, in dB</td>
</tr>
<tr>
<td>gtd_maxamp2</td>
<td>specifies the maximum amplitude of tone 2, in dB</td>
</tr>
</tbody>
</table>

If this function is not called, then the MINERG firmware parameters that were downloaded remain at the following settings: -42 dBm for minimum amplitude and 0 dBm for maximum amplitude.

<table>
<thead>
<tr>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GT_MIN_DEF</td>
<td>Default value in dB for minimum GTD amplitude that can be entered for gtd_minamp1 parameters.</td>
</tr>
<tr>
<td>GT_MAX_DEF</td>
<td>Default value in dB for maximum GTD amplitude that can be entered for gtd_maxamp1 parameters.</td>
</tr>
</tbody>
</table>

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, call this function with the defines GT_MIN_DEF and GT_MAX_DEF for minimum and maximum defaults.
dx_setgtdamp( ) — set up the tone detection amplitudes

- When using this function in a multi-threaded application, use critical sections or a semaphore around the function call to ensure a thread-safe application. Failure to do so will result in “Bad Tone Template ID” errors.

## Errors

None.

## Example

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

#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_bldtt(TID1, 1000, 50, 500, 25, TN LEADING) ==-1) {
  /* Perform system error processing */
  exit(3);
}
.
.
.
```

## See Also

None.
**dx_setparm( )**

**Name:** int dx_setparm(dev, parm, valuep)

**Inputs:**
- int dev: valid 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

**Mode:** synchronous

---

**Description**

The `dx_setparm( )` function sets physical parameters of a channel or board device, such as off-hook delay, length of a pause, and flash character. You can set only one parameter at a time.

A different set of parameters is available for board and channel devices. Board parameters affect all channels on the board. Channel parameters affect the specified channel only.

The channel must be idle (that is, no I/O function running) when calling `dx_setparm( )`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dev</td>
<td>Specifies the valid channel or board device handle obtained when the channel or board was opened using <code>dx_open( )</code>.</td>
</tr>
<tr>
<td>parm</td>
<td>Specifies the channel or board parameter to set. The voice device parameters allow you to query and control device-level information and settings related to the voice functionality. See Table 5 for board parameter defines and Table 6 for channel parameter defines. Note: The parameters set in parm will remain valid after the device has been closed and reopened.</td>
</tr>
<tr>
<td>valuep</td>
<td>Points to the 4-byte 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 section.</td>
</tr>
</tbody>
</table>

The `dxxxlib.h` file contains defined masks for parameters that can be examined and set using `dx_getparm( )` and `dx_setparm( )`. 
dx_setparm() — set physical parameters of a channel or board device

The voice device parameters fall into two classes:

- **Board parameters**, which apply to all channels on the board; voice board parameter defines have a DXBD_ prefix.
- **Channel parameters**, which apply to individual channels on the board; voice channel parameter defines have a DXCH_ prefix.

### Board Parameter Defines

The supported board parameter defines are shown in Table 5.

**Table 5. Voice Board Parameters**

<table>
<thead>
<tr>
<th>Define</th>
<th>Bytes</th>
<th>Read/Write</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXBD_CHNUM</td>
<td>1</td>
<td>R</td>
<td>-</td>
<td>Channel Number. Number of channels on the board</td>
</tr>
<tr>
<td>DXBD_SYSCFG</td>
<td>1</td>
<td>R</td>
<td>-</td>
<td>System Configuration. On HMP, 1 is always returned.</td>
</tr>
</tbody>
</table>

### Channel Parameter Defines

The supported channel parameter defines are shown in Table 6. All time units are in multiples of 10 msec unless otherwise noted.

**Table 6. Voice Channel Parameters**

<table>
<thead>
<tr>
<th>Define</th>
<th>Bytes</th>
<th>Read/Write</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXCH_AGC_MAXGAIN</td>
<td>2</td>
<td>W</td>
<td>116</td>
<td>Automatic Gain Control. Specifies the maximum gain measured in 0.1 dB units. The default value of 116 is equivalent to 11.6 dB.</td>
</tr>
<tr>
<td>DXCH_AGC_MEMORY_MAXIMUMSIZE</td>
<td>2</td>
<td>W</td>
<td>300</td>
<td>Automatic Gain Control. Specifies the maximum size of memory measured in 1 msec units.</td>
</tr>
<tr>
<td>DXCH_AGC_MEMORY_SILENCERESSET</td>
<td>2</td>
<td>W</td>
<td>50</td>
<td>Automatic Gain Control. Specifies the size of memory after each long silence between words or sentences measured in 1 msec units.</td>
</tr>
<tr>
<td>DXCH_AGC_NOISE_THRESHOLD</td>
<td>2</td>
<td>W</td>
<td>-780</td>
<td>Automatic Gain Control. AGC noise threshold level. Specifies the lower threshold for noise level estimate: below is considered noise. Measured in 0.1 dB units. The default value of -780 is equivalent to -78 dB.</td>
</tr>
<tr>
<td>DXCH_AGC_SPEECH_THRESHOLD</td>
<td>2</td>
<td>W</td>
<td>-400</td>
<td>Automatic Gain Control. AGC speech threshold level. Specifies the upper threshold for noise level estimate: above is considered speech. Measured in 0.1 dB units. The default value of -400 is equivalent to -40 dB.</td>
</tr>
<tr>
<td>DXCH_AGC_TARGET_OUTPUTLEVEL</td>
<td>2</td>
<td>W</td>
<td>-196</td>
<td>Automatic Gain Control. Specifies the AGC target level; also known as AGC K constant. Measured in 0.1 dB units. The default value of -196 is equivalent to -19.6 dB.</td>
</tr>
</tbody>
</table>
set physical parameters of a channel or board device — dx_setparm( )

Table 6. Voice Channel Parameters (Continued)

<table>
<thead>
<tr>
<th>Define</th>
<th>Bytes</th>
<th>Read/Write</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
</table>
| DXCH_PLAYDRATE    | 2     | R/W        | 6000    | Play Digitization Rate. Sets the digitization rate of the voice data that is played on this channel. Voice data must be played at the same rate at which it was recorded. Valid values are: | 6000 – 6 kHz sampling rate
|                   |       |            |         |                                                                               | 8000 – 8 kHz sampling rate |
| DXCH_RECRDRATE    | 2     | R/W        | 6000    | Record Digitization Rate. Sets the rate at which the recorded voice data is digitized. Valid values are: | 6000 – 6 kHz sampling rate
|                   |       |            |         |                                                                               | 8000 – 8 kHz sampling rate |

- **Cautions**
  - 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.
  - When setting channel parameters, the channel must be open and in the idle state.
  - When setting board parameters, all channels on that board must be idle.

- **Errors**
  If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function `ATDV_LASTERR()` to obtain the error code or use `ATDV_ERRMSGP()` to obtain a descriptive error message. One of the following error codes may be returned:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDX_BADPARM</td>
<td>Invalid parameter</td>
</tr>
<tr>
<td>EDX_SYSTEM</td>
<td>Error from operating system</td>
</tr>
</tbody>
</table>

- **Example**

```c
#include <srllib.h>
#include <dxxxlib.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 */
  }
}
```

_Voice API for HMP Library Reference — September 2004_
dx_setparm( ) — set physical parameters of a channel or board device

/* now wait for an incoming ring */
   . . .
}

See Also

- dx_getparm( )
**dx_setsvcond( )**

**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
dxxlib.h

**Category:** Volume

**Mode:** synchronous

---

**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 volume 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 Volume Modification Table. For more information about this table, see the *Voice API Programming Guide*. On HMP, speed adjustment is not supported.

**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 and Volume Adjustment Condition Blocks structure (DX_SVCB):

- Volume Modification Table to use
- adjustment type (increase/decrease, absolute value, toggle)
- adjustment conditions (incoming digit, beginning of play)
- level/edge sensitivity for incoming digits

See **DX_SVCB** for a full description of the data structure. Up to 20 DX_SVCB blocks can be specified in the form of an array.

**Notes:**

1. For volume adjustment, 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 volume is based on the most recent modification.
dx_setsvcond( ) — set conditions that adjust volume of play

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <code>dx_open( )</code></td>
</tr>
<tr>
<td>numblk</td>
<td>specifies the number of DX_SVCB blocks in the array. Set to a value between 1 and 20.</td>
</tr>
<tr>
<td>svcbp</td>
<td>points to an array of DX_SVCB structures</td>
</tr>
</tbody>
</table>

### Cautions

- Speed control is supported only at the 8 kHz sampling rate using the PCM voice coder with A-law or mu-law coding, or the OKI ADPCM voice coder.
- Digits that are used for play adjustment may also be used as a terminating condition. If a digit is defined as both, then both actions are applied upon detection of that digit.
- When adjustment is associated with a DTMF digit, speed can be increased or decreased in increments of 1 (10%) only.
- When adjustment is associated with a DTMF digit, volume can be increased or decreased in increments of 1 (2 dB) only.
- 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. For example, if DTMF digit 1 has already been set to increase play volume by one step, a second call that attempts to redefine digit 1 to the origin will have no effect; the digit will retain its original setting.
- The digit that causes the play adjustment will not be passed to the digit buffer, so it cannot be retrieved using `dx_getdig( )`.

### Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function `ATDV_LASTERR( )` to obtain the error code or use `ATDV_ERRMSGP( )` to obtain a descriptive error message. One of the following error codes may be returned:

- **EDX_BADPARM**
  - Invalid parameter
- **EDX_BADPROD**
  - Function not supported on this board
- **EDX_SVADJBLKS**
  - Invalid number of speed/volume adjustment blocks
- **EDX_SYSTEM**
  - Error from operating system

### Example

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

---

**Note:** The content above is a natural representation of the document in question. It has been formatted to adhere to the guidelines for natural text. The original document is intended for programming and may contain specific technical details that are relevant to the software development process, particularly in the context of the Intel Voice API for HMP Library Reference. The focus is on providing clear and detailed information about the `dx_setsvcond( )` function, including its parameters, conditions, cautions, errors, and example usage.
set conditions that adjust volume of play — dx_setsvcond()

```c
/*
 * Global Variables
 */
DX_SVCB svcb[ 10 ] = {
    /* BitMask AdjustmentSize 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", 0 ) ) == -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\n");
        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 );
}

See Also

- dx_setsvcond()
- DX_SVCB structure
- dx_setsvmt()
- dx_getcursv()
```
dx_setsvcond() — set conditions that adjust volume of play

- dx_getsvmt()
- dx_adjsv()
- volume modification table in Voice API Programming Guide
**dx_setsvmt( )**

**Name:** int dx_setsvmt(chdev, tabletype, svmt, flag)

**Inputs:**
- chdev: valid channel device handle
- unsigned short tabletype: type of table to update (volume)
- DX_SVMT * svmt: pointer to volume modification table to modify
- unsigned short flag: optional modification flag

**Returns:**
- 0 if success
- -1 if failure

**Includes:**
- srllib.h
- dxxxlib.h

**Category:** Volume

**Mode:** synchronous

---

**Description**

The `dx_setsvmt( )` function updates the volume modification table for a channel using the values contained in a specified `DX_SVMT` structure. On HMP, speed adjustment is not supported.

This function can modify the volume modification table so that the following occurs:

- When volume adjustment reaches its 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 volume modification table to its default values. Defaults are listed in the *Voice API Programming Guide*.

For more information on volume modification tables, refer to `DX_SVMT` and see also the *Voice API Programming Guide*.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <code>dx_open( )</code></td>
</tr>
<tr>
<td>tabletype</td>
<td>specifies the volume modification table:</td>
</tr>
<tr>
<td></td>
<td>• SV_VOLUMETBL – update the volume modification table values</td>
</tr>
</tbody>
</table>
**dx_setsvmt( ) — change default values of the volume modification table**

**Cautions**

If you close a device via `dx_close( )` after modifying volume table values using `dx_setsvmt( )`, the `dx_getcursv( )` function may return incorrect volume settings for the device. This is because the next `dx_open( )` resets the volume table to default values. Therefore, it is recommended that you do not issue a `dx_close( )` during a call where you have modified volume table values.

**Errors**

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function `ATDV_LASTERR( )` to obtain the error code or use `ATDV_ERRMSGP( )` to obtain a descriptive error message. One of the following error codes may be returned:

- **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**
  - Neither SV_SPEEDTBL nor SV_VOLUMETBL was specified
- **EDX_SVMTRANGE**
  - An entry in DX_SVMT was out of range
- **EDX_SVMTSIZE**
  - Invalid table size specified
- **EDX_SYSTEM**
  - Error from operating system

**Example**

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>svmtp</td>
<td>points to the DX_SVMT structure whose contents are used to update either the speed or volume modification table (speed is not supported in HMP) This structure is not used when SV_SETDEFAULT has been set in the flag parameter.</td>
</tr>
</tbody>
</table>
| flag      | Specifies one of the following:  
  - SV_SETDEFAULT – reset the table to its default values. See the *Voice API Programming Guide* for a list of default values. In this case, the DX_SVMT pointed to by svmtp is ignored.  
  - SV_WRAPMOD – wrap around the volume adjustments that occur at the top or bottom of the volume modification table.  
  **Note:** Set flag to 0 if you do not want to use either SV_WRAPMOD or SV_SETDEFAULT. |

**Parameter Description**
/* 
 * Global Variables
 */
main()
{
    DX_SVMT   svmt;
    int       dxxxdev, index;

    /*
    * Open the Voice Channel Device and Enable a Handler
    */
    if ( ( dxxxdev = dx_open( "dxxxBlCl", 0 ) ) == -1 ) {
        perror( "dxxxBlCl" );
        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 ] = -128;
    svmt.decrease[ 6 ] = -128;
    svmt.decrease[ 7 ] = -128;
    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 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 );
}

See Also

- dx_adjsv()
- dx_getcursv()
- dx_getsvmt()
- volume modification table in Voice API Programming Guide
- DX_SVMT data structure
dx_setuio( )

**Name:** int dx_setuio(uioblk)

**Inputs:**
- uioblk  
  - DX_UIO data structure

**Returns:**
- 0 if success
- -1 if failure

**Includes:**
- srllib.h
- dxxxlib.h

**Category:** I/O

**Mode:** synchronous

---

### Description

The `dx_setuio()` function installs user-defined `read()`, `write()`, and `lseek()` functions in your application. These functions are then used by play and record functions, such as `dx_play()` and `dx_rec()`, to read and/or write to nonstandard storage media.

The application provides the addresses of user-defined `read()`, `write()` and `lseek()` functions by initializing the DX_UIO structure. See DX_UIO, on page 300 for more information on this structure.

You 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. You must OR the IO_UIO flag with the IO_DEV flag for this feature to function properly. See DX_IOTT, on page 290 for more information.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uioblk</td>
<td>specifies the DX_UIO structure, a user-defined I/O structure</td>
</tr>
</tbody>
</table>

### Cautions

- In order for the application to work properly, the user-provided functions must conform to standard I/O function semantics.
- A user-defined function must be provided for all three I/O functions. NULL is not permitted.
- User-defined I/O functions installed by `dx_setuio()` are called in a different thread than the main application thread. If data is being shared among these threads, the application must carefully protect access to this data using appropriate synchronization mechanisms (such as mutex) to ensure data integrity.

### Errors

None.
dx_setuio( ) — install user-defined I/O functions

Example

```c
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h> /* 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[];
{
if(argc == 1)
  printf("Usage: \n");
else
  printf("Usage: \n");
}
```

```c
/* 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);
```
install user-defined I/O functions — dx_setuio( )

/*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 */
iotti++;     
iotti->io_type = IO_DEV|IO_UIO|IO_CONT    
iotti->io_fhandle = vodat_fd;        
iotti->io_offset = 20001;           
iotti->io_length = 20000;

/*This block uses standard I/O functions */
iotti++;     
iotti->io_type = IO_DEV|IO_CONT
iotti->io_fhandle = vodat_fd;
iotti->io_offset = 20002;
iotti->io_length = 20000;

/*This block uses my I/O functions */
iotti->io_type = IO_DEV|IO_UIO|IO_EOT
iotti->io_fhandle = vodat_fd;
iotti->io_offset = 10003;
iotti->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);

See Also

- dx_cacheprompt( )
- dx_play( )
- dx_playiottdata( )
- dx_rec( )
- dx_reciottdata( )
dx_SetWaterMark( )

Name: int dx_SetWaterMark(hBuffer, parm_id, value)

Inputs: int hBuffer
        • circular stream buffer handle
        int parm_id
        • LOW_MARK or HIGH_MARK
        int value
        • value of water mark in bytes

Returns: 0 if successful
         -1 if failure

Includes: srllib.h
         dxxxlib.h

Category: streaming to board

Mode: synchronous

Description

The dx_SetWaterMark( ) function sets the low and high water marks for the specified stream buffer. If you don’t use this function, default values are in place for the low and high water marks based on the stream buffer size. See parameter description table for more information.

When setting the low and high water mark values for the stream buffer, do so in conjunction with the buffer size in dx_OpenStreamBuffer( ). For hints and tips on setting water mark values, see the streaming to board topic in the Voice API Programming Guide.

The application receives TDX_LOWATER and TDX_HIGHATER events regardless of whether or not dx_SetWaterMark( ) is used in your application. These events are generated when there is a play operation with this buffer and are reported on the device that is performing the play. If there is no active play, the application will not receive any of these events.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hBuffer</td>
<td>specifies the circular stream buffer handle</td>
</tr>
<tr>
<td>parm_id</td>
<td>specifies the type of water mark. Valid values are:</td>
</tr>
<tr>
<td></td>
<td>• LOW_MARK – low water mark, which by default is set to 10% of the stream buffer size</td>
</tr>
<tr>
<td></td>
<td>• HIGH_MARK – high water mark, which by default is set to 90% of the stream buffer size</td>
</tr>
<tr>
<td>value</td>
<td>specifies the value of the water mark in bytes</td>
</tr>
</tbody>
</table>

Cautions

None.
set water mark for the circular stream buffer — dx_SetWaterMark( )

- Errors

This function returns -1 in case of error.

Unlike other voice API library functions, the streaming to board functions do not use SRL device handles. Therefore, ATDV_LASTERR() and ATDV_ERRMSGP() cannot be used to retrieve error codes and error descriptions.

- Example

```c
#include <srllib.h>
#include <dxlib.h>

main()
{
    int nBuffSize = 32768;
    int hBuffer = -1;

    if ((hBuffer = dx_OpenStreamBuffer(nBuffSize)) < 0)
    {
        printf("Error opening stream buffer \n");
        exit(1);
    }
    if (dx_SetWaterMark(hBuffer, LOW_MARK, 1024) < 0)
    {
        printf("Error setting low water mark \n");
        exit(2);
    }
    if (dx_SetWaterMark(hBuffer, HIGH_MARK, 31744) < 0)
    {
        printf("Error getting setting high water mark \n");
        exit(3);
    }
    if (dx_CloseStreamBuffer(hBuffer) < 0)
    {
        printf("Error closing stream buffer \n");
    }
}
```

- See Also

- dx_OpenStreamBuffer( )
**dx_stopch( ) — force termination of currently active I/O functions**

**dx_stopch( )**

**Name:** int dx_stopch(chdev, mode)

**Inputs:**
- int chdev • valid channel device handle
- unsigned short mode • mode flag

**Returns:**
- 0 if success
- -1 if failure

**Includes:** srllib.h
dxxlib.h
dxxxlib.h

**Category:** I/O

**Mode:** asynchronous or 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 **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 you issue **dx_stopch( )** to terminate an I/O function, the termination reason returned by **ATDX_TERMMSK( )** is TM_USRSTOP. However, if **dx_stopch( )** terminates a **dx_dial( )** function with call progress analysis, use **ATDX_CPTERM( )** to determine the reason for call progress analysis termination, which is CR_STOPD.
force termination of currently active I/O functions — *dx_stopch*()

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <em>dx_open()</em></td>
</tr>
<tr>
<td>mode</td>
<td>a bit mask that specifies the mode:</td>
</tr>
<tr>
<td></td>
<td>• EV_SYNC – synchronous mode</td>
</tr>
<tr>
<td></td>
<td>• EV_ASYNC – asynchronous mode. The stop will be issued, but the driver does not “sleep” and wait for the channel to become idle before <em>dx_stopch()</em> returns.</td>
</tr>
<tr>
<td></td>
<td>• EV_NOSTOP – if this bit is set and the channel is idle, TDX_NOSTOP event is generated.</td>
</tr>
<tr>
<td></td>
<td>• EV_STOPGETEVT – if this bit is set and <em>dx_stopch()</em> is issued during <em>dx_getevt()</em> , TDX_CST event is generated with reason of DE_STOPGETEVT.</td>
</tr>
<tr>
<td></td>
<td>• IGNORESTATE – (Windows only) Ignores the busy/idle state of the channel. Performs a stop on the channel regardless of whether the channel is busy or idle. If this flag is used, the function will not check for a busy state on the channel and will issue a stop even if the channel is busy.</td>
</tr>
</tbody>
</table>

#### Cautions

- *dx_stopch()* has no effect on a channel that has any of the following functions issued:
  - *dx_dial()* without call progress analysis enabled
    The functions will continue to run normally, and *dx_stopch()* will return a success. For *dx_dial()* , the digits specified in the *dialstrp* parameter will still be dialed.
  - If *dx_stopch()* is called on a channel dialing with call progress analysis enabled, the call progress analysis process will stop but dialing will be completed. Any call progress analysis information collected prior to the stop will be returned by extended attribute functions.
  - 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.
  - When calling *dx_stopch()* from a signal handler, *mode* must be set to EV_ASYNC.
  - On Linux, when issued on a channel that is already idle, *dx_stopch()* will return an event, TDX_NOSTOP, to specify that no STOP was needed or issued. To use this functionality, “OR” the mode flag with the EV_NOSTOP flag. This does not affect the existing functionality of *dx_stopch()* . If a function is in progress when *dx_stopch()* is called with the EV_NOSTOP flag, that function will be stopped as usual and EV_NOSTOP will be ignored.
  - On Linux, an application can use *dx_stopch()* from within a signal handler to stop the *dx_getevt()* function. To do so, “OR” the mode flag with the EV_STOPGETEVT flag. The *dx_getevt()* function will successfully return with the event DE_STOPGETEVT.

#### Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function *ATDV_LASTERR()* to obtain the error code or use *ATDV_ERRMSGP()* to obtain a descriptive error message. One of the following error codes may be returned:

- EDX_BADPARM
  - Invalid parameter

---

*Voice API for HMP Library Reference — September 2004* 255
**dx_stopch() — force termination of currently active I/O functions**

**EDX_SYSTEM**  
Error from operating system

### Example

```c
#include <srllib.h>
#include <dxxxlib.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 */
    }
}
```

### See Also

- dx_dial()
- dx_getdig()
- dx_play()
- dx_playf()
- dx_playouttdata()
- dx_playtone()
- dx_playvox()
- dx_rec()
- dx_recf()
- dx_recioytdata()
- dx_recvox()
- ATDX_TERMMSK()
- ATDX_CPTERM() - dx_dial() with call progress analysis
**dx_unlisten( )**

- **Name:** int dx_unlisten(chdev)
- **Inputs:**
  - int chdev • voice channel device handle
- **Returns:**
  - 0 on success
  - -1 on error
- **Includes:** srllib.h
dxxlib.h
- **Category:** TDM Routing
- **Mode:** synchronous

### Description

The `dx_unlisten( )` function disconnects the voice receive (listen) channel from the TDM bus.

Calling the `dx_listen( )` function to connect to a different TDM bus time slot automatically breaks an existing connection. Thus, when changing connections, you do not need to call the `dx_unlisten( )` function first.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <code>dx_open( )</code></td>
</tr>
</tbody>
</table>

### Cautions

This function will fail when an invalid channel device handle is specified.

### Errors

If the function returns -1, use the Standard Runtime Library (SRL) Standard Attribute function `ATDV_LASTERR( )` to obtain the error code or use `ATDV_ERRMSGP( )` to obtain a descriptive error message. One of the following error codes may be returned:

- **EDX_BADPARM**
  - Parameter error
- **EDX_SH_BADCMD**
  - Command is not supported in current bus configuration
- **EDX_SH_BADEXTTS**
  - TDM bus time slot is not supported at current clock rate
- **EDX_SH_BADINDX**
  - Invalid Switch Handler index number
- **EDX_SH_BADLCLTS**
  - Invalid channel number
dx_unlisten() — disconnect voice receive channel from TDM bus

EDX_SH_BADMODE
Function is not supported in current bus configuration

EDX_SH_BADTYPE
Invalid channel type (voice, analog, etc.)

EDX_SH_CMDBLOCK
Blocking command is in progress

EDX_SH_LCLDSCNCT
Channel is already disconnected from TDM bus

EDX_SH_LIBBSY
Switch Handler library is busy

EDX_SH_LIBNOTINIT
Switch Handler library is uninitialzed

EDX_SH_MISSING
Switch Handler is not present

EDX_SH_NOCLK
Switch Handler clock failback failed

EDX_SYSTEM
Error from operating system

## Example

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

main()
{
    int chdev;           /* Voice Channel device handle */

    /* Open board 1 channel 1 device */
    if ((chdev = dx_open("dxxxB1C1", 0)) == -1) {
        /* process error */
    }

    /* Disconnect receive of board 1, channel 1 from all TDM bus time slots */
    if (dx_unlisten(chdev) == -1) {
        printf("Error message = \%s", ATDV_ERRMSGP(chdev));
        exit(1);
    }
}
```

## See Also

- dx_listen()
nr_scroute( )

**Description**

The nr_scroute( ) convenience function makes a full or half-duplex connection between two devices connected to the time division multiplexing (TDM) bus.

This convenience function is not a part of any library and is provided in a separate C source file called sctools.c in the sctools subdirectory.

The nr_sc prefix to the function signifies network (analog and digital) devices and resource (voice, and fax) devices accessible via the TDM bus.

**Note:** Fax functionality may be conditionally compiled in or out of the function using the FAXSC defines in the makefile provided with the function. For example, to compile in fax functionality, link with the fax library. Error message printing may also be conditionally compiled in or out by using the PRINTON define in the makefile.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>devh1</td>
<td>specifies the valid channel device handle obtained when the channel was opened for the first device (the transmitting device for half duplex)</td>
</tr>
<tr>
<td>devtype1</td>
<td>specifies the type of device for devh1:</td>
</tr>
<tr>
<td></td>
<td>• SC_VOX – voice channel device</td>
</tr>
<tr>
<td></td>
<td>• SC_FAX – fax channel device</td>
</tr>
</tbody>
</table>
nr_scroute( ) — make a full or half-duplex connection

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>devh2</td>
<td>specifies the valid channel device handle obtained when the channel was opened for the second device (the listening device for half duplex)</td>
</tr>
<tr>
<td>devtype2</td>
<td>specifies the type of device for devh1. See devtype1 for a list of defines.</td>
</tr>
</tbody>
</table>
| mode      | specifies full or half-duplex connection. This parameter contains one of the following defines from sctools.h to specify full or half duplex:  
  * SC_FULLDUP – full-duplex connection (default)  
  * SC_HALFDUP – half-duplex connection  

When SC_HALFDUP is specified, the function returns with the second device listening to the TDM bus time slot connected to the first device.

**Cautions**

- The devtype1 and devtype2 parameters must match the types of the device handles in devh1 and devh2.
- If you have not defined FAXSC when compiling the sctools.c file, you cannot use this function to route fax channels.
- If you have not defined PRINTON in the makefile, errors will not be displayed.
- It is recommended that you do not use the nr_scroute( ) convenience function in high performance or high density applications because this convenience function performs one or more xx_getxmitslot invocations that consume CPU cycles unnecessarily.

**Errors**

None.

**Example**

See source code.

**Source Code**

The C source code for this function is provided in the sctools.c file located in the sctools subdirectory. On HMP, digital network interface (DTI) library functionality and modular station interface (MSI) library functionality are not supported.

```c
#include <stdio.h>
#include <varargs.h>
#include <srllib.h>
#include <dxxxlib.h>
#ifdef DTISC
#include <dtilib.h>
#endif
#ifdef FAXSC
#include <faxlib.h>
#endif
#include "sctools.h"
```
#if ( defined( __STDC__ ) || defined( __cplusplus ) )
int nr_scroute( int devh1, unsigned short devtype1,
    int devh2, unsigned short devtype2 unsigned char mode )
#else
int nr_scroute( devh1, devtype1, devh2, devtype2, mode )
#endif

int            devh1;
unsigned short devtype1;
int            devh2;
unsigned short devtype2;
unsigned char  mode;
#endif

{ SC_TSINFO sc_tsinfo; /* TDM bus time slot info structure */
  long scts; /* TDM bus time slot */

  /*
  * Setup the TDM bus time slot information structure.
  */
  sc_tsinfo.sc_numts = 1;
  sc_tsinfo.sc_tsarrayp = &scts;

  /*
  * Get the TDM bus time slot connected to the transmit of the first
  * device.
  */
  switch (devtype1)
  { case SC_VOX:

      if (dx_getxmitslot(devh1, &sc_tsinfo) == -1) {
          nr_scerror("nr_scroute: %s: dx_getxmitslot ERROR: %s\n",
              ATDV_NAMEP(devh1),ATDV_ERRMSGP(devh1));
          return -1;
      }
      break;

  case SC_LSI:
      if (ag_getxmitslot(devh1, &sc_tsinfo) == -1) {
          nr_scerror("nr_scroute: %s: ag_getxmitslot ERROR: %s\n",
              ATDV_NAMEP(devh1),ATDV_ERRMSGP(devh1));
          return -1;
      }
      break;

#ifndef DTISC
  case SC_DTI:
      if (dt_getxmitslot(devh1, &sc_tsinfo) == -1) {
          nr_scerror("nr_scroute: %s: dt_getxmitslot ERROR: %s\n",
              ATDV_NAMEP(devh1),ATDV_ERRMSGP(devh1));
          return -1;
      }
      break;
#endif

#ifndef FAXSC
  case SC_FAX:
      if (fx_getxmitslot(devh1, &sc_tsinfo) == -1) {
          nr_scerror("nr_scroute: %s: fx_getxmitslot ERROR: %s\n",
              ATDV_NAMEP(devh1),ATDV_ERRMSGP(devh1));
          return -1;
      }
      break;
#endif

make a full or half-duplex connection — nr_scroute( )
nr_scroute() — make a full or half-duplex connection

```c
default:
    nr_serror("nr_scroute: \%s: ERROR: Invalid 1st device type\n", ATDV_NAMEP(devh1));
    return -1;
}

/*
 * Make the second device type listen to the time slot that the first
 * device is transmitting on. If a half duplex connection is desired,
 * then return. Otherwise, get the TDM bus time slot connected to the
 * transmit of the second device.
 */
switch (devtype2) {
    case SC_VOX:
        if (dx_listen(devh2, &sc_tsinfo) == -1) {
            nr_serror("nr_scroute: \%s: dx_listen ERROR: \%s
", ATDV_NAMEP(devh2), ATDV_ERRMSGP(devh2));
            return -1;
        }
        if (mode == SC_HALFDUP) {
            return 0;
        }
        if (dx_getxmitslot(devh2, &sc_tsinfo) == -1) {
            nr_serror("nr_scroute: \%s: dx_getxmitslot ERROR: \%s
", ATDV_NAMEP(devh2), ATDV_ERRMSGP(devh2));
            return -1;
        }
        break;
    case SC_LSI:
        if (ag_listen(devh2, &sc_tsinfo) == -1) {
            nr_serror("nr_scroute: \%s: ag_listen ERROR: \%s
", ATDV_NAMEP(devh2), ATDV_ERRMSGP(devh2));
            return -1;
        }
        if (mode == SC_HALFDUP) {
            return 0;
        }
        if (ag_getxmitslot(devh2, &sc_tsinfo) == -1) {
            nr_serror("nr_scroute: \%s: ag_getxmitslot ERROR: \%s
", ATDV_NAMEP(devh2), ATDV_ERRMSGP(devh2));
            return -1;
        }
        break;
    #ifdef DTISC
    case SC_DTI:
        if (dt_listen(devh2, &sc_tsinfo) == -1) {
            nr_serror("nr_scroute: \%s: dt_listen ERROR: \%s
", ATDV_NAMEP(devh2), ATDV_ERRMSGP(devh2));
            return -1;
        }
        if (mode == SC_HALFDUP) {
            return 0;
        }
        if (dt_getxmitslot(devh2, &sc_tsinfo) == -1) {
            nr_serror("nr_scroute: \%s: dt_getxmitslot ERROR: \%s
", ATDV_NAMEP(devh2), ATDV_ERRMSGP(devh2));
            return -1;
        }
        break;
    #endif
```
#ifdef FAXSC
    case SC_FAX:
        if (fx_listen(devh2, &sc_tsinfo) == -1) {
            nr_scerror("nr_scroute: %s: fx_listen ERROR: %s\n", ATDV_NAMEP(devh2), ATDV_ERRMSGP(devh2));
            return -1;
        }
        if (mode == SC_HALFDUP) {
            return 0;
        }
        if (fx_getxmitslot(devh2, &sc_tsinfo) == -1) {
            nr_scerror("nr_scroute: %s: fx_getxmitslot ERROR: %s\n", ATDV_NAMEP(devh2), ATDV_ERRMSGP(devh2));
            return -1;
        }
        break;
    #endif
    default:
        nr_scerror("nr_scroute: %s: ERROR: Invalid 2nd device type\n", ATDV_NAMEP(devh2));
        return -1;
    }
    /*
    * Now make the first device listen to the transmit TDM bus time slot
    * of the second device.
    */
    switch (devtype1) {
    case SC_VOX:
        if (dx_listen(devh1, &sc_tsinfo) == -1) {
            nr_scerror("nr_scroute: %s: dx_listen ERROR: %s\n", ATDV_NAMEP(devh1), ATDV_ERRMSGP(devh1));
            return -1;
        }
        break;
    case SC_LSI:
        if (ag_listen(devh1, &sc_tsinfo) == -1) {
            nr_scerror("nr_scroute: %s: ag_listen ERROR: %s\n", ATDV_NAMEP(devh1), ATDV_ERRMSGP(devh1));
            return -1;
        }
        break;
    #ifdef DTISC
    case SC_DTI:
        if (dt_listen(devh1, &sc_tsinfo) == -1) {
            nr_scerror("nr_scroute: %s: dt_listen ERROR: %s\n", ATDV_NAMEP(devh1), ATDV_ERRMSGP(devh1));
            return -1;
        }
        break;
    #endif
    #ifdef FAXSC
    case SC_FAX:
        if (fx_listen(devh1, &sc_tsinfo) == -1) {
            nr_scerror("nr_scroute: %s: fx_listen ERROR: %s\n", ATDV_NAMEP(devh1), ATDV_ERRMSGP(devh1));
            return -1;
        }
        break;
    #endif
    }
    return 0;
}
nr_scroute() — make a full or half-duplex connection

static void nr_scerror(va_list)
    va_dcl
{
#define PRINTON
    va_list args;
    char *fmt;

    /*
     * Make args point to the 1st unnamed argument and then print
     * to stderr.
     */
    va_start(args);
    fmt = va_arg(args, char *);
    vfprintf(stderr, fmt, args);
    va_end(args);
#undef PRINTON
}

- See Also

- nr_scunroute()
nr_scunroute( )

**Name:** int nr_scunroute(devh1, devtype1, devh2, devtype2, mode)

**Inputs:**
- int devh1: valid channel device handle
- unsigned short devtype1: type of device for devh1
- int devh2: valid channel device handle
- unsigned short devtype2: type of device for devh2
- unsigned char mode: half or full duplex connection

**Returns:**
- 0 on success
- -1 on error

**Includes:**
- stdio.h
- varargs.h
- srllib.h
- dxxxlib.h
- faxlib.h (optional)
- sctools.h

**Category:** TDM Routing

**Mode:** synchronous

---

**Description**

The `nr_scunroute( )` convenience function breaks a full or half-duplex connection between two devices connected to the time division multiplexing (TDM) bus.

This convenience function is not a part of any library and is provided in a separate C source file called `sctools.c` in the sctools subdirectory.

The `nr_sc` prefix to the function signifies network (analog and digital) devices and resource (voice, and fax) devices accessible via the TDM bus.

**Note:** Fax functionality may be conditionally compiled in or out of the function using the FAXSC defines in the makefile provided with the function. For example, to compile in fax functionality, link with the fax library. Error message printing may also be conditionally compiled in or out by using the PRINTON define in the makefile.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>devh1</td>
<td>specifies the valid channel device handle obtained when the channel was opened for the first device (the transmitting device for half duplex)</td>
</tr>
<tr>
<td>devtype1</td>
<td>specifies the type of device for devh1:</td>
</tr>
<tr>
<td></td>
<td>• SC_VOX – voice channel device</td>
</tr>
<tr>
<td></td>
<td>• SC_FAX – fax channel device</td>
</tr>
</tbody>
</table>
**nr_scunroute( ) — break a full or half-duplex connection**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>devh2</td>
<td>specifies the valid channel device handle obtained when the channel was opened for the second device (the listening device for half duplex)</td>
</tr>
<tr>
<td>devtype2</td>
<td>specifies the type of device for <code>devh1</code>. See <code>devtype1</code> for a list of defines.</td>
</tr>
</tbody>
</table>
| mode | specifies full or half-duplex connection. This parameter contains one of the following defines from `sctools.h` to specify full or half duplex:  
  - SC_FULLDUP – full-duplex connection (default)  
  - SC_HALFDUP – half-duplex connection  
  When SC_HALFDUP is specified, the function returns with the second device listening to the TDM bus time slot connected to the first device. |

**Cautions**

- The `devtype1` and `devtype2` parameters must match the types of the device handles in `devh1` and `devh2`.
- If you have not defined FAXSC when compiling the `sctools.c` file, you cannot use this function to route fax channels.
- If you have not defined PRINTON in the makefile, errors will not be displayed.
- It is recommended that you do not use the `nr_scunroute( )` convenience function in high performance or high density applications because this convenience function performs one or more `xx_getxmitslot` invocations that consume CPU cycles unnecessarily.

**Errors**

None.

**Example**

See source code.

**Source Code**

The C source code for this function is provided in the `sctools.c` file located in the sctools subdirectory. On HMP, digital network interface (DTI) library functionality and modular station interface (MSI) library functionality are not supported.

```c
#include <stdio.h>
#include <varargs.h>
#include <srllib.h>
#include <dxxxlib.h>
#ifdef DTISC
#include <dtilib.h>
#endif
#ifdef FAXSC
#include <faxlib.h>
#endif
#include "sctools.h"
```

---

266

Voice API for HMP Library Reference — September 2004
```c
#include <stdlib.h>  // probably.

int nr_scunroute(int devh1, unsigned short devtype1,
                  int devh2, unsigned short devtype2, unsigned char mode)
{
    /*
    * Disconnect listen of second device from TDM bus listen time slot.
    * Switch (devtype2) {
    * case SC_VOX:
    *     if (dx_unlisten(devh2) == -1) {
    *         nr_scerror("nr_scunroute: %s: dx_unlisten ERROR: %s\n",
    *                     ATDV_NAMEP(devh2), ATDV_ERRMSGP(devh2));
    *         return -1;
    *     }
    *     break;
    * case SC_LSI:
    *     if (ag_unlisten(devh2) == -1) {
    *         nr_scerror("nr_scunroute: %s: ag_unlisten ERROR: %s\n",
    *                     ATDV_NAMEP(devh2), ATDV_ERRMSGP(devh2));
    *         return -1;
    *     }
    *     break;
    * #ifdef DTISC
    *     case SC_DTI:
    *         if (dt_unlisten(devh2) == -1) {
    *             nr_scerror("nr_scunroute: %s: dt_unlisten ERROR: %s\n",
    *                         ATDV_NAMEP(devh2), ATDV_ERRMSGP(devh2));
    *             return -1;
    *         }
    *         break;
    * #endif
    * #ifdef FAXSC
    *     case SC_FAX:
    *         if (fx_unlisten(devh2) == -1) {
    *             nr_scerror("nr_scunroute: %s: fx_unlisten ERROR: %s\n",
    *                         ATDV_NAMEP(devh2), ATDV_ERRMSGP(devh2));
    *             return -1;
    *         }
    *         break;
    * #endif
    * default:
    *     nr_scerror("nr_scunroute: %s: ERROR: Invalid 2nd device type\n",
    *                     ATDV_NAMEP(devh2));
    *     return -1;
    */
    
    /*
    * A half duplex connection has already been broken. If this is all
    * that is required, then return now.
    */
    if (mode == SC_HALFDUP) {
        return 0;
    }
}
```
/*
 * Disconnect listen of first device from TDM bus listen time slot
 */
switch (devtype1) {
    case SC_VOX:
        if (dx_unlisten(devh1) == -1) {
            nr_scerror("nr_scunroute: %s: dx_unlisten ERROR: %s
", ATDV_NAMEP(devh1), ATDV_ERRMSGP(devh1));
            return -1;
        }
        break;
    case SC_LSI:
        if (ag_unlisten(devh1) == -1) {
            nr_scerror("nr_scunroute: %s: ag_unlisten ERROR: %s
", ATDV_NAMEP(devh1), ATDV_ERRMSGP(devh1));
            return -1;
        }
        break;
#ifdef DTISC
    case SC_DTI:
        if (dt_unlisten(devh1) == -1) {
            nr_scerror("nr_scunroute: %s: dt_unlisten ERROR: %s
", ATDV_NAMEP(devh1), ATDV_ERRMSGP(devh1));
            return -1;
        }
        break;
#endif
    #ifdef FAXSC
    case SC_FAX:
        if (fx_unlisten(devh1) == -1) {
            nr_scerror("nr_scunroute: %s: fx_unlisten ERROR: %s
", ATDV_NAMEP(devh1), ATDV_ERRMSGP(devh1));
            return -1;
        }
        break;
    #endif
    default:
        nr_scerror("nr_scunroute: %s: ERROR: Invalid 1st device type\n", ATDV_NAMEP(devh1));
        return -1;
    }
    return 0;
}

static void nr_scerror(va_alist)
    va_dcl
{
    #ifdef PRINTON
    va_list args;
    char *fmt;
    /*
      * Make args point to the 1st unnamed argument and then print
      * to stderr.
      */
    va_start(args);
    fmt = va_arg(args, char *);
    vfprintf(stderr, fmt, args);
    va_end(args);
    #endif
}
break a full or half-duplex connection — nr_scunroute()
nr_scunroute() — break a full or half-duplex connection
3 Events

This chapter provides information on events that may be returned by the voice software. The following topics are discussed:

- **Overview of Events** .............................................................. 271
- **Termination Events** ............................................................ 271
- **Unsolicited Events** ............................................................. 273
- **Call Status Transition (CST) Events** ................................. 273

3.1 **Overview of Events**

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

Events in the voice library can be categorized as follows:

- termination events, which are produced when a function running in asynchronous mode terminates
- unsolicited events, which are not generated in response to the completion of a function. Rather, they are either generated in response to a condition of a given function or as a result of a call status transition (CST) condition that has been met.
- call status transition (CST) events, which indicate changes in the status of a call, such as rings or a tone detected, or the line going on-hook or off-hook. CST events are unsolicited events that are produced as a consequence of setting a CST mask.

For information on event handling, see the *Voice API Programming Guide*. For details on event management and event handling, see the *Standard Runtime Library API Programming Guide*.

3.2 **Termination Events**

Termination events are produced when a function running in asynchronous mode terminates. To collect termination event codes, use Standard Runtime Library (SRL) functions such as `sr_waitevt()` and `sr_enbhdlr()` depending on the programming model in use. For more information, see the Standard Runtime Library documentation.
The following termination events may be returned by the voice library:

**TDX_CALLP**
Termination event. Returned by `dx_dial()` to indicate that dialing with call progress analysis completed. Use `ATDX_CPTERM()` to determine the reason for termination.

**TDX_CST**
Termination event. Specifies a call status transition (CST) event. See Section 3.4, “Call Status Transition (CST) Events”, on page 273 for more information on these events.

**TDX_CREATETONE**
Termination event. Returned by `dx_createtone()` to indicate completion of create tone.

**TDX_CREATETONE_FAIL**
Termination event. Returned by `dx_createtone()` to indicate failure of create tone.

**TDX_DELETETONE**
Termination event. Returned by `dx_deletetone()` to indicate completion of delete tone.

**TDX_DELETETONE_FAIL**
Termination event. Returned by `dx_deletetone()` to indicate failure of delete tone.

**TDX_DIAL**
Termination event. Returned by `dx_dial()` to indicate that dialing without call progress analysis completed. Use `ATDX_TERMMSK()` to determine the reason for termination.

**TDX_ERROR**
Termination event. Returned by a function running in asynchronous mode to indicate an error. May also indicate that the TN_GEN tone generation template contains an invalid tg_dflag, or the specified amplitude or frequency is outside the valid range.

**TDX_GETDIG**
Termination event. Returned by `dx_getdig()` to indicate completion of asynchronous digit collection from a channel digit buffer.

**TDX_NOSTOP**
Termination event. Returned by `dx_stopch()` . On Linux, when issued on a channel that is already idle, `dx_stopch()` with EV_NOSTOP flag will return this event to indicate that no STOP was needed or issued.

**TDX_PLAY**
Termination event. Returned by play functions such as `dx_play()` to indicate completion of play.

**TDX_PLAYTONE**
Termination event. Returned by `dx_playtone()` and `dx_playtoneEx()` to indicate completion of play tone.

**TDX_QUERYTONE**
Termination event. Returned by `dx_querytone()` to indicate completion of query tone.

**TDX_QUERYTONE_FAIL**
Termination event. Returned by `dx_querytone()` to indicate failure of query tone.

**TDX_RECORD**
Termination event. Returned by record functions such as `dx_rec()` to indicate completion of record.
3.3 Unsolicited Events

Unsolicited events are produced in response to a condition of a given function or as a result of a call status transition (CST) condition that has been met. They are not generated in response to the completion of a function. For more information on CST events, see Section 3.4, “Call Status Transition (CST) Events”, on page 273.

The following non-CST unsolicited events may be returned by the voice library:

TDX_UNDERRUN
Unsolicited event. Generated when an underrun condition occurs during a streaming to board operation. This event is generated when the firmware (not the stream buffer) runs out of data. This event will only be generated when dx_setevtmsk() is set to DM_UNDERRUN. This works like a toggle key. If set once, the next call to the function will unset this mask.

TDX_LOWWATER
Unsolicited event. Generated when a low water mark is reached during a streaming to board operation.

TDX_HIGHWATER
Unsolicited event. Generated when a high water mark is reached during a streaming to board operation.

3.4 Call Status Transition (CST) Events

Call status transition (CST) events indicate changes in the status of a call, such as rings or a tone detected, or the line going on-hook or off-hook. A CST event is an unsolicited event that is produced as a consequence of setting a CST mask.

The dx_setevtmsk() function enables detection of CST events. User-defined tones are CST events, but detection for these events is enabled using dx_addtone() or dx_enbtone().

The dx_getevt() function retrieves CST events in a synchronous environment. Events are returned to DX_EBLK, on page 289. To retrieve CST events in an asynchronous environment, use the Standard Runtime Library (SRL) Event Management functions such as sr_getevtdatap(). Events are returned to the DX_CST structure.

The following CST events may be returned by the voice library:

DE_DIGITS
Call status transition event. Indicates digit received. Returned by dx_getdig().

Instead of getting digits from the DV_DIGIT structure using dx_getdig(), an alternative method is to enable the DE_DIGITS call status transition event using dx_setevtmsk() and get them from the DX_EBLK event queue data (ev_data) using dx_getevt() or from the DX_CST call status transition data (cst_data) using sr_getevtdatap().

DE_DIGOFF
Call status transition event. Specifies digit tone off event.

DE_SILOFF
Call status transition event. Indicates non-silence detected on the channel.
**Events**

DE_SILON
Call status transition event. Indicates silence detected on the channel.

DE_STOPGETEVT
Call status transition event. Indicates that the \texttt{dx_getevt()} function which was in progress has been stopped.

DE_TONEOFF
Call status transition event. Indicates tone off event received.

DE_TONEON
Call status transition event. Indicates tone on event received.

\textbf{Note:} Cadence tone on events are reported differently on HMP versus Springware boards. On HMP, if a cadence tone occurs continuously, a DE_TONEON event is reported for each on/off cycle. On Springware boards, a DE_TONEON event is reported for the first on/off cycle only. On HMP and on Springware boards, a DE_TONEOFF event is reported when the tone is no longer present.
This chapter provides an alphabetical reference to the data structures used by voice library functions. The following data structures are discussed:

- **CT_DEVINFO** .................................................. 276
- **DV_DIGIT** ...................................................... 278
- **DV_TPT** ........................................................ 279
- **DX_CAP** ....................................................... 285
- **DX_CST** ....................................................... 288
- **DX_EBLK** ..................................................... 289
- **DX_IOTT** ..................................................... 290
- **DX_STREAMSTAT** ........................................... 293
- **DX_SVCB** .................................................... 295
- **DX_SVMT** .................................................... 298
- **DX_UIO** ....................................................... 300
- **DX_XPB** ...................................................... 301
- **FEATURE_TABLE** ............................................ 303
- **SC_TSINFO** .................................................. 306
- **TN_GEN** ....................................................... 307
- **TN_GENCAD** ................................................ 308
- **TON_EDATA** ................................................ 310
typedef struct ct_devinfo {
  unsigned long  ct_prodid;       /* product ID */
  unsigned char  ct_devfamily;    /* device family */
  unsigned char  ct_devmode;      /* device mode */
  unsigned char  ct_nettype;      /* network interface */
  unsigned char  ct_busmode;      /* bus architecture */
  unsigned char  ct_busencoding;  /* bus encoding */
  union {
    unsigned char ct_RFU[7];     /* reserved */
    struct {
      unsigned char ct_prottype;
    } ct_net_devinfo;
  } ct_ext_devinfo;
} CT_DEVINFO;

## Description

The CT_DEVINFO data structure supplies information about a device. On return from the dx_getctinfo() function, CT_DEVINFO contains the relevant device and device configuration information.

The valid values for each field of the CT_DEVINFO structure are defined in ctinfo.h, which is referenced by dxxlib.h.

## Field Descriptions

The fields of the CT_DEVINFO data structure are described as follows:

- **ct_prodid**
  - Contains a valid product identification number for the device.

- **ct_devfamily**
  - Specifies the device family. Possible values are:
    - CT_DFDM3 – DM3 device
    - CT_DFHMPDM3 – HMP device (Host Media Processing)

- **ct_devmode**
  - Specifies the device mode. Possible values are:
    - CT_DMRESOURCE – voice device
    - CT_DMNETWORK – network device

- **ct_nettype**
  - Specifies the type of network interface for the device. Possible values are:
    - CT_NTIPT – IP connectivity

- **ct_busmode**
  - Specifies the bus architecture used to communicate with other devices in the system. Possible values are:
    - CT_BMSCBUS – TDM bus architecture
    - CT_H100 – H.100 bus
    - CT_H110 – H.110 bus
ct_busencoding
Describes the PCM encoding used on the bus. Possible values are:
- CT_BEULAW – mu-law encoding
- CT_BEALAW – A-law encoding
- CT_BELLAW – linear encoding
- CT_BEBYPASS – encoding is being bypassed

c_t_ext_devinfo.ct_RFU
Not used in HMP.

c_t_ext_devinfo.ct_net_devinfo.ct_prottype
Not used in HMP.

Example

For an example of how to use the CT_DEVINFO structure, see the Example section for dx_getctinfo().
**DV_DIGIT**

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;

- **Description**

The DV_DIGIT data structure stores an array of digits. When `dx_getdig()` is called, 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 DG_MAXDIGS define in `dxxxlib.h` indicates the maximum number of digits that can be returned by a single call to `dx_getdig()`. The maximum size of the digit buffer varies with the board type and technology.

- **Field Descriptions**

  The fields of the DV_DIGIT data structure are described as follows:

  - **dg_value**
    Specifies a null-terminated string of the ASCII values of the digits collected.

  - **dg_type**
    Specifies an array (terminated by DG_END) of the digit types that correspond to each of the digits contained in the dg_value string.

    Use the following defines to identify the digit type:
    - `DG_DTMF_ASCII` – DTMF
    - `DG_MF_ASCII` – MF
    - `DG_USER1` – GTD user-defined
    - `DG_USER2` – GTD user-defined
    - `DG_USER3` – GTD user-defined
    - `DG_USER4` – GTD user-defined
    - `DG_USER5` – GTD user-defined
    - `DG_END` – Terminator for dg_type array

- **Example**

  For an example of how to use this data structure, see the Example section for `dx_getdig()`.
DV_TPT

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;

■ Description

The DV_TPT data structure specifies a termination condition for an I/O function. To specify multiple termination conditions for a function, use multiple DV_TPT structures configured as a linked list, an array, or a combined linked list and array, with each DV_TPT specifying a termination condition. The first termination condition that is met will terminate the I/O function.

For a list of functions in the I/O category, see Chapter 1, “Function Summary by Category”. For more information on termination conditions, see the I/O terminations topic in the Voice API Programming Guide.

The DV_TPT structure is defined in the Standard Runtime Library (srllib.h).

Notes: 1. Not all termination conditions are supported by all I/O functions. Exceptions are noted in the description of the termination condition.

2. Use the dx_clrtpt( ) function to clear the field values of the DV_TPT structure before using this structure in a function call. This action prevents possible corruption of data in the allocated memory space.

■ Field Descriptions

The fields of the DV_TPT data structure are described as follows:

tp_type
    Describes whether the structure is part of a linked list, part of an array, or the last DV_TPT entry in the DV_TPT table. Specify one of the following values:

    • IO_CONT – next DV_TPT entry is contiguous in an array
    • IO_EOT – last DV_TPT in the chain
    • IO_LINK – tp_nextp points to next DV_TPT structure in linked list

tp_termno
    Specifies a condition that will terminate an I/O function.

The supported termination conditions are:

    • DX_DIGMASK – digit termination for a bit mask of digits received
    • DX_DIGTYPE – digit termination for user-defined tone. The ASCII value set in the tp_length field must match a real DTMF tone (0-9, a-d, *, #).
    • DX_IDDTIME – maximum delay between digits. This termination condition is only supported by the dx_getdig( ) function.
    • DX_MAXDTMF – maximum number of digits received
DV_TPT — termination parameter table

- DX_MAXSIL – maximum length of silence. The range is 10 msec to 250 sec (25000 in 10 msec units).
- DX_MAXTIME – maximum function time. This termination condition is not supported by tone generation functions such as dx_playtone() and dx_playtoneEx().
- DX_TONE – tone on or tone off termination for global tone detection (GTD)

**Note:** If you specify DX_IDDTIME in tp_termno, then you must specify TF_IDDTIME in tp_flags. Similarly, if you specify DX_MAXTIME in tp_termno, then you must specify TF_MAXTIME in tp_flags.

**Note:** It is not valid to set both DX_MAXTIME and DX_IDDTIME to 0. If you do so and no other termination conditions are set, the function will never terminate.

You can call the extended attribute function ATDX_TERMMSK() to determine all the termination conditions that occurred. This function returns a bitmap of termination conditions. The “TM_” defines corresponding to this bitmap of termination conditions are provided in the function description for ATDX_TERMMSK().

**tp_length**

Refers to the length or size for each specific termination condition. When tp_length represents length of time for a termination condition, the maximum value allowed is 60000. This field can represent the following:

- time in 10 or 100 msec units – Applies to any termination condition that specifies termination after a specific period of time, up to 60000. Units is specified in tp_flags field. Default units is 100 msec.
- number 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 topic in the Voice API Programming Guide for information.
- digit bit mask – Applies to DX_DIGMASK, which specifies a bit mask of digits to terminate on. Set the digit bit mask using one or more of the appropriate “Digit Defines” from the table below:

<table>
<thead>
<tr>
<th>Digit</th>
<th>Digit Define</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DM_0</td>
</tr>
<tr>
<td>1</td>
<td>DM_1</td>
</tr>
<tr>
<td>2</td>
<td>DM_2</td>
</tr>
<tr>
<td>3</td>
<td>DM_3</td>
</tr>
<tr>
<td>4</td>
<td>DM_4</td>
</tr>
<tr>
<td>5</td>
<td>DM_5</td>
</tr>
<tr>
<td>6</td>
<td>DM_6</td>
</tr>
<tr>
<td>7</td>
<td>DM_7</td>
</tr>
<tr>
<td>8</td>
<td>DM_8</td>
</tr>
<tr>
<td>9</td>
<td>DM_9</td>
</tr>
<tr>
<td>*</td>
<td>DM_S</td>
</tr>
<tr>
<td>#</td>
<td>DM_P</td>
</tr>
<tr>
<td>a</td>
<td>DM_A</td>
</tr>
<tr>
<td>b</td>
<td>DM_B</td>
</tr>
</tbody>
</table>
tp_flags
A bit mask representing various characteristics of the termination condition to use. The defines for the termination flags are:

- **TF_10MS** – Set units of time for tp_length to 10 msec. If not set, the default unit is 100 msec.
- **TF_CLRBEG** – History of this termination condition is cleared when the function begins. This bit overrides the TF_LEVEL bit. If both are set, the history will be cleared and no past history of this terminator will be taken into account.
- **TF_CLREND** – History of this termination condition is cleared when the function terminates. This bit has special meaning for DX_IDDTIME (interdigit delay). If set, the terminator will be started after the first digit is received; otherwise, the terminator will be started as soon as the function is started. This bit has no effect on HMP and will be ignored.
- **TF_EDGE** – Termination condition is edge-sensitive. Edge-sensitive means that the function will not terminate unless the condition occurs after the function starts. Refer to the table later in this section to see which termination conditions can be edge-sensitive and which can be level-sensitive. This bit has no effect on HMP and will be ignored.
- **TF_FIRST** – This bit is only used for DX_IDDTIME termination. If set, start looking for termination condition (interdigit delay) to be satisfied after first digit is received.
- **TF_IMMEDIATE** – This bit is only used for DX_MAXSIL and DX_MAXNOSIL termination. If set, the silence timer starts immediately at the onset of `ec_stream()` or `ec_recottdata()` instead of waiting for `dx_play()` to finish. For more information on `ec` functions, see the *Continuous Speech Processing API Library Reference*.
- **TF_LEVEL** – Termination condition is level-sensitive. Level-sensitive means that if the condition is satisfied when the function starts, termination will occur immediately. Termination conditions that can be level-sensitive have a history associated with them which records the state of the terminator before the function started. Refer to the table later in this section to see which termination conditions can be edge-sensitive and which can be level-sensitive. This bit has no effect on HMP and will be ignored.
- **TF_SETINIT** – 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 tp_data value 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.
- **TF_USE** – Terminator used for termination. 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 TF_CLRBEG and TF_CLREND bits), but the terminator will still not be used for termination. This bit is not valid for the following termination conditions: DX_DIGMASK
  DX_IDDTIME
DX_MAXTIME

A set of default tp_flags values appropriate to the various termination conditions is also available. These default values are:

<table>
<thead>
<tr>
<th>Default Define</th>
<th>Underlying Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>TF_DIGMASK</td>
<td>(TF_LEVEL)</td>
</tr>
<tr>
<td>TF_DIGTYPE</td>
<td>(TF_LEVEL)</td>
</tr>
<tr>
<td>TF_IDDTIME</td>
<td>(TF_EDGE)</td>
</tr>
<tr>
<td>TF_MAXDTMF</td>
<td>(TF_LEVEL</td>
</tr>
<tr>
<td>TF_MAXNOSIL</td>
<td>(TF_EDGE</td>
</tr>
<tr>
<td>TF_MAXSIL</td>
<td>(TF_EDGE</td>
</tr>
<tr>
<td>TF_MAXTIME</td>
<td>(TF_EDGE)</td>
</tr>
<tr>
<td>TF_TONE</td>
<td>(TF_LEVEL</td>
</tr>
</tbody>
</table>

Note: If you specify TF_IDDTIME in tp_flags, then you must specify DX_IDDTIME in tp_termno. Similarly, if you specify TF_MAXTIME in tp_flags, then you must specify DX_MAXTIME 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:

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>rfu</td>
<td>rfu</td>
<td>units</td>
<td>ini</td>
<td>use</td>
<td>beg</td>
<td>end</td>
<td>level</td>
</tr>
</tbody>
</table>

The following table shows the default sensitivity of a termination condition.

<table>
<thead>
<tr>
<th>Termination Condition</th>
<th>Level-sensitive</th>
<th>Edge-sensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>DX_DIGMASK</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>DX_DIGTYPE</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>DX_IDDTIME</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>DX_MAXDTMF</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>DX_MAXNOSIL</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>DX_MAXSIL</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>DX_MAXTIME</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>DX_TONE</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

tp_data

Specifies optional additional data. This field can be used as follows:

- If tp_termno contains DX_MAXSIL, tp_data can specify the initial length of silence to terminate on.
- If tp_termno contains DX_TONE, tp_data can specify one of the following values:
  - DX_TONEOFF (for termination after a tone-off event)
  - DX_TONEON (for termination after a tone-on event)

tp_nextp

Points to the next DV_TPT structure in a linked list if the tp_type field is set to IO_LINK.
Table 7 indicates how DV_TPT fields should be filled. In the table, 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 default defines for the tp_flags field are listed in the description of the tp_flags, above. To override defaults, set the bits in tp_flags individually, as required.

Table 7. DV_TPT Field Settings Summary

<table>
<thead>
<tr>
<th>tp_termno</th>
<th>tp_type</th>
<th>tp_length</th>
<th>tp_flags: not set</th>
<th>tp_flags: set</th>
<th>tp_data</th>
<th>tp_nextp</th>
</tr>
</thead>
<tbody>
<tr>
<td>DX_MAXDTMF</td>
<td>IO_LINK</td>
<td>max number of digits</td>
<td>bit 0: TF_EDGE</td>
<td>TF_LEVEL*</td>
<td>N/A</td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td>IO_EOT</td>
<td></td>
<td>bit 1: no clr*</td>
<td>TF_CLREN</td>
<td>N/A</td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td>IO_CONT</td>
<td></td>
<td>bit 2: no clr*</td>
<td>TF_CLRBEG</td>
<td>N/A</td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 3: clr hist</td>
<td>TF_USE*</td>
<td>N/A</td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td>DX_MAXSIL</td>
<td>IO_LINK</td>
<td>max length silence</td>
<td>bit 0: TF_EDGE*</td>
<td>N/A</td>
<td>N/A</td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td>IO_EOT</td>
<td></td>
<td>bit 1: no clr*</td>
<td>N/A</td>
<td>N/A</td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td>IO_CONT</td>
<td></td>
<td>bit 2: no clr*</td>
<td>N/A</td>
<td>N/A</td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 3: clr hist</td>
<td>N/A</td>
<td>N/A</td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 4: no-setinit</td>
<td>N/A</td>
<td>N/A</td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 5: 100 msec*</td>
<td>N/A</td>
<td>N/A</td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td>DX_IDDTIME</td>
<td>IO_LINK</td>
<td>max length interdigit delay</td>
<td>bit 0: TF_EDGE*</td>
<td>TF_CLREN</td>
<td>N/A</td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td>IO_EOT</td>
<td></td>
<td>bit 1: start@call*</td>
<td>TF_CLRBEG</td>
<td>N/A</td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td>IO_CONT</td>
<td></td>
<td>bit 2: N/A</td>
<td>TF_USE*</td>
<td>N/A</td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 3: N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 4: N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 5: 100 msec*</td>
<td>N/A</td>
<td>N/A</td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td>DX_MAXTIME</td>
<td>IO_LINK</td>
<td>max length function time</td>
<td>bit 0: TF_EDGE*</td>
<td>N/A</td>
<td>N/A</td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td>IO_EOT</td>
<td></td>
<td>bit 1: N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td>IO_CONT</td>
<td></td>
<td>bit 2: N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 3: N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 4: N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 5: 100 msec*</td>
<td>N/A</td>
<td>N/A</td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td>DX_DIGMASK</td>
<td>IO_LINK</td>
<td>bit 0: d (set)</td>
<td>bit 1: 1</td>
<td>TF_LEVEL*</td>
<td>N/A</td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td>IO_EOT</td>
<td>bit 2: 2</td>
<td>bit 1: 1</td>
<td></td>
<td></td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td>IO_CONT</td>
<td>bit 3: 3</td>
<td>bit 2: 2</td>
<td></td>
<td></td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 4: 4</td>
<td>bit 3: 3</td>
<td></td>
<td></td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 5: 5</td>
<td>bit 4: 4</td>
<td></td>
<td></td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 6: 6</td>
<td>bit 5: 5</td>
<td></td>
<td></td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 7: 7</td>
<td>bit 6: 6</td>
<td></td>
<td></td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 8: 8</td>
<td>bit 7: 7</td>
<td></td>
<td></td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 9: 9</td>
<td>bit 8: 8</td>
<td></td>
<td></td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 10: 0</td>
<td>bit 9: 9</td>
<td></td>
<td></td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 11: -</td>
<td>bit 10: 0</td>
<td></td>
<td></td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 12: #</td>
<td>bit 11: -</td>
<td></td>
<td></td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 13: a</td>
<td>bit 12: #</td>
<td></td>
<td></td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 14: b</td>
<td>bit 13: a</td>
<td></td>
<td></td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 15: c</td>
<td>bit 14: b</td>
<td></td>
<td></td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
</tbody>
</table>
Table 7. DV_TPT Field Settings Summary (Continued)

<table>
<thead>
<tr>
<th>tp_termno</th>
<th>tp_type</th>
<th>tp_length</th>
<th>tp_flags: not set</th>
<th>tp_flags: set</th>
<th>tp_data</th>
<th>tp_nextp</th>
</tr>
</thead>
<tbody>
<tr>
<td>DX_TONE</td>
<td>IO_LINK</td>
<td>Tone ID</td>
<td>bit 0: TF_EDGE</td>
<td>TF_LEVEL*</td>
<td>DX_TONEON</td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td>IO_EOT</td>
<td></td>
<td>bit 1: no clr</td>
<td>TF_CRLREN*</td>
<td>DX_TONEOFF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IO_CONT</td>
<td></td>
<td>bit 2: no clr*</td>
<td>TF_CLRBEG</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 3: clr hist</td>
<td>TF_USE*</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DX_DIGTYPE</td>
<td>IO_LINK</td>
<td>low byte:</td>
<td>bit 0: TF_EDGE</td>
<td>TF_LEVEL</td>
<td>N/A</td>
<td>pointer to next DV_TPT if linked list</td>
</tr>
<tr>
<td></td>
<td>IO_EOT</td>
<td>ASCII val.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IO_CONT</td>
<td>*hi byte:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>digit type</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example**

See `dx_playiottdata()` and `dx_recioottdata()` for an example of how to use the DV_TPT structure.
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_lo1tolb;        /* % acceptable neg. dev of short low sig. */
    unsigned short ca_lo2tolb;        /* % acceptable pos. dev of long low sig. */
    unsigned short ca_lcdly1;         /* Delay after lc drop con. Before msg. */
    unsigned short ca_lcdly2;         /* 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 neg. dev of short low sig. */
    unsigned short ca_lo2tola;        /* % acceptable pos. dev of long low sig. */
    unsigned short ca_lo1tolb;        /* % acceptable neg. dev of short low sig. */
    unsigned short ca_lo2tolb;        /* % acceptable pos. 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 lst high for busy */
    unsigned short ca_nbnsbusy;       /* 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 ca_silfltr;        /* reserved for future use */
    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_lower2frq;      /* Lower allowable frequency in Hz. */
    unsigned short ca_upper2frq;      /* Upper allowable frequency in Hz. */
    unsigned short ca_time2frq;       /* 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_lower3frq;      /* Lower allowable frequency in Hz. */
    unsigned short ca_upper3frq;      /* Upper allowable frequency in Hz. */
    unsigned short ca_time3frq;       /* 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_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=100msec) */
    unsigned short ca_pamd_falltime;   /* Wait for PAMD/PVD after cadence break (def=4s) */
    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 1st ring (def=30s) */
    unsigned short ca_maxintering;    /* Max inter ring delay before connect (8sec) */
} DX_CAP;
**DX_CAP — call progress analysis parameters**

### Description

The DX_CAP data structure contains call progress analysis parameters.

The DX_CAP structure modifies parameters that control frequency detection, cadence detection, loop current, positive voice detection (PVD), and positive answering machine detection (PAMD). The DX_CAP structure is used to modify call progress analysis channel parameters when using `dx_dial()`.

For more information about call progress analysis as well as how and when to use the DX_CAP structure, see the *Voice API Programming Guide*.

**Notes:**

1. Use the `dx_clrcap()` function to clear the field values of the DX_CAP structure before using this structure in a function call. This action prevents possible corruption of data in the allocated memory space.

2. 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 previous call to `dx_dial()` is ignored.

### Field Descriptions

The following fields of the DX_CAP data structure are supported:

- **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.

  Length: 2  Default: 4000  Units: 10 msec

- **ca_intflg**

  Intercept Mode Flag. Enables or disables SIT frequency detection, positive voice detection (PVD), and/or positive answering machine detection (PAMD), and selects the mode of operation for SIT frequency detection.

  - DX_OPTDIS – Disable SIT frequency detection, PAMD, and PVD.
  
    This setting provides call progress without SIT frequency detection.

  - DX_OPTNOCON – Enable SIT frequency detection and return an “intercept” immediately after detecting a valid frequency.

    This setting provides call progress with SIT frequency detection.

  - DX_PVDENABLE – Enable PVD and fax tone detection.

    This setting provides PVD call analysis only (no call progress).

  - DX_PVDOPTNOCON – Enable PVD, DX_OPTNOCON, and fax tone detection.

    This setting provides call progress with SIT frequency detection and PVD call analysis.

  - DX_PAMDENABLE – Enable PAMD, PVD, and fax tone detection.

    This setting provides PAMD and PVD call analysis only (no call progress).

  - DX_PAMDOPTEN – Enable PAMD, PVD, DX_OPTNOCON, and fax tone detection.

    This setting provides full call progress and call analysis.

  Length: 1  Default: DX_OPTNOCON

- **ca_noanswer**

  No Answer. Length of time to wait after first ringback before deciding that the call is not answered.

  Default: 3000  Units: 10 msec
call progress analysis parameters — DX_CAP

ca_pamd_failtime
   PAMD Fail Time. Maximum time to wait for positive answering machine detection or positive voice detection after a cadence break.
   Default: 400  Units: 10 msec

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
   • PAMD_ACCU – Recommended setting. Does the most accurate evaluation detecting live voice as accurately as PAMD_FULL but is more accurate than PAMD_FULL (although slightly slower) in detecting an answering machine. Use PAMD_ACCU when accuracy is more important than speed.
   Default: PAMD_ACCU

Example

For an example of DX_CAP, see the Example section for dx_dial().
DX_CST — call status transition (CST) information

dx_cst — call status transition (CST) information

typedef struct DX_CST {
    unsigned short cst_event;
    unsigned short cst_data;
} DX_CST;

- Description

The DX_CST data structure contains parameters for call status transition.

DX_CST contains call status transition information after an asynchronous TDX_CST termination event occurs. Use Standard Runtime Library (SRL) Event Management function, sr_getevtdatap(), to retrieve the structure.

- Field Descriptions

The fields of the DX_CST data structure are described as follows:

cst_event
- Contains the event type.
  - Use the following defines to identify the event type:
    - DE_DIGITS – digit received
    - DE_DIGOFF – digit tone-off event
    - DE_SILOFF – non-silence detected
    - DE_SILON – silence detected
    - DE_STOPGETEVT – dx_getevt() stopped
    - DE_TONEOFF – tone off event
    - DE_TONEON – tone on event

cst_data
- Contains data associated with the CST event.
  - The data are described for each event type as follows:
    - DE_DIGITS – ASCII digit (low byte) and the digit type (high byte)
    - DE_DIGOFF – digit tone-off event
    - DE_SILOFF – time since previous silence started in 10 msec units
    - DE_SILON – time since previous silence stopped in 10 msec units
    - DE_STOPGETEVT – monitoring of channels for call status transition events has been stopped
    - DE_TONEOFF – user-specified tone ID
    - DE_TONEON – user-specified tone ID

- Example

For an example of how to use the DX_CST structure, see the Example section for dx_sendevt() and dx_setevtsmk().
**DX_EBLK**

```c
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;
```

**Description**

The DX_EBLK data structure contains parameters for the Call Status Event Block. This structure is returned by `dx_getevt()` and indicates which call status transition event occurred. `dx_getevt()` is a synchronous function which blocks until an event occurs. For information about asynchronously waiting for CST events, see `dx_setevtsk()`.

**Field Descriptions**

The fields of the DX_EBLK data structure are described as follows:

- **ev_event**
  - Contains the event type.
  - Use the following defines to identify the event type:
    - `DE_DIGITS` – digit received
    - `DE_SILOFF` – non-silence detected
    - `DE_SILON` – silence detected
    - `DE_TONEOFF` – tone off event
    - `DE_TONEON` – tone on event

- **ev_data**
  - Contains data associated with the CST event. All durations of time are in 10 msec units.
  - The data are described for each event type as follows:
    - `DE_DIGITS` – ASCII digit (low byte) and the digit type (high byte)
    - `DE_SILOFF` – length of time that silence occurred before non-silence (noise or meaningful sound) was detected
    - `DE_SILON` – length of time that non-silence occurred before silence was detected
    - `DE_TONEOFF` – user-specified tone ID for the tone-off event
    - `DE_TONEON` – user-specified tone ID for the tone-on event

**Example**

For an example of how to use the DX_EBLK structure, see the Example section for `dx_getevt()` and `dx_setevtsk()`.


DX_IOTT — input/output transfer table

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;      /* Pointer to next DX_IOTT if IO_LINK set */
    DX_IOTT          *io_prevp;      /* (Optional) Pointer to previous DX_IOTT */
}DX_IOTT;

### Description

The DX_IOTT data structure contains parameters for input/output transfer. The DX_IOTT structure identifies a source or destination for voice data. It is used with various play and record functions, such as `dx_play()` and `dx_rec()`, as well as other categories of functions.

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 Linux or Windows device interface. The device must support `open()`, `close()`, `read()`, and `write()`.

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.

**Note:** The DX_IOTT data area must remain in scope for the duration of the function if running asynchronously.

### Field Descriptions

The fields of the DX_IOTT data structure are described as follows:

**io_type**

This field is a bitmap that 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. It is also used to enable WAVE data offset I/O. Set the io_type field to an OR combination of the following defines.

Specify the data transfer type as follows:
- **IO_DEV** – file data
- **IO_MEM** – memory data
- **IO_STREAM** – data for streaming to board
- **IO_UIO** – nonstandard storage media data using the `dx_setuio()` function; must be ORed with **IO_DEV**

Specify the 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 no value is specified, **IO_CONT** is assumed.

Other Types:
• IO_USEOFFSET – enables use of the io_offset and io_length fields for WAVE data
To enable offset I/O for WAVE data, set the DX_IOTT io_type field to IO_USEOFFSET ORed with the IO_DEV define (to indicate file data rather than memory buffer).

Note: Wave files cannot be recorded to memory buffers or played from memory buffers.

io_fhandle
In Linux, specifies 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.
In Windows, specifies a unique file descriptor provided by the `dx_fileopen()` function 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
Specifies a base memory address if IO_MEM is set in io_type.

io_offset
Specifies one of the following:
• if IO_DEV is specified in io_type, an offset from the beginning of a file
• for WAVE file offset I/O (IO_DEV is ORed with IO_USEOFFSET in io_type), a file offset value that is calculated from the beginning of the WAVE audio data rather than the beginning of the file (that is, the first 80 bytes that make up the file header are not counted).
• if IO_MEM is specified in io_type, an offset from the base buffer address specified in io_bufp

io_length
Specifies 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()` is called. The io_prevp field of the first DX_IOTT structure is set to NULL.

Example

The following example uses different sources for playback, an array or linked list of DX_IOTT structures.

```c
#include <srllib.h>
#include <dxxxlib.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. */

typedef struct streamStat
{
    unsigned int version;            // version of the structure
    unsigned int bytesIn;            // total number of bytes put into stream buffer
    unsigned int bytesOut;           // total number of bytes sent to board
    unsigned int headPointer;        // internal pointer to position in stream buffer
    unsigned int tailPointer;        // internal pointer to position in stream buffer
    unsigned int currentState;       // idle, streaming etc.
    unsigned int numberOfBufferUnderruns;
    unsigned int numberOfBufferOverruns;
    unsigned int BufferSize;         // buffer size
    unsigned int spaceAvailable;     // space in bytes available in stream buffer
    unsigned int highWaterMark;      // high water mark for stream buffer
    unsigned int lowWaterMark;       // low water mark for stream buffer
} DX_STREAMSTAT;

Description

The DX_STREAMSTAT data structure contains the current status of the circular stream buffer for a voice device. This structure is used by the streaming to board feature and returned by the dx_GetStreamInfo() function. This structure is defined in dxxxlib.h.

Field Descriptions

The fields of the DX_STREAMSTAT data structure are described as follows:

version
    Contains the version of the data structure. The value is currently hardcoded to 1. This field is reserved for future use.

bytesIn
    Contains the total number of bytes put into the circular stream buffer.

bytesOut
    Contains the total number of bytes sent to the board.

headPointer
    Contains an internal pointer to the head position in the circular stream buffer.

tailPointer
    Contains an internal pointer to the tail position in the circular stream buffer.

currentState
    Contains the current state of the circular stream buffer.
    • ASSIGNED_STREAM_BUFFER – stream buffer is in use by a play operation and therefore is not available to any other play operation at this time
    • UNASSIGNED_STREAM_BUFFER – stream buffer is free to be used by a play operation at this time

numberOfBufferUnderruns
    Represents the number of times the host library tries to read from the circular stream buffer and finds that there is not enough data to satisfy that read request to send the data to the firmware. The size of the read request for the host library is determined by the transfer buffer size of the player.
DX_STREAMSTAT — status of stream buffer

- **numberOfBufferOverruns**
  Represents the number of times the application tries to write the data into the buffer beyond the circular stream buffer limit.

- **BufferSize**
  Contains the total size of the circular stream buffer.

- **spaceAvailable**
  Specifies the space, in bytes, available in the circular stream buffer.

- **highWaterMark**
  Specifies the high point in the circular stream buffer used to signal an event.

- **lowWaterMark**
  Specifies the low point in the circular stream buffer used to signal an event.

**Example**

See `dx_GetStreamInfo()` for an example of how to use the DX_STREAMSTAT structure.
**DX_SVCB**

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;

### Description

The DX_SVCB data structure contains parameters for the speed and volume adjustment condition block. Speed adjustment is not supported on HMP.

This structure is used by `dx_setsvcond()` function to specify a play adjustment condition that is added to the internal volume condition table (SVCT). The play adjustment conditions in the SVCT are used to adjust volume automatically at the beginning of playback or in response to digits entered by the user during playback.

The `dx_setsvcond()` and `dx_addvoldig()` functions can be used to add play adjustment conditions to the SVCT. These functions tie a volume adjustment to an external event, such as a DTMF digit.

You cannot change an existing adjustment condition in the SVCT without using the `dx_clrsvcond()` function to clear the SVCT of all conditions and then adding a new set of adjustment conditions to the SVCT.

This structure is used to specify the following:
- table type (volume modification table)
- adjustment type (step, index, toggle)
- adjustment size or action
- adjustment condition (incoming digit, beginning of play)
- level/edge sensitivity for incoming digits

For more information on the volume modification table, see the *Voice API Programming Guide*.

### Field Descriptions

The fields of the DX_SVCB data structure are described as follows:

**type**

**Type of Playback Adjustment**: specifies an OR combination of the following:

- **Adjustment Table Type** (required): specifies the volume adjustment type
  - SV_VOLUMETBL – selects volume table to be modified
- **Adjustment Method** (required): specifies one adjustment method (step, index, or toggle), which also determines how the adjsize value is used
• **SV_ABSPOS – Index Mode**: Sets adjsize field to specify an absolute adjustment position (index) in the volume modification table. The index value can be from -10 to +10, based on position 0, the origin, or center, of the table.

**Note:** In the volume modification table, the default entries for index values +6 to +10 are -128 which represent a null-entry. To customize the table entries, use the `dx_setsvmt()` function.

• **SV_RELCURPOS – Step Mode**: Sets adjsize field to specify a number of steps by which to adjust the volume relative to the current position in the table. Specify a positive number of steps to increase the current volume, or a negative number of steps to decrease it. For example, specify -2 to lower the volume by two steps in the volume modification table.

• **SV_TOGGLE – Toggle Mode**: Sets adjsize field to specify one of the toggle defines, which control the values for the current and last-modified volume settings and allow you to toggle the volume between standard (the origin) and any setting selected by the user. See the description of the adjsize field for the toggle defines.

**Options** specifies one or no options from the following:

• **SV_LEVEL – Level**: Sets the digit adjustment condition to be level-sensitive.

  On Linux, 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.

  On Windows, at the start of play, existing digits in the digit buffer will be checked to see if they are level-sensitive play adjustment digits. If the first digit in the buffer is a level-sensitive play adjustment digit, it will cause a play adjustment and be removed from the buffer. Subsequent digits in the buffer will be treated the same way until the first occurrence of any digit that is not an **SV_LEVEL** play adjustment digit. If **SV_LEVEL** is not specified, the digit adjustment condition is edge-sensitive. Existing edge-sensitive play adjustment digits in the digit buffer will not cause a play adjustment; but after the playback starts, edge-sensitive digits will cause a play adjustment.

• **SV_BEGINPLAY – Automatic**: Sets the play adjustment to occur automatically at the beginning of the next playback. This sets a volume level without using a digit condition. The digit and digtype fields are ignored.

**adjsize**

**Adjustment Size**: Specifies the adjustment size. The valid values follow according to the adjustment method:

**For Index Mode** (**SV_ABSPOS** in type field)

  an integer from -10 to +10 representing an absolute position in the SVMT

**For Step Mode** (**SV_RELCURPOS** in type field)

  a positive or negative integer representing the number of steps to adjust the level relative to the current setting in the SVMT

**For Toggle Mode** (**SV_TOGGLE** in type field)

The following are valid values:

• **SV_TOGORIGIN** – sets the digit to toggle between the origin and the last modified volume level (for example, between the -5 and 0 levels)

• **SV_CURORIGIN** – resets the current volume level to the origin (same effect as **SV_ABSPOS** with adjsize 0)

---

**DX_SVCB — speed and volume adjustment condition block**
digit

**Digit**: Specifies an ASCII digit that will adjust the play.

Values: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, #, *

digtype

**Digit Type**: Specifies the type of digit:
- DG_DTMF – DTMF digits

### Example

This example illustrates how to set a DTMF digit to adjust playback volume. The following DX_SVCB structure is set to decrease the volume by one step whenever the DTMF digit 1 is detected:

```c
svcb[0].type     = SV_VOLUMETBL | SV_RELCURPOS;
svcb[0].adjsize  = -1;
svcb[0].digit    = '1';
svcb[0].digtype  = DG_DTMF;
```

For additional examples of how to use the DX_SVCB structure, see the Example section for `dx_setsvcond()`.
**DX_SVMT — speed and volume modification tables**

```c
typedef struct DX_SVMT{
    char decrease[10];        /* Ten Downward Steps */
    char origin;              /* Regular Speed or Volume */
    char increase[10];        /* Ten Upward Steps */
} DX_SVMT;
```

**Description**

The DX_SVMT data structure contains parameters for the speed modification table and volume modification table. Speed adjustment is not supported on HMP.

You can specify the rate of change for volume adjustments by customizing the volume modification table (SVMT) per channel. The DX_SVMT structure has 21 entries that represent different levels of volume. This structure is used to set or retrieve the SVMT values, using `dx_setsvmt()` or `dx_getsvmt()` respectively.

For detailed information on volume modification tables, see the *Voice API Programming Guide*.

**Note:** Although there are 21 entries available in the DX_SVMT structure, all do not have to be utilized for changing volume; the number of entries can be as small as you require. Ensure that you insert -128 (80h) in any table entries that do not contain a volume setting.

**Field Descriptions**

The fields of the DX_SVMT data structure are described as follows:

- **decrease[10]**
  - Array that provides a maximum of 10 downward steps from the standard (normal) volume. The size of the steps is specified in this table. Specify the value -128 (80h) in any entry you are not using. This represents a null-entry and end-of-table marker. Valid values are:
    - Volume – Decibel decrease from the origin (which is set to 0). Values must be between -1 and -30.

- **origin**
  - Specifies the standard play volume. This is the original setting or starting point for volume control. Set the origin to 0 to assume normal playback volume for the standard (normal volume is -8 dB).

- **increase[10]**
  - Array that provides a maximum of 10 upward steps from the standard (normal) volume. The size of the steps is specified in this table. Specify the value -128 (80h) in any entry you are not using. This represents a null-entry and end-of-table marker. Valid values are:
    - Volume – Decibel decrease from the origin (which is set to 0). Values must be between 1 and 10.

If you use `dx_setsvmt()` to customize the DX_SVMT, the changes are saved permanently. You can obtain the manufacturer’s original defaults by specifying SV_SETDEFAULT for the `dx_setsvmt()` function.
Example

For an example of how to use the DX_SVMT structure, see the Example section for `dx_setsvmt()`.
DX_UIO — user-defined input/output

```
typedef struct DX_UIO {
    int (*u_read) ( );
    int (*u_write) ( );
    int (*u_seek) ( );
} DX_UIO;
```

**Description**

The DX_UIO data structure contains parameters for user-defined input/output.

This structure, returned by `dx_setuio()` , contains pointers to user-defined I/O functions for accessing non-standard storage devices.

**Field Descriptions**

The fields of the DX_UIO data structure are described as follows:

- **u_read**
  points to the user-defined `read()` function, which returns an integer equal to the number of bytes read or -1 for error

- **u_write**
  points to the user-defined `write()` function, which returns an integer equal to the number of bytes written or -1 for error

- **u_seek**
  points 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

**Example**

For an example of how to use the DX_UIO structure, see the Example section for `dx_setuio()`.
**DX_XPB**

```c
typedef struct {
    USHORT    wFileFormat;         // file format
    USHORT    wDataFormat;         // audio data format
    ULONG     nSamplesPerSec;      // sampling rate
    ULONG     wBitsPerSample;      // bits per sample
} DX_XPB;
```

### Description

The DX_XPB data structure contains parameters for the input/output transfer parameter block.

Use the I/O transfer parameter block (DX_XPB) data structure to specify the file format, data format, sampling rate, and resolution for certain play and record functions, such as `dx_playvox()`, `dx_recvox()`, `dx_playiottdata()`, `dx_reciottdata()`, and `dx_recwav()`.

The `dx_playwav()` convenience function does not specify a DX_XPB structure because the WAVE file header contains the necessary format information.

### Field Descriptions

The fields of the DX_XPB data structure are described as follows:

**wFileFormat**

Specifies the audio file format. Note that this field is ignored by the convenience functions `dx_recwav()`, `dx_recvox()`, and `dx_playvox()`.

- `FILE_FORMAT_VOX` – Dialogic VOX file format
- `FILE_FORMAT_WAVE` – Microsoft WAVE file format

**wDataFormat**

Specifies the data format.

Use one of the following data formats:

- `DATA_FORMAT_DIALOGIC_ADPCM` – 4-bit OKI ADPCM (Dialogic registered format)
- `DATA_FORMAT_MULAW` or `DATA_FORMAT_G711_MULAW` – 8-bit mu-law G.711 PCM
- `DATA_FORMAT_ALAW` or `DATA_FORMAT_G711_ALAW` – 8-bit A-law G.711 PCM
- `DATA_FORMAT_PCM` – 8-bit linear PCM
- `DATA_FORMAT_G726` – G.726 bit-exact coder

**nSamplesPerSec**

Specifies one of the following sampling rates:

- `DRT_6KHZ` – 6 kHz sampling rate
- `DRT_8KHZ` – 8 kHz sampling rate
- `DRT_11KHZ` – 11 kHz sampling rate. Note: 11 kHz OKI ADPCM is not supported.

**wBitsPerSample**

Specifies the number of bits per sample.
Examples

The following examples explain how to fill the DX_XPB structure for various voice coders.

Table 8. G.711 Voice Coder Support Fields

<table>
<thead>
<tr>
<th>DX_XPB Field</th>
<th>DX_XPB Field Value</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>wFileFormat</td>
<td>FILE_FORMAT_WAV or FILE_FORMAT_VOX</td>
<td></td>
</tr>
<tr>
<td>wDataFormat</td>
<td>DATA_FORMAT_G711_ALAW or</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DATA_FORMAT_ALAW</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DATA_FORMAT_G711_MULAW or</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DATA_FORMAT_MULAW</td>
<td></td>
</tr>
<tr>
<td>nSamplesPerSec</td>
<td>DRT_6KHZ or</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DRT_8KHZ</td>
<td></td>
</tr>
<tr>
<td>wBitsPerSample</td>
<td>8</td>
<td>48 or 64 kbps</td>
</tr>
</tbody>
</table>

Table 9. Linear PCM Voice Coder Support Fields

<table>
<thead>
<tr>
<th>DX_XPB Field</th>
<th>DX_XPB Field Value</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>wFileFormat</td>
<td>FILE_FORMAT_WAV or FILE_FORMAT_VOX</td>
<td></td>
</tr>
<tr>
<td>wDataFormat</td>
<td>DATA_FORMAT_PCM</td>
<td></td>
</tr>
<tr>
<td>nSamplesPerSec</td>
<td>DRT_11KHZ</td>
<td></td>
</tr>
<tr>
<td>wBitsPerSample</td>
<td>8</td>
<td>88 kbps</td>
</tr>
</tbody>
</table>

Table 10. OKI ADPCM Voice Coder Support Fields

<table>
<thead>
<tr>
<th>DX_XPB Field</th>
<th>DX_XPB Field Value</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>wFileFormat</td>
<td>FILE_FORMAT_WAV or FILE_FORMAT_VOX</td>
<td></td>
</tr>
<tr>
<td>wDataFormat</td>
<td>DATA_FORMAT_DIALOGIC_ADPCM</td>
<td></td>
</tr>
<tr>
<td>nSamplesPerSec</td>
<td>DRT_6KHZ or</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DRT_8KHZ</td>
<td></td>
</tr>
<tr>
<td>wBitsPerSample</td>
<td>4</td>
<td>24 or 32 kbps</td>
</tr>
</tbody>
</table>

Table 11. G.726 Voice Coder Support Fields

<table>
<thead>
<tr>
<th>DX_XPB Field</th>
<th>DX_XPB Field Value</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>wFileFormat</td>
<td>FILE_FORMAT_WAV or FILE_FORMAT_VOX</td>
<td></td>
</tr>
<tr>
<td>wDataFormat</td>
<td>DATA_FORMAT_G726</td>
<td></td>
</tr>
<tr>
<td>nSamplesPerSec</td>
<td>DRT_8KHZ</td>
<td></td>
</tr>
<tr>
<td>wBitsPerSample</td>
<td>2, 4</td>
<td>16, 32 kbps</td>
</tr>
</tbody>
</table>
typedef struct feature_table {
    unsigned short ft_play;
    unsigned short ft_record;
    unsigned short ft_tone;
    unsigned short ft_e2p_brd_cfg;
    unsigned short ft_fax;
    unsigned short ft_front_end;
    unsigned short ft_misc;
    unsigned short ft_send;
    unsigned short ft_receive;
    unsigned int   ft_play_ext;
    unsigned int   ft_record_ext;
    unsigned short ft_device;
    unsigned short ft_rfu[8];
} FEATURE_TABLE;

## Description

The FEATURE_TABLE data structure provides information about the features supported on a device. This structure is used by the `dx_getfeaturelist()` function. On return from the function, the FEATURE_TABLE structure contains the relevant information for the device.

Features reported by each member of the FEATURE_TABLE structure are defined in `dxxlib.h`. To determine what features are enabled on a device, “bitwise AND” the returned bitmask with the defines (see the example code for `dx_getfeaturelist()`).

## Field Descriptions

The fields of the FEATURE_TABLE data structure are described as follows:

- **ft_play**
  - Contains a bitmask of the play features supported on the specified device.
    - **FT_ADPCM** – supports ADPCM encoding
    - **FT_ALAW** – supports A-law encoding
    - **FT_DRT6KHZ** – supports 6 kHz sampling rate
    - **FT_DRT8KHZ** – supports 8 kHz sampling rate
    - **FT_DRT11KHZ** – supports 11 kHz sampling rate
    - **FT_ITU_G_726** – supports ITU-T G.726 encoding
    - **FT_LINEAR** – supports linear PCM encoding
    - **FT_PCM** – supports PCM encoding
    - **FT_RAW64KBIT** – supports raw 64 Kbps
    - **FT_RESRVD1** – reserved
    - **FT_RESRVD2** – reserved
    - **FT_ULAW** – supports mu-law encoding

- **ft_record**
  - Contains a bitmask of the record features supported on the specified device.
    - **FT_ADPCM** – supports ADPCM encoding
    - **FT_ALAW** – supports A-law encoding
    - **FT_DRT6KHZ** – supports 6 kHz sampling rate
    - **FT_DRT8KHZ** – supports 8 kHz sampling rate
    - **FT_DRT11KHZ** – supports 11 kHz sampling rate
<table>
<thead>
<tr>
<th>Feature Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FT_ITU_G_726</td>
<td>supports ITU-T G.726 encoding</td>
</tr>
<tr>
<td>FT_LINEAR</td>
<td>supports linear PCM encoding</td>
</tr>
<tr>
<td>FT_PCM</td>
<td>supports PCM encoding</td>
</tr>
<tr>
<td>FT_RAW64KBIT</td>
<td>supports raw 64 Kbps</td>
</tr>
<tr>
<td>FT_RESRVD1</td>
<td>reserved</td>
</tr>
<tr>
<td>FT_RESRVD2</td>
<td>reserved</td>
</tr>
<tr>
<td>FT_ULAW</td>
<td>supports mu-law encoding</td>
</tr>
</tbody>
</table>

**ft_tone**
Contains a bitmask of the tone features supported on the specified device.
- FT_GTDENABLED – supports global tone detection (GTD)
- FT_GTGENABLED – supports global tone generation (GTG)
- FT_CADENCE_TONE – supports cadenced tone generation

**ft_e2p_brd_cfg**
Contains a bitmask of the board configuration features supported on the specified device.
- FT_CONFERENCE – supports conferencing
- FT_CSP – supports continuous speech processing

**ft_fax**
Contains a bitmask of the board type and fax features supported on the specified device.
- FT_FAX – specifies that the device has a fax daughterboard
- FT_FAX_EXT_TBL – specifies send fax and receive fax feature support.
- FT_FAX_T38UDP – supports T.38 fax

**ft_front_end**
Not used on HMP. Contains a bitmask of the front-end features supported on the specified device.

**ft_misc**
Contains a bitmask of miscellaneous features supported on the specified device.
- FT_CSPEXTRASLOT – reserves extra transmit time slot for continuous speech processing
- FT_PROMPTEDREC – supports prompted record (triggered by VAD)
- FT_RECFLOWCONTROL – supports flow control on recording channels
- FT_VAD – supports voice activity detection

**ft_send**
Contains a bitmask of send fax features supported on the specified device.
- FT_SENDFAX_TXFILE_ASCII – indicates that ASCII file transfer is supported. If this bit is turned off and the FT_FAX_EXT_TBL bit (in ft_fax) is turned on, then the device supports DSP Fax (also known as Softfax).
- FT_TX14400 – supports fax transmission at 14.4 kbps
- FT_TXASCII – supports ASCII data fax transmission
- FT_TXFILEMR – supports MR encoded file format
- FT_TXFILEMMR – supports MMR encoded file format
- FT_TXLINEMR – supports MR encoded file format over the phone line
- FT_TXLINEMMR – supports MMR encoded file format over the phone line
- FT_TXECM – capable of fax line transmission with error correction mode
- FT_TXCCTFAX – supports the header “CCT FAX” when enabled in a download parameter file

**ft_receive**
Contains a bitmask of receive fax features supported on the specified device.
• FT_RX14400 – supports fax reception at 14.4 kbps
• FT_RX12000 – supports fax reception at 12 kbps
• FT_RXFILEMR – supports MR encoded file format
• FT_RXFILEMMR – supports MMR encoded file format
• FT_RXLINEMR – supports MR encoded file format over the phone line
• FT_RXLINEMMR – supports MMR encoded file format over the phone line
• FT_RXECM – capable of fax line reception with error correction mode

ft_play_ext
   Not used on HMP. Contains a bitmask of extended play features supported on the specified device.

ft_record_ext
   Not used on HMP. Contains a bitmask of extended record features supported on the specified device.

ft_device
   Reserved for future use.

ft_rfu
   Reserved for future use.

■ Example

See dx_getfeaturelist( ) for an example of how to use the FEATURE_TABLE structure.
SC_TSINFO — TDM bus time slot information

Description

The SC_TSINFO data structure contains the number of time division multiplexing (TDM) bus time slots associated with a particular device and a pointer to an array that holds the actual TDM bus time slot number(s). The SC_TSINFO structure is used by TDM bus routing functions identified by the suffix:

- _getxmitslot( ) to supply TDM bus time slot information about a device and fill the data structure
- _listen( ) to use this time slot information to connect two devices.

The prefix for these functions identifies the type of device, such as dx_ (voice) and fx_ (fax).

The TDM bus includes the CT Bus and SCbus. The CT Bus has 4096 bi-directional time slots, while the SCbus has 1024 bi-directional time slots. On HMP, no physical TDM bus exists but its functionality is implemented in the software.

This structure is defined in dxxxlib.h.

Field Descriptions

The fields of the SC_TSINFO structure are described as follows:

- **sc_numts**
  - initialized with the number of TDM bus time slots associated with a device, typically 1.

- **sc_tsarrayp**
  - initialized with a pointer to an array of long integers. The first element of this array contains a valid TDM bus time slot number which is obtained by issuing a call to a _getxmitslot( ) function.

Example

See dx_getxmitslot( ) for an example of how to use the SC_TSINFO structure.
TN_GEN

type 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 = 10 msec */
} TN_GEN;

### Description

The TN_GEN data structure contains parameters for the tone generation template.

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

### Field Descriptions

The fields of the TN_GEN data structure are described as follows:

- **tg_dflag**
  - Tone Generation Dual Tone Flag: Flag indicating single- or dual-tone definition. If single, the values in `tg_freq2` and `tg_ampl2` will be ignored.
  - TN_SINGLE – single tone
  - TN_DUAL – dual tone

- **tg_freq1**
  - specifies the frequency for tone 1 in Hz (range: 200 to 2000 Hz)

- **tg_freq2**
  - specifies the frequency for tone 2 in Hz (range: 200 to 2000 Hz)

- **tg_ampl1**
  - specifies the amplitude for tone 1 in dB (range: -40 to 0 dB)

- **tg_ampl2**
  - specifies the amplitude for tone 2 in dB (range: -40 to 0 dB)

- **tg_dur**
  - specifies the duration of the tone in 10 msec units; -1 = infinite duration

### Example

For an example of how to use the TN_GEN structure, see the Example section for `dx_bldtngen()`.
TN_GENCAD — cadenced tone generation template

TN_GENCAD

typedef struct {
  unsigned char cycles;      /* Number of cycles */
  unsigned char numsegs;     /* Number of tones */
  short       offtime[4];  /* Array of off-times */
  /* one for each tone */
  TN_GEN      tone[4];     /* Array of tone templates */
} TN_GENCAD;

Description

The TN_GENCAD data structure contains parameters for the cadenced tone generation template. It defines a cadenced tone that can be generated by using the `dx_playtoneEx()` function.

TN_GENCAD defines a signal by specifying the repeating elements of the signal (the cycle) and the number of desired repetitions. The cycle can contain up to 4 segments, each with its own tone definition and on/off duration, which creates the signal pattern or cadence. Each segment consists of a TN_GEN single- or dual-tone definition (frequency, amplitude, & duration) followed by a corresponding off-time (silence duration) that is optional. The `dx_bldtngen()` convenience function can be used to set up the TN_GEN components of the TN_GENCAD structure. The segments are seamlessly concatenated in ascending order to generate the signal cycle.

TN_GENCAD is defined in `dxxxlib.h`.

Field Descriptions

The fields of the TN_GENCAD data structure are described as follows:

cycles
  The cycles field specifies the number of times the cycle will be played.
  Valid values are 1 to 40 cycles.

numsegs
  The numsegs field specifies the number of segments used in the cycle, from 1 to 4. A segment consists of a tone definition in the tone[ ] array plus the corresponding off-time in the offtime[ ] array. If you specify less than four segments, any data values in the unused segments will be ignored (if you specify two segments, the data in segments 3 and 4 will be ignored). The segments are seamlessly concatenated in ascending order to generate the cycle.

offtime[4]
  The offtime[ ] array contains four elements, each specifying an off-time (silence duration) in 10 msec units that corresponds to a tone definition in the tone[ ] array. The offtime[ ] element is ignored if the segment is not specified in numsegs.
  The off-times are generated after the tone on-time (TN_GEN tg_dur), and the combination of tg_dur and offtime produce the cadence for the segment. Set the offtime = 0 to specify no off-time for the tone.
The tone[] array contains four elements that specify TN_GEN single- or dual-tone definitions (frequency, amplitude, & duration). The tone[] element is ignored if the segment is not specified in numsegs.

The dx_bldtngen() function can be used to set up the TN_GEN tone[] elements. At least one tone definition, tone[0], is required for each segment used, and you must specify a valid frequency (tg_freq1); otherwise an EDX_FREQGEN error is produced. See the TN_GEN structure for more information.

Example

For examples of TN_GENCAD, see the standard call progress signals used with the dx_playtoneEx() function.
**TONE_DATA**

typedef struct {
    unsigned int structver;       /* version of TONE_SEG struct */
    unsigned short tn_dflag;      /* Dual Tone - 1, Single Tone - 0 */
    unsigned short tn1_min;       /* Min. Frequency for Tone 1 (in Hz) */
    unsigned short tn1_max;       /* Max. Frequency for Tone 1 (in Hz) */
    unsigned short tn2_min;       /* Min. Frequency for Tone 2 (in Hz) */
    unsigned short tn2_max;       /* Max. Frequency for Tone 2 (in Hz) */
    unsigned short tn_twinmin;    /* Min. Frequency for twin of dual tone (in Hz) */
    unsigned short tn_twinmax;    /* Max. Frequency for twin of dual tone (in Hz) */
    unsigned short tnon_min;      /* Debounce Min. ON Time (in 10msec units) */
    unsigned short tnon_max;      /* Debounce Max. ON Time (in 10msec units) */
    unsigned short tnoff_min;     /* Debounce Min. OFF Time (in 10msec units) */
    unsigned short tnoff_max;     /* Debounce Max. OFF Time (in 10msec units) */
} TONE_SEG;

typedef struct {
    unsigned int structver;       /* version of TONE_DATA struct */
    unsigned short tn_rep_cnt;    /* Debounce Rep Count */
    unsigned int numofseg;        /* Number of segments for a MultiSegment Tone */
    TONE_SEG toneseg[6];         /* */
} TONE_DATA

### Description

The TONE_DATA data structure contains tone information for a specific call progress tone. This structure is used by the `dx_createtone()` function. This structure is defined in `dxxxlib.h`. For information on call progress analysis and default tone definitions, see the Voice API Programming Guide.

The TONE_DATA structure contains a nested array of TONE_SEG substructures. A maximum of six TONE_SEG substructures can be specified.

**Note:** Be sure to set all unused fields in the structure to 0 before using this structure in a function call. This action prevents possible corruption of data in the allocated memory space.

### Field Descriptions

The fields of the TONE_DATA structure are described as follows:

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

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

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

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

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

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

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

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

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

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

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

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

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

TONE_DATA.tn_rep_cnt
  Specifies the debounce repetition count.

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

Example

For an example of this structure, see the Example code for dx_createtone().
TONE_DATA — tone information
This chapter lists the error codes that may be returned for the voice library functions.

If a library function fails, use the standard attribute function ATDV_LASTERR() to return the error code and ATDV_ERRMSGP() to return the error description. These functions are described in the Standard Runtime Library API Library Reference.

The following errors can be returned using the ATDV_LASTERR() and ATDV_ERRMSGP() functions:

- **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**
  - Invalid parameter
- **EDX_BADPROD**
  - Function not supported on this board
- **EDX_BADREGVALUE**
  - Unable to locate value in registry
- **EDX_BADTPT**
  - DV_TPT structure error
- **EDX_BADTSFDATA**
  - Tone Set File (TSF) data was not consolidated
- **EDX_BADTSFFILE**
  - Filename doesn’t exist, or not valid TSF
- **EDX_BADWAVEFILE**
  - Bad/unsupported WAVE file
- **EDX_BUSY**
  - Device or channel is busy; or invalid state
- **EDX_CADENCE**
  - Invalid cadence component values in tone template description
- **EDX_CHANNUM**
  - Invalid channel number specified
**Error Codes**

EDX_DIGTYPE  
Invalid dg_type value in user digit buffer, DV_DIGIT data structure

EDX_FEATUREDISABLED  
Feature disabled

EDX_FLAGGEN  
Invalid tg_dflag field in tone generation template, TN_GEN data structure

EDX_FREQDET  
Invalid frequency component values in tone template description

EDX_FREQGEN  
Invalid frequency component in tone generation template, TN_GEN data structure

EDX_FWERROR  
Firmware error

EDX_IDLE  
Device is idle

EDX_INVSUBCMD  
Invalid sub-command number

EDX_MAXTMPLT  
Maximum number of user-defined tones for the board

EDX_MSGSTATUS  
Invalid message status setting

EDX_NOERROR  
No error

EDX_NONZEROSIZE  
Reset to default was requested but size was non-zero

EDX_NOSUPPORT  
Data format is not supported or function parameter is not supported

EDX_NOTENOUGHBRDMEM  
Error when downloading a cached prompt from multiple sources: total length of data to be downloaded exceeds the available on-board memory

EDX_NOTIMP  
Function is not implemented

EDX_SH_BADCMD  
Command is not supported in current bus configuration

EDX_SH_BADEXTTS  
TDM bus time slot is not supported at current clock rate

EDX_SH_BADINDX  
Invalid Switch Handler library index number

EDX_SH_BADCLTS  
Invalid channelts number

EDX_SH_BADMODE  
Function is not supported in current bus configuration
Error Codes

EDX_SH_BADTYPE
   Invalid time slot channel type (voice, analog, etc.)

EDX_SH_CMDBLOCK
   Blocking command is in progress

EDX_SH_LCLDSCNCT
   Channel is already disconnected from TDM bus

EDX_SH_LCLTSCNCT
   Channel is already connected to TDM bus

EDX_SH_LIBBSY
   Switch Handler library is busy

EDX_SH_LIBNOTINIT
   Switch Handler library is uninitialized

EDX_SH_MISSING
   Switch Handler is not present

EDX_SH_NOCLK
   Switch Handler clock fallback failed

EDX_SPDVOL
   Must specify SV_VOLUME_TBL

EDX_SVADJBLKS
   Invalid number of volume adjustment blocks

EDX_SVMRANGE
   Entry out of range in volume modification table, SV_SVMT

EDX_SVMTSIZE
   Invalid table size specified

EDX_SYSTEM
   Error from operating system. In Windows, use dx_fileerrno() to obtain error value. In Linux, check the global variable errno for more information.

EDX_TIMEOUT
   I/O function timed out

EDX_TONEID
   Invalid tone template ID

EDX_TNMSGSTATUS
   Invalid message status setting

EDX_UNSUPPORTED
   Function is not supported

EDX_XBPARM
   Bad XPB structure
This chapter provides reference information on the following topics:

- DTMF and MF Tone Specifications ........................................ 317
- DTMF and MF Detection Errors ........................................... 318

6.1 DTMF and MF Tone Specifications

Table 12 provides information on DTMF specifications. Table 13 provides information on MF tone specifications.

Table 12. DTMF Tone Specifications

<table>
<thead>
<tr>
<th>Code</th>
<th>Tone Pair Frequencies (Hz)</th>
<th>Default Length (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>697, 1209</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>697, 1336</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>697, 1477</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>770, 1209</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>770, 1336</td>
<td>100</td>
</tr>
<tr>
<td>6</td>
<td>770, 1477</td>
<td>100</td>
</tr>
<tr>
<td>7</td>
<td>852, 1209</td>
<td>100</td>
</tr>
<tr>
<td>8</td>
<td>852, 1336</td>
<td>100</td>
</tr>
<tr>
<td>9</td>
<td>852, 1477</td>
<td>100</td>
</tr>
<tr>
<td>0</td>
<td>941, 1336</td>
<td>100</td>
</tr>
<tr>
<td>*</td>
<td>941, 1209</td>
<td>100</td>
</tr>
<tr>
<td>#</td>
<td>941, 1477</td>
<td>100</td>
</tr>
<tr>
<td>a</td>
<td>697, 1633</td>
<td>100</td>
</tr>
<tr>
<td>b</td>
<td>770, 1633</td>
<td>100</td>
</tr>
<tr>
<td>c</td>
<td>852, 1633</td>
<td>100</td>
</tr>
<tr>
<td>d</td>
<td>941, 1633</td>
<td>100</td>
</tr>
</tbody>
</table>
6.2 DTMF and MF Detection Errors

Some MF digits use approximately the same frequencies as DTMF digits (see Table 12 and Table 13). 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. See the \texttt{dx_setdigtyp()} function description for information on setting the type of digit detection.

Digit detection accuracy depends on two things:

- the digit sent
- the kind of detection enabled when the digit is detected

Table 14 and Table 15 show the digits that are detected when each type of detection is enabled. Table 14 shows which digits are detected when MF digits are sent. Table 15 shows which digits are detected when DTMF digits are sent.
Table 14. Detecting MF Digits

<table>
<thead>
<tr>
<th>MF Digit Sent</th>
<th>String Received</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Only MF Detection Enabled</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
</tr>
</tbody>
</table>

* = detection error

Table 15. Detecting DTMF Digits

<table>
<thead>
<tr>
<th>DTMF Digit Sent</th>
<th>String Received</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Only DTMF Detection Enabled</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

* = detection error
Table 15. Detecting DTMF Digits (Continued)

<table>
<thead>
<tr>
<th>DTMF Digit Sent</th>
<th>String Received</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Only DTMF Detection Enabled</td>
</tr>
<tr>
<td></td>
<td>Only DTMF Detection Enabled</td>
</tr>
<tr>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>d</td>
<td>d</td>
</tr>
</tbody>
</table>

* = detection error
**Glossary**

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

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

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

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

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

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

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

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

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

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

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

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

**bus:** An electronic path that allows communication between multiple points or devices in a system.
**busy device:** A device that has one of the following characteristics: is stopped, being configured, has a multitasking or non-multitasking function active on it, or I/O function active on it.

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

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

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

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

**caller ID:** calling party identification information.

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

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

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

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

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

**configuration file:** An unformatted ASCII file that stores device initialization information for an application.
**convenience function:** A class of functions that simplify application writing, sometimes by calling other, lower-level API functions.

**CPE:** customer premise equipment.

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

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

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

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

**deglitching:** See debouncing.

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

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

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

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

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

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

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

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

**downloadable Springware firmware:** Software features loaded to Intel voice hardware. Features include voice recording and playback, enhanced voice coding, tone detection, tone generation, dialing, call progress analysis, voice detection, answering machine detection, speed control, volume control, ADSI support, automatic gain control, and silence detection.
**driver**: A software module which provides a defined interface between an application program and the firmware interface.

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

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

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

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

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

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

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

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

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

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

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

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

**I/O**: Input-Output

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

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

**kernel**: A set of programs in an operating system that implement the system’s functions.
**mu-law:** (1) Pulse Code Modulation (PCM) algorithm used in digitizing telephone audio signals in T1 areas. (2) The PCM coding and companding standard used in Japan and North America. See also **A-law.**

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

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

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

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

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

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

**RFU:** reserved for future use

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

**route:** Assign a resource to a time slot.

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

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

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

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

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

**solicited event:** An expected event. It is specified using one of the device library’s asynchronous functions.
**Springware:** Software algorithms built into the downloadable firmware that provide the voice processing features available on older-generation Intel® Dialogic® voice boards. The term Springware is also used to refer to a whole set of boards from Intel built using this architecture. Contrast with DM3, which is a newer-generation architecture.

**SRL:** See Standard Runtime Library.

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

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

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

**string:** An array of ASCII characters.

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

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

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

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

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

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

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

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

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

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

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

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

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

A

adjusting speed and volume
  explicitly 61
  using conditions 241
  using digits 241

adjustment conditions
  digits 242
  maximum number 242
  setting 241

ADPCM 169, 202

AGC 202

A-law 169, 301

array 291

asynchronous operation
  dialing 100
  digit collection 133
  playing 170
  playing tone 182
  recording 203
  stopping I/O functions 254

ATDX_ functions 21

ATDX_BDNAMEP( ) 24

ATDX_BDTYPE( ) 26

ATDX_BUFDIGS( ) 59

ATDX_CHNAMES( ) 28

ATDX_CHTYPE( ) 30

ATDX_CONNTYPE( ) 32

ATDX_CPERROR( ) 35

ATDX_CPTERM( ) 35, 38

ATDX_DEVTYPE( ) 41

ATDX_STATE( ) 43

ATDX_TERMSK( ) 45, 48

ATDX_TONEID( ) 48

ATDX_TRCOUNT( ) 51

automatic gain control 202

B

base memory address 291

bits per sample 301

board
  device 41, 163
  device name 24
  parameters 237, 238
    setting 24

board device
  handle 28

breaking
  connection to a time slot 257

buffer
  firmware digit 84

busy channel
  forcing to idle state 254

C

calling
  no answer 286
  pamd failtime 287
  pamd spdval 287

caching prompts
  playing 178

cadence
  repetition for user-defined tones 68

cadenced tone
  playing 186

call progress analysis 38
  data structure 286
  enabling 100
  errors 35
  functions 20
  parameter structure 82
  results
    busy 38
    called line answered by 38
    connect 38
    error 39
    no answer 38
    no ringback 38
    operator intercept 38
    stopped 38
    stopping 102, 255
    termination 38
    using dx_dial( ) 99

    call progress tone 91, 94, 199
call status transition
  DX_CST data structure 288
event block structure 289
event handling 231
  synchronously monitoring events 140
call status transition event functions 17
dx_getevt() 140
dx_setevtmsk() 230
call status transition structure 288
channel
  current state 43
device 41, 163
digit buffer 132
names 28
number 30
number of processes 141
parameters 238
status
dial 43
get digit 43
idle 43
play 43
playing tone 43
record 43
stopped 43
channel device information structure 276
channel parameters 239
clearing structures 82, 88
close(_) 78
close(_) function, Windows 78
closing devices 78
cnosig 286
CON_CAD 32
CON_PAMD 32
CON_PVD 32
configuration functions 14
dx_clrdigbuf() 84
dx_getparm() 147
dx_libinit() 158
dx_setdigtyp() 227
dx_setparm() 237
connect
type 32
convenience functions
dx_playf() 175
dx_playvox() 190
dx_recf() 208
dx_recvox() 216
dx_recwav() 219
I/O 15
R2/MF 19
TDM Routing 17

CR_BUSY 38
CR_CEPT 38
CR_CNCT 32, 38
CR_CNCT 32, 38
CR_ERROR 35
CR_FAXTONE 38
CR_LGTUERR 35
CR_MEMERR 35
CR_MXFRQERR 35
CR_NOANS 38
CR_NORB 38
CR_OVRLPERR 36
CR_STOPD 38
CR_TMOUTOFF 36
CR_TMOUTON 36
CR_UNEXPTN 36
CR_UPFRQERR 36
CS_CALL 43
CS_DIAL 43
CS_GTDIG 43
CS_HOOK 43
CS_IDLE 43
CS_PLAY 43
CS_RECORD 43
CS_STOPD 43
CS_TONE 43
cst_data 288
cst_event 288
CT_DEVINFO data structure 127, 276
current parameter settings 147
cycles 308

D
data formats 301
data structure
  user digit buffer 278
data structures
  cadenced tone generation template 308
  call progress analysis parameters 286
  call status transition 288
  clearing 21
  event block 289
  feature information 303
  I/O
    user-definable 300
  I/O transfer table 290
  input/output transfer parameter block 301
  speed and volume adjustment conditions 295
  speed modification table 298
  TDM bus time slot information 306
  termination parameter table 279
  tone generation template 307
DE_DIGITS event 273, 288, 289
DE_DIGOFF event 273
DE_SILOFF event 273, 288, 289
DE_SILOFF event 274, 288, 289
DE_STOPGETEV event 274
DE_TONEOFF event 274, 288, 289
DE_TONEON event 274, 288, 289
device
  opening 163
device handle 13, 26, 163
  freeing 78
device information structure 276
device management functions 13
  dx_close() 78
  dx_open() 163
device names
  displaying 28
device type 41
devices
  closing 78
  multiple processes 78
  returning features 303
type 26
DG_DTMF 278
DG_END 278
DG_MAXDIGS 133, 278
DG_MF 278
dg_type 278
dg_value 278
DI_D41BD 26
DI_D41CH 26
dialing
  ASCIIZ string 99
  asynchronous 100
  DTMF 101
  enabling call progress analysis 100
  flash 101
  MF 101
  pause 101
  pulse 101
  specifying dial string 99, 101
  stopping 101
  synchronous 100
  synchronous termination 100
termination events
  TDX_CALLP 100, 272
  TDX_DIAL 100, 272
with call progress analysis 100
digit buffer 132, 133
  flushing 84
digit buffer, user 278
digit collection 132
  asynchronous 133
  DTMF digits 132
  MF digits 132
  synchronous 133
  termination 133
  user-defined digits 132
digit detection 132
  disabling 105
  DTMF vs. MF tones 228
  errors 318
  multiple types 227
  setting digit types 227
digits
  adjustment conditions 242
  speed and volume 59
  disabling detection
  user-defined tones 105
disconnecting
  voice receive channel 257
DLL version number functions
  dx_GetDllVersion() 138
DM_DIGITS 231
DM_DIGOFF 231
DM_SILOF 231
DM_SILOF 231
DM_SILON 231
DM_UNDERRUN 231
DSP fax 304
DT_DXBD 41
DT_DXCH 41
DTMF 319
detection errors 318
tone specifications 317

DTMF digits
   collection 132
   overlap with MF digits 133

DV_DIGIT data structure 132, 278
   specifying 132

DV_TPT data structure 279
   clearing 88
   contiguous 88
   last entry in 88
   linked 88

dx_addtone( ) 53

dx_addvoldig( ) 58

dx_adjsv( ) 61

dx_blddt( ) 64

dx_blddtcad( ) 67

dx_bldst( ) 73

dx_bldstcad( ) 70

dx_bldning( ) 76

DX_CAP data structure 286
   clearing 82

dx_close( ) 78

dx_CloseStream() 80

dx_crlcap( ) 82

dx_chrdigbuf( ) 84, 133

dx_chrsvcond( ) 86, 241

dx_clrtpt( ) 88

dx_createitone( ) 90

DX_CST data structure 288

dx_deltones( ) 97

dx_dial( ) 45, 82, 102, 254

dx_distone( ) 53, 105

DX_EBLK data structure 140, 289

dx_enttone( ) 53, 108

dx_fileclose( ) 111

dx_fileerrno( ) 113

dx_filenopen( ) 116

dx_fileread( ) 118

dx_fileseek( ) 121

dx_filewrite( ) 124

dx_getctinfo( ) 127

dx_getcursv( ) 129

dx_getdigt( ) 84, 132, 278

dx_GetDIIVersion( ) 138

dx_getevt( ) 140, 232, 289

dx_getfeaturelist( ) 143

FEATURE_TABLE data structure 303

dx_getparm( ) 147, 169, 202, 237

dx_GetStreamInfo( ) 150

dx_getsvmt( ) 152

dx_getxmitslot( ) 155

DX_IOTT data structure 290

dx_listen( ) 160

dx_open( ) 163

dx_OpenStreamBuffer( ) 166

dx_play( ) 84, 168, 175, 291

dx_playf( ) 175

dx_playiootdata( ) 178

dx_playtone( ) 182

dx_playtoneEx( ) 186

dx_playwav( ) 193

dx_PlayStreamData( ) 196

dx_query( ) 198

dx_querytone( ) 94

dx_rec( ) 84, 201, 291

dx_recf( ) 208

dx_reciootdata( ) 212

dx_recoxx( ) 216

dx_recowav( ) 219

R_ResetStreamBuffer( ) 222

dx_setdevuo( ) 224

dx_setdigtyp( ) 132

dx_setevstmks( ) 140, 230

dx_setgdamp( ) 235

dx_setparm( ) 169, 202, 237

dx_setsvcond( ) 241

dx_setsvmt( ) 245

dx_setuo( ) 249

dx_SetWaterMark( ) 252

dx_StopCh( ) 101, 201, 254

DX_STREAMSTAT data structure 293

DX_SVCB data structure 241, 295

DX_SVMT data structure 245, 298

DX_UIO data structure 300
   used by dx_setdevuo( ) 224

dx_unlisten( ) 257

DX_XPB data structure 301

DXCH_PLAYDRATE 169

DXCH_RECRDRATE 202

dxxlib.h 237
E

enabling detection
user-defined tones 108
e enhanced call progress analysis 20
errors
call progress analysis 35
listing (voice library) 313
ev_data 289
ev_event 289
event
mask 231
event block structure 140
events 17
call status transition (CST) 273
categories 271
disabling 78
termination, list 271
extended attribute functions
ATDX_BDNAMEP( ) 24
ATDX_BDTYPE( ) 26
ATDX_CHNAMES( ) 28
ATDX_CHNUM( ) 30
ATDX_CONNTYPE( ) 32
ATDX_CPERROR( ) 35
ATDX_CPTERM( ) 38
ATDX_DEVTYPE( ) 41
ATDX_STATE( ) 43
ATDX_TERMMSK( ) 45
ATDX_TONEID( ) 48
ATDX_TRCOUNT( ) 51
extended attribute functions category 21

F

feature information data structure 304
FEATURE_TABLE data structure 303
file format 301
file manipulation functions 20
dx_fileclose( ) 111
dx_fileerrno( ) 113
dx_filenopen( ) 116
dx_fileread( ) 118
dx_fileseek( ) 121
dx_filewrite( ) 124
firmware digit buffer 84
fixed length string 148
flushing digit buffer 84
functions
ATDX_ 21
call progress analysis 20
call status transition Event 17
configuration 14
device management 13
extended attribute 21
global tone detection 18
global tone generation 19
I/O 14
I/O convenience 15
speed and volume 19
structure clearance 21
TDM routing 17
Windows
close( ) 78
G

G.711 PCM voice coder 301
G.726 voice coder 301
global tone detection
adding a tone 53
deleting tones 97
disabling 105
dual frequency cadence tones 67
dual frequency tones 64
enabling 108
enabling detection 53
functions 18
dx_addtone( ) 53
dx_blddt( ) 64
dx_blddtcad( ) 67
dx_bldst( ) 73
dx_bldstcad( ) 70
dx_deltone( ) 97
dx_distone( ) 105
dx_enbtone( ) 108
dx_setgtdamp( ) 235
removing tones 97
single frequency cadence tones 70
single frequency tones 73
global tone generation
functions 19
dx_bldtngen( ) 76
dx_playtone( ) 182
dx_playtoneEx( ) 186
playing a cadenced tone 186
playing a tone 182
template 307
GTD Frequency Amplitude
setting 235
H
hook state 78

I
I/O
  function 45
  transfer parameter block structure 301
  transfer table 290
  user-defined structure for 300
I/O convenience functions 15
I/O functions 14
  dx_dial() 99
  dx_getdig() 132
  dx_play() 168
  dx_playoutdata() 178
  dx_rec() 201
  dx_recvdata() 212
  dx_stopch() 254
intflg 286
io_bufp 291
IO_CONT 88, 290
IO_DEV 290
IO_EOT 88, 290
io_fhandle 291
io_length 291
IO_LINK 88, 290
IO_MEM 290
io_nextp 291
io_offset 291
io_prevp 291
IO_STREAM 290
io_type 290
IO_UIO 290
IO_USEOFFSET 291

L
leading edge notification
  user-defined tones 64
learn mode functions 90, 94, 198
line status 43

M
MD_ADPCM 169, 202
MD_GAIN 202
MD_NOGAIN 202
MD_PCM 169, 202

MF
detection 319
detection errors 318
digits
  collection 132
  support 227
tone specifications 317
MF digits
  overlap with DTMF digits 133
monitor channels 140
monitoring events 140
mu-law 301

N
names
  board device 24
non-standard I/O devices
  dx_setdevuio() 224
  dx_setuio() 249
numsegs 308

O
offset 291
offtime 308
OKI ADPCM voice coder 301
open() function 163
opening devices 163

P
parameter settings
  getting current 147
parameters
  board and channel 237, 238, 239
call progress analysis 82
  sizes 148
play
asynchronous 170
convenience function 175
default algorithm 169
default rate 169
mode 169
specifying mode 169
specifying number of bytes 291
synchronous 170
termination 170
   TDX_PLAY 170
termination events 170
tone
   asynchronous 182
   asynchronous termination events 182
   synchronous operation 183
transmitting tone before 169
voice data 190
play and record functions
dx_play( ) 168
dx_playf( ) 175
dx_playvox( ) 190
dx_rec( ) 201
dx_recf( ) 208
dx_recioctdata( ) 212
dx_recvox( ) 216
dx_recwav( ) 219
playback
   bytes transferred 51
playing
   see play 170
playing voice data 178
PM_BYTE 148
PM_FLSTR 148
PM_INT 148
PM_LONG 148
PM_SHORT 148
PM_SR6 169
PM_SR8 169
PM_TONE 169
PM_VLSTR 148
positive answering machine detection 32
positive voice detection 32
processes per channel 141
Pulse Code Modulation 169, 202

R
recording
   algorithm 202
   asynchronous 203
   asynchronous termination event
   TDX_RECORD 203
   bytes transferred 51
   convenience function 208
default algorithm 202
default gain setting 202
default sampling rate 202
gain control 202
mode 202
sampling rate 202
specifying mode 202
specifying number of bytes 291
stopping 201
synchronous 203
synchronous termination 204
voice data 201, 212, 216
WAVE data 219
with A-law 202
with tone 202
RM_ALAW 202
RM_SR6 202
RM_SR8 202
RM_TONE 202
routing functions
dx_getctinfo( ) 127
dx_getxmitslot( ) 155
dx_listen( ) 160
dx_unlisten( ) 257

S
sampling rates 301
SC_TSINFO data structure 306
sctools.c 259, 265
Softfax 304
speed
   adjustment conditions 241
   explicitly adjusting 61
   retrieving current 129
speed and volume
   current 62
   data structure 295
   last modified 62
   modification table
      setting 298
      resetting to origin 62
speed and volume convenience functions
dx_addvoldig( ) 58
speed and volume function
dx_setsvmt( ) 245
speed and volume functions 19
dx_adjsv( ) 61
dx_clrsvcond( ) 86
dx_getcursv( ) 129
dx_getsvmt( ) 152
dx_setsvcond( ) 241
speed and volume modification table
resetting to defaults 245, 246
retrieving contents 152
specifying speed 245
specifying volume 245
updating 245
speed control 298
sr_getevtdatap( ) 232
stop I/O functions
dial 254
termination reason
TM_USRSTOP 254
stopping call progress analysis 255
stopping I/O functions
synchronous 254
streaming to board
creating stream buffer 166
deleting stream buffer 80
DX_STREAMSTAT data structure 293
function summary 16
getting status info 150
putting data in buffer 196
resetting internal data 222
setting water mark 252
structure clearance functions 21
dx_clrcap( ) 82
dx_clrtpt( ) 88
structures
clearing 82, 88
digit buffer 132
DV_DIGIT 132
DX_CAP 82
DX_EBLK 140
DX_IOTT 168
event block 140
SV_ABSPOS 62
SV_CURLASTMOD 62
SV_CURORIGIN 62
SV_RELCURPOS 62
SV_RESETCURPOS 62
SV_TOGGLE 62
SV_TOGORIGIN 62
SV_VOLUMETBL 61
synchronous operation
dial 100
digit collection 133
play 170
playing tone 183
record 203
stopping I/O functions 254, 255

T

TDM bus
time slot information structure 306
TDM bus routing functions 17
dx_getctinfo( ) 127
dx_getxmitslot( ) 155
dx_listen( ) 160
dx_unlisten( ) 257
TDX_CALLP event 100, 272
TDX_CREATETONE event 272
TDX_CREATETONE_FAIL event 272
TDX_CST event 272
TDX_DELETETONE event 272
TDX_DELETETONE_FAIL 272
TDX_DIAL event 100, 272
TDX_ERROR event 272
TDX_GETDIG event 272
TDX_HIGHWATER event 273
TDX_LOWWATER event 273
TDX_NOSTOP event 272
TDX_PLAY event 170, 272
TDX_PLAYTONE event 182, 187, 272
TDX_QUERYTONE event 272
TDX_QUERYTONE_FAIL 272
TDX_RECORD event 203, 272
TDX_UNDERRUN event 273
termination
call progress analysis 38
stop I/O function 254
synchronous record 204
termination conditions 15
termination events 271
termination parameter table structure 279
terminations
- asynchronous play 170
- ATDX_TERMMSK() 45
- end of data 45
- function stopped 46
- I/O device error 45
- I/O function 45
- I/O functions 254
- inter-digit delay 45
- maximum DTMF count 45
- maximum function time 45
- maximum period of silence 45
- normal termination 45
- specific digit received 45
- synchronous play 170
- tone-on/off event 45

user-defined functions
- installing 224, 249

user-defined input/output data structure 300

user-defined tone ID 48

user-defined tones 53
- cadence repetition 68
- disabling detection 105
- dual frequency 64
- dual frequency cadence 67
- enabling detection 108
- first frequency 64
- first frequency deviation 64
- ID 64
- leading or trailing edge notification 64
- playing 186
- also see playing tone 182
- removing 97
- second frequency 64
- second frequency deviation 64
- single frequency 73
- single frequency cadence 70

V
- variable length string 148

version number
- product 138

volume
- adjusting 58
- adjustment conditions 241
- explicitly adjusting 61
- retrieving current 129

volume control 298

W
- water mark 252

WAVE files
- playing 193

Windows functions
- close(_) 78