# Contents

**Revision History**  
10

**About This Publication**  
13  
- Purpose  
- Intended Audience  
- How to Use This Publication  
- Related Information

### 1 Function Summary by Category

1.1 Device Management Functions  
1.2 Configuration Functions  
1.3 I/O Functions  
1.4 I/O Convenience Functions  
1.5 Streaming to Board Functions  
1.6 Analog Display Services Interface (ADSI) Functions  
1.7 Audio Input Functions  
1.8 Transaction Record Functions  
1.9 Cached Prompt Management Functions  
1.10 Call Status Transition (CST) Event Functions  
1.11 TDM Routing Functions  
1.12 Global Tone Detection (GTD) Functions  
1.13 Global Tone Generation (GTG) Functions  
1.14 R2/MF Convenience Functions  
1.15 Speed and Volume Functions  
1.16 Call Progress Analysis Functions  
1.17 Caller ID Functions  
1.18 File Manipulation Functions  
1.19 Echo Cancellation Resource Functions  
1.20 Structure Clearance Functions  
1.21 Syntellect License Automated Attendant Functions  
1.22 Extended Attribute Functions  
1.23 Voice Function Support by Platform

### 2 Function Information

2.1 Function Syntax Conventions  
2.2 Function Syntax Conventions

- ag_getctinfo( ) – get information about an analog device  
- ag_getxmitslot( ) – get TDM bus time slot number of analog transmit channel  
- ag_listen( ) – connect analog receive channel to TDM bus time slot  
- ag_unlisten( ) – disconnect analog receive channel from TDM bus  
- ai_close( ) – close an audio input device  
- ai_getxmitslot( ) – get TDM bus time slot number of audio input transmit channel  
- ai_open( ) – open an audio input device  
- ATDX_ANSRSIZ( ) – return the duration of the answer  
- ATDX_BDNAMEP( ) – return a pointer to the board device name  
- ATDX_BDTYPE( ) – return the board type for the device

---

3
Contents

ATDX_BUFDIGS() – return the number of uncollected digits ........................................... 58
ATDX_CHNAMES() – retrieve all channel names for a board .............................................. 60
ATDX_CHNUM() – return the channel number ................................................................. 62
ATDX_CONNTYPE() – return the connection type for a completed call ............................... 64
ATDX_CPERROR() – return the call progress analysis error ............................................... 67
ATDX_CPTERM() – return the last result of call progress analysis termination ....................... 70
ATDX_CRTNID() – return the last call progress analysis termination ................................. 73
ATDX_DEVTYPE() – return the device type ................................................................. 76
ATDX_DTNFAIL() – return character for dial tone ............................................................. 78
ATDX_FRQDUR() – return the duration of the first SIT sequence ..................................... 81
ATDX_FRQDUR2() – return the duration of the second SIT sequence ................................. 83
ATDX_FRQDUR3() – return the duration of the third SIT sequence ..................................... 85
ATDX_FRQHZ() – return the frequency of the first SIT sequence ....................................... 87
ATDX_FRQHZ2() – return the frequency of the second SIT sequence ................................. 89
ATDX_FRQHZ3() – return the frequency of the third SIT sequence ..................................... 91
ATDX_FRQOUT() – return percentage of time SIT tone was out of bounds ......................... 93
ATDX_FWVER() – return the voice firmware version number ............................................ 95
ATDX_HOOKST() – return the current hook-switch state .................................................. 97
ATDX_LINEST() – return the current activity on the channel ........................................... 99
ATDX_LONGLOW() – return duration of longer silence detected ....................................... 101
ATDX_PHYADDR() – return the physical board address .................................................... 103
ATDX_SHORTLOW() – return duration of shorter silence detected ..................................... 105
ATDX_SIZEH() – return duration of initial non-silence .................................................. 107
ATDX_STATE() – return the current state of the channel ............................................... 109
ATDX_TERMMSK() – return the reason for the last I/O function termination ....................... 111
ATDX_TONEID() – return user-defined tone ID that terminated I/O function ....................... 115
ATDX_TRCOUNT() – return the byte count for the last I/O transfer ................................ 118
dx_addspddig() – set a DTMF digit to adjust speed ....................................................... 120
dx_addtone() – add a user-defined tone ........................................................................ 124
dx_advdig() – set a DTMF digit to adjust volume .............................................................. 129
dx_adjsv() – adjust speed or volume immediately ............................................................. 133
dx_bldthd() – define a user-defined dual-frequency tone ................................................. 136
dx_bldthcad() – define a user-defined dual-frequency cadenced tone ............................... 139
dx_bldst() – define a user-defined single-frequency tone ................................................ 142
dx_bldstcad() – define a user-defined single-frequency cadenced tone ............................ 145
dx_bldtngen() – define a tone for generation .................................................................. 148
dx_cacheprompt() – download a cached prompt from multiple sources ......................... 151
dx_chgudur() – change the duration definition for a tone ................................................. 154
dx_chgfreq() – change the frequency definition for a tone .............................................. 158
dx_chgprecnt() – change the repetition definition for a tone ........................................... 162
dx_close() – close a channel or board device handle ...................................................... 166
dx_CloseStreamBuffer() – delete a circular stream buffer ............................................. 169
dx_clrcap() – clear all fields in a DX_CAP structure ...................................................... 171
dx_clrdigbuf() – clear all digits in the firmware digit buffer .......................................... 173
dx_clrsvcond() – clear all speed or volume adjustment conditions ................................ 175
dx_clrtpt() – clear all fields in a DV_TPT structure ....................................................... 177
dx_createtone() – create a new tone definition for a specific call progress tone .................. 179
dx_deletetone() – delete a specific call progress tone .................................................. 183
dx_deltones() – delete all user-defined tones ................................................................. 186
dx_dial() – dial an ASCIIZ string ...................................................................................... 188
dx_distone() – disable detection of a user-defined tone ...................................................... 197
dx_enbtone() – enable detection of a user-defined tone ....................................................... 200
dx_fileclose() – close a file ................................................................................................ 203
dx_fileerrno() – return the system error value ................................................................... 205
 dx_fileopen() – open a file .................................................................................................. 208
dx_fileread() – read data from a file .................................................................................... 210
 dx_fileseek() – move a file pointer ......................................................................................... 213
dx_filewrite() – write data from a buffer into a file .............................................................. 216
dx_getcachesize() – get size of on-board memory for cached prompts ................................. 219
dx_getcinfo() – get information about a voice device .......................................................... 221
 dx_getcursv() – return the specified current speed and volume settings ............................... 223
 dx_getdig() – collect digits from a channel digit buffer ........................................................... 226
dx_GetDllVersion() – retrieve the voice DLL version number ............................................ 232
 dx_getevent() – monitor channel events synchronously ....................................................... 234
dx_getfeaturelist() – retrieve feature support information for the device ............................. 237
 dx_getparm() – get the current parameter settings ................................................................. 242
 dx_GetRscStatus() – return assignment status of a shared resource ..................................... 245
dx_GetStreamInfo() – retrieve information about the circular stream buffer ......................... 247
 dx_getsvmt() – return the current speed or volume modification table ................................. 249
 dx_getxmitslotecr() – get echo cancellation resource transmit time slot number ................. 255
 dx_getxmitslot() – get TDM bus time slot number of a voice transmit channel ................ 252
dx_gtextcallid() – retrieve a caller ID message .................................................................... 262
dx_gtsernum() – retrieve the board serial number ............................................................... 270
dx_initcallp() – initialize and activate call progress analysis ............................................. 272
 dx_libinit() – initialize the voice library DLL ................................................................. 275
dx_listen() – connect a voice listen channel to TDM bus time slot ....................................... 277
 dx_listencr() – enable echo cancellation resource mode ...................................................... 280
 dx_listencrex() – modify characteristics of the echo canceller ............................................ 283
 dx_mreciottdata() – record voice data from two TDM bus time slots ................................... 286
 dx_open() – open a voice device and return a unique device handle ..................................... 292
 dx_OpenStreamBuffer() – create and initialize a circular stream buffer .............................. 295
 dx_pause() – pause on-going play .......................................................................................... 297
 dx_play() – play recorded voice data .................................................................................... 299
 dx_playf() – synchronously play voice data ......................................................................... 307
 dx_playiottdata() – play back recorded voice data from multiple sources ......................... 310
 dx_playtone() – play tone defined by TN_GEN structure ...................................................... 315
 dx_playtoneEx() – play the cadenced tone defined by TN_GENCAD ................................... 319
 dx_playvox() – play voice data stored in a single VOX file ................................................. 324
 dx_playwav() – play voice data stored in a single WAVE file ............................................... 327
 dx_PutStreamData() – place data into a circular stream buffer ........................................... 330
 dx_querytone() – get tone information for a specific call progress tone ............................... 333
dx_rec() — record voice data to a single channel .......................... 336
dx_rec() — record voice data to a single file ................................. 343
dx_reciottdata() — record voice data to multiple destinations .......... 347
dx_recwav() — record voice data to a single WAVE file .................... 354
dx_ResetStreamBuffer() — reset internal data for a circular stream buffer 357
dx_resume() — resume paused play ............................................ 359
dx_RxIottData() — receive data on a specified channel ................. 361
dx_sendevt() — allow inter-process event communication ............... 365
dx_setchxfercnt() — set the bulk queue buffer size ...................... 367
dx_setdevuio() — install and retrieve user-defined I/O functions .... 369
dx_setdigit() — set the digit buffering mode .............................. 372
dx_setdigitf( ) — control the types of digits detected by the voice channel 374
dx_setevmsk() — enable detection of call status transition (CST) events 377
dx_setgtdamp() — set up the tone detection amplitudes ................ 382
dx_sethook() — provide control of the hook switch status ............... 384
dx_setparm() — set physical parameters of a channel or board device 388
dx_SetRecordNotifyBeepTone() — specify the template of the cadenced tone 399
dx_setsvcond() — set conditions that adjust speed or volume of play 401
dx_setsvmt() — change default values of the speed or volume modification table 405
dx_settonelen() — change the duration of the built-in beep tone .......... 409
dx_setuio() — install user-defined I/O functions .......................... 412
dx_SetWaterMark() — set water mark for the circular stream buffer 415
dx_stopch() — force termination of currently active I/O functions .... 417
dx_TSFStatus() — return the status of tone set file loading ............ 421
dx_TxIottData() — transmit data on a specified channel ............... 423
dx_TxRxIottData() — start a transmit-initiated reception of data .... 426
dx_unlisten() — disconnect voice receive channel from TDM bus ....... 431
ndx_unlistenercr() — disable echo cancellation resource (ECR) mode 433
dx_wink() — generate an outbound wink .................................... 436
dx_wtcallid() — wait for rings and report caller ID ...................... 440
dx_wtring() — wait for a specified number of rings ....................... 443
li_attendant() — perform the actions of an automated attendant ........ 446
li_ilslicened_syntelletc() — verify Syntelletc patent license on board 450
nr_scroute() — make a full or half-duplex connection .................... 451
nr_scunroute() — break a full or half-duplex connection ............... 457
r2_creatfsig() — create R2/MF forward signal tone ....................... 462
r2_playbsig() — play R2/MF backward signal tone ....................... 466

3 Events ................................................................. 471
3.1 Overview of Events .................................................. 471
3.2 Termination Events .................................................. 471
3.3 Unsolicited Events ................................................... 473
3.4 Call Status Transition (CST) Events ............................... 473

4 Data Structures ..................................................... 477
ADSI_XFERSTRUC — ADSI 2-way FSK data transfer buffer .............. 478
## Contents

CT_DEVINFO – channel/time slot device information .................................................. 479
DV_DIGIT – user digit buffer ....................................................................................... 483
DV_TPT – termination parameter table ........................................................................ 485
DX_ATTENDANT – Syntellect License Automated Attendant ......................................... 493
DX_CAP – call progress analysis parameters ............................................................... 496
DX_CST – call status transition (CST) information ...................................................... 504
DX_EBLK – call status transition event block ............................................................... 506
DX_ECRCT – echo cancellation resource (ECR) characteristics ..................................... 508
DX_IOTT – input/output transfer table ........................................................................ 509
DX_STREAMSTAT – status of stream buffer ................................................................. 512
DX_SVCB – speed and volume adjustment condition block .......................................... 514
DX_SVMT – speed and volume modification tables ...................................................... 517
DX_UIO – user-defined input/output ............................................................................ 519
DX_XPB – input/output transfer parameter block ....................................................... 520
FEATURE_TABLE – feature information ..................................................................... 525
SC_TSINFO – TDM bus time slot information ............................................................... 529
TN_GEN – tone generation template ............................................................................ 530
TN_GENCAD – cadenced tone generation template ....................................................... 531
TONE_DATA – tone information .................................................................................. 533

5  Error Codes ................................................................................................................. 535

6  Supplementary Reference Information ....................................................................... 539
6.1  DTMF and MF Tone Specifications ........................................................................ 539
6.2  DTMF and MF Detection Errors ............................................................................ 540

Glossary ......................................................................................................................... 543

Index .............................................................................................................................. 551
Contents

Figures

1  Format of General Caller ID Information ................................................. 264
Contents

Tables

1 Voice Function Support by Platform .......................................................... 29
2 Valid Dial String Characters (DM3) .......................................................... 189
3 Valid Dial String Characters (Springware) ............................................... 190
4 System Error Values .................................................................................. 205
5 Caller ID Common Message Types ............................................................. 264
6 Caller ID CLASS Message Types (Multiple Data Message) ......................... 265
7 Caller ID ACLIP Message Types (Multiple Data Message) ......................... 265
8 Caller ID CLIP Message Types .................................................................. 266
9 Caller ID JCLIP Message Types (Multiple Data Message) ......................... 266
10 Play Mode Selections ................................................................................. 301
11 Record Mode Selections ........................................................................... 338
12 Voice Board Parameters (DM3) ................................................................. 390
13 Voice Board Parameters (Springware) ....................................................... 390
14 Voice Channel Parameters (DM3) ............................................................. 391
15 Voice Channel Parameters (Springware) ................................................... 394
16 DV_TPT Field Settings Summary .............................................................. 490
17 G.711 Voice Coder Support Fields (DM3) ................................................. 521
18 G.721 Voice Coder Support Fields (DM3) ................................................. 522
19 Linear PCM Voice Coder Support Fields (DM3) ........................................ 522
20 OKI ADPCM Voice Coder Support Fields (DM3) ..................................... 522
21 G.726 Voice Coder Support Fields (DM3) ................................................. 522
22 GSM Voice Coder Support Fields (DM3) .................................................. 523
23 TrueSpeech Voice Coder Support Fields (DM3) ........................................ 523
24 IMA ADPCM Voice Coder Support Fields (DM3) ..................................... 523
25 G.726 Voice Coder Support Fields (Springware) ...................................... 523
26 GSM Voice Coder Support Fields (Springware) ........................................ 524
27 DTMF Tone Specifications ....................................................................... 539
28 MF Tone Specifications (CCITT R1 Tone Plan) ........................................ 540
29 Detecting MF Digits ................................................................................ 541
30 Detecting DTMF Digits ........................................................................... 541
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>
</table>
## Revision History

<table>
<thead>
<tr>
<th>Document No.</th>
<th>Publication Date</th>
<th>Description of Revisions</th>
</tr>
</thead>
</table>
About This Publication

The following topics provide information about this publication:

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

Purpose

This publication provides a reference to all voice functions, parameters and data structures in the voice API, also called the R4 voice API, supported on Windows* operating systems. It is a companion document to the Voice API Programming Guide, the Standard Runtime Library API Programming Guide, and the Standard Runtime Library API Library Reference.

Intended Audience

This information is intended for:

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

How to Use This Publication

This document assumes that you are familiar with and have prior experience with Windows* operating systems and the C programming language. Use this document together with the following: the Voice API Programming Guide, the Standard Runtime Library API Programming Guide, and the Standard Runtime Library API Library Reference.

The information in this guide is organized as follows:

- Chapter 1, “Function Summary by Category” introduces the various categories of voice functions and provides a brief description of each function.
- Chapter 2, “Function Information” provides an alphabetical reference to all voice functions.
About This Publication

- Chapter 3, “Events” provides an alphabetical reference to events that may be returned by the voice software.
- Chapter 4, “Data Structures” provides an alphabetical reference to all voice data structures.
- Chapter 5, “Error Codes” presents a listing of error codes that may be returned by the voice software.
- Chapter 6, “Supplementary Reference Information” provides additional reference information on topics such as DTMF Tone Specifications, and MF Tone Specifications.

A glossary and index are provided for your reference.

Related Information

See the following for more information:

- For information about voice library features and guidelines for building applications using voice software, see the Voice API Programming Guide.
- For details on the Standard Runtime Library, supported programming models, and programming guidelines for building all applications, see the 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 library, see the Standard Runtime Library API Library Reference.
- For information on the system release, system requirements, software and hardware features, supported hardware, and release documentation, see the Release Guide for the system release you are using.
- For details on compatibility issues, restrictions and limitations, 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 system 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/releases/index.html.
- For details on installing the system software, see the System Release Installation Guide.
- 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 Programming Guide.
- For details on all functions and data structures in the Global Call library, see the Global Call API Library Reference.
- For details on configuration files (including FCD/PCD files) and instructions for configuring products, see the Configuration Guide for your product or product family.
- For technical support, see http://developer.intel.com/design/telecom/support/. This website contains developer support information, downloads, release documentation, technical notes, application notes, a user discussion forum, and more.
- For product and services information, see http://www.intel.com/network/csp/.
Function Summary by Category

This chapter describes the categories into which the voice library functions can be logically grouped. This chapter also includes a table listing function support on various platforms (DM3, Springware) as well as synchronous/asynchronous support.

- Device Management Functions ................................................................. 15
- Configuration Functions ........................................................................ 16
- I/O Functions ......................................................................................... 17
- I/O Convenience Functions .................................................................... 19
- Streaming to Board Functions ................................................................. 19
- Analog Display Services Interface (ADSI) Functions .............................. 20
- Audio Input Functions ........................................................................... 20
- Transaction Record Functions ................................................................. 20
- Cached Prompt Management Functions .................................................... 21
- Call Status Transition (CST) Event Functions .......................................... 21
- TDM Routing Functions ......................................................................... 21
- Global Tone Detection (GTD) Functions .................................................. 23
- Global Tone Generation (GTG) Functions ............................................... 23
- R2/MF Convenience Functions ............................................................... 24
- Speed and Volume Functions .................................................................. 24
- Call Progress Analysis Functions ............................................................ 25
- Caller ID Functions ................................................................................ 26
- File Manipulation Functions .................................................................. 26
- Echo Cancellation Resource Functions .................................................... 26
- Structure Clearance Functions ................................................................. 27
- Syntellect License Automated Attendant Functions .................................. 27
- Extended Attribute Functions ................................................................ 27
- Voice Function Support by Platform ....................................................... 29

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.

A set of device management functions exists for each Intel® Dialogic® library, such as fax (fx_ functions), modular station interface (ms_ functions), and conferencing (dcb_ functions). See the appropriate API Library Reference for more information on these functions.

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, particularly on DM3 boards, 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 physical boards, a list of all virtual boards on a physical board, and a list of all subdevices on a virtual board. 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.

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()`
  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
Function Summary by Category

dx_GetRscStatus( )
returns the assignment status of a shared resource for the specified channel

dx_gtsernum( )
returns the board serial number

dx_libinit( )
initializes the voice dynamic link library (DLL)

dx_setchxfercnt( )
sets the bulk queue buffer size for the channel

dx_setdigbuf( )
sets the digit buffering mode

dx_setdigtyp( )
controls the types of digits detected by the device

dx_sethook( )
sets the hook switch state

dx_setparm( )
sets physical parameters for the device

dx_SetRecordNotifyBeepTone( )
specifies the template of the cadenced tone for record notification beep tone

dx_settonelen( )
changes the duration of the built-in beep tone (pre-record beep)

dx_TSFStatus( )
returns the status of tone set file loading

dx_wtring( )
waits for a specified number of rings

Note: The dx_sethook( ) and dx_setdigbuf( ) functions can also be classified as an I/O function and all I/O characteristics apply.

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 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( ) and dx_wink( ). These conditions dictate what events will cause an I/O function to terminate. The
termination conditions are specified just before the I/O function call is made. Obtain termination reasons for I/O functions by calling the extended attribute function ATDX_TERMMSK(). See the Voice API Programming Guide for information about I/O terminations.

**Note:** To send and receive FSK data from an Analog Display Services Interface (ADSI) device, see Section 1.6, “Analog Display Services Interface (ADSI) Functions”, on page 20.

The I/O functions are:

- **dx_dial()**
  dials an ASCII string of digits

- **dx_getdig()**
  collects digits from a channel digit buffer (for up to 31 digits plus the null terminator)

- **dx_pause()**
  pauses on-going play

- **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_recioxtdata()**
  records voice data to any combination of data files, memory, or custom devices, and lets the user specify format information

- **dx_resume()**
  resumes paused play

- **dx_setdevuio()**
  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

- **dx_wink()**
  generates an outbound wink

**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
Function Summary by Category

dx_PutStreamData( )
places data into the circular stream buffer

dx_ResetStreamBuffer( )
resets internal data for a circular stream buffer

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

1.6 Analog Display Services Interface (ADSI) Functions

The send and receive frequency shift keying (FSK) data interface is used for Analog Display Services Interface (ADSI) and fixed line short message service (SMS). Frequency shift keying is a frequency modulation technique to send digital data over voiced band telephone lines.

The functions listed here support both one-way and two-way frequency shift keying (FSK). See the Voice API Programming Guide for more information about ADSI, two-way FSK, and SMS.

dx_RxIottData( )
receives data on a specified channel

dx_TxIottData( )
transmits data on a specified channel

dx_TxRxIottData( )
starts a transmit-initiated reception of data

1.7 Audio Input Functions

The Audio Input (AI) functions are used to provide music or other information on-hold.

ai_open( )
opens an audio input device

ai_close( )
closes an audio input device

ai_gettxmitslot( )
gets the TDM bus time slot number of the audio input transmit channel

1.8 Transaction Record Functions

Transaction record enables the recording of a two-party conversation by allowing data from two time division multiplexing (TDM) bus time slots from a single channel to be recorded.

dx_mreciottdata( )
records voice data from two TDM bus time slots to a data file, memory or custom device
1.9 **Cached Prompt Management Functions**

The cached prompt management feature enables you to store prompts in on-board memory and play them from this location rather than from the host disk drive. See the *Voice API Programming Guide* for more information about cached prompt management.

- `dx_cacheprompt()` downloads voice data (prompts) from multiple sources to the on-board memory
- `dx_getcachesize()` returns the size of the on-board memory used to store cached prompts

1.10 **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.

- `dx_setevtmsk()` enables detection of CST event(s). User-defined tones are CST events, but detection for these events is enabled using `dx_addtone()` or `dx_enbtone()`, which are global tone detection functions.

The call status transition event functions are:

- `dx_getevt()` gets a CST event in a synchronous environment
- `dx_sendevt()` allows inter-process event communication and sends a specified CST event to a specified device
- `dx_setevtmsk()` enables detection of CST events

1.11 **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 resource sharing bus that allows information to be transmitted and received among resources over multiple time slots.

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.

**Note:** When you see references to the SCbus or SCbus routing, this information also applies to the CT Bus. That is, the physical interboard connection can be either SCbus or CT Bus. The SCbus protocol is used and the SCbus routing API applies to all the boards regardless of whether they use an SCbus or CT Bus physical interboard connection.

A set of TDM routing functions exist for each Intel® Dialogic library, such as fax (fx_ functions), modular station interface (ms_ functions), and conferencing (dcb_ functions). See the appropriate API Library Reference for more information on these functions.

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:

- `ag_getctinfo()`
  - returns information about an analog device connected to the TDM bus

- `ag_getxmitslot()`
  - returns the number of the TDM bus time slot connected to the transmit component of an analog channel

- `ag_listen()`
  - connects the listen (receive) component of an analog channel to the TDM bus time slot

- `ag_unlisten()`
  - disconnects the listen (receive) component of an analog channel from the TDM bus time slot

- `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
1.12 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()` function. 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
- `dx_enbtone()` - enables detection of user-defined tones
- `dx_setgtdamp()` - sets amplitudes used by global tone detection (GTD)

1.13 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_bldtngen()` 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_bldtngen()` - builds a user-defined tone template structure, TN_GEN
Function Summary by Category

---

**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.

1.14 **R2/MF Convenience Functions**

R2/MF convenience functions enable detection of R2/MF forward signals on a channel, and play R2/MF backward signals in response. For more information about voice support for R2/MF, see the Voice API Programming Guide.

*Note:* R2/MF signaling is typically accomplished through the Global Call API. For more information, see the Global Call documentation set. The R2/MF functions listed here are provided for backward compatibility only and should not be used for R2/MF signaling.

**r2_creatfsig()**

creates R2/MF forward signal tone

**r2_playbsig()**

plays R2/MF backward signal tone

1.15 **Speed and Volume Functions**

Speed and volume functions adjust the speed and volume of the play. A speed modification table and volume modification table are associated with each channel, and can be used for increasing or decreasing the speed or volume. These tables have default values which can be changed using the **dx_setsvmt()** function.

The **dx_addspddig()** and **dx_addvoldig()** functions are convenience functions that specify a digit and an adjustment to occur on that digit, without having to set any data structures. These functions use the default settings of the speed and volume modification tables.

See the Voice API Programming Guide for more information about the speed and volume feature in general, and speed and volume modification tables in particular.

The speed and volume functions are:

**dx_adjsv()**

- adjusts speed or volume immediately

**dx_addspddig()**

- sets a dual tone multi-frequency (DTMF) digit for speed adjustment

**dx_addvoldig()**

- adds a dual tone multi-frequency (DTMF) digit for volume adjustment
dx_clrsvcond() clears speed or volume conditions

dx_getcursv() returns current speed and volume settings

dx_getsvmt() returns current speed or volume modification table

dx_setsvcond() sets conditions (such as digit) for speed or volume adjustment; also sets conditions for play (pause and resume)

dx_setsvmt() changes default values of speed or volume modification table

1.16 Call Progress Analysis Functions

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

Notes: 1. Two forms of call progress analysis exist: basic and PerfectCall (formerly called “enhanced call analysis”). PerfectCall call progress analysis uses an improved method of signal identification and can detect fax machines and answering machines. Basic call progress analysis provides backward compatibility for older applications written before PerfectCall call progress analysis became available.

2. Throughout this document, call progress analysis refers to PerfectCall call progress analysis unless otherwise noted.

The call progress analysis functions are:

dx_chgdur() changes the default call progress analysis signal duration

dx_chgfreq() changes the default call progress analysis signal frequency

dx_chgrepct() changes the default call progress analysis signal repetition count

dx_initcallp() initializes call progress analysis on a channel

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
1.17 **Caller ID Functions**

Caller ID functions are used to handle caller ID requests. Caller ID is enabled by setting a channel-based parameter in `dx_setparm()` function. See the *Voice API Programming Guide* for more information about caller ID.

- `dx_gtcallid()` returns the calling line directory number (DN)
- `dx_gtextcallid()` returns the requested caller ID message by specifying the message type ID
- `dx_wtcallid()` waits for rings and reports caller ID, if available

1.18 **File Manipulation Functions**

These file manipulation functions map to C run-time functions, and can only be used if the file is opened with the `dx_fileopen()` 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()` obtains 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

1.19 **Echo Cancellation Resource Functions**

The echo cancellation resource (ECR) feature is a voice channel mode that reduces the echo component in an external TDM bus time slot signal. The echo cancellation resource functions enable use of the ECR feature.

*Note:* The ECR functions have been replaced by the continuous speech processing (CSP) API functions. CSP provides enhanced echo cancellation. For more information, see the *Continuous Speech Processing API Programming Guide* and *Continuous Speech Processing API Programming Guide*.

- `dx_getxmitslotecr()` provides the TDM bus transmit time-slot number of the specified voice channel device when in ECR mode
dx_listenecr( )

enables echo cancellation on a specified voice channel and connects the voice channel to the
echo-referenced signal on the specified TDM bus time slot (ECR mode)

dx_listene crex( )

performs identically to dx_listenecr( ) and also provides the ability to modify the
characteristics of the echo canceller

dx_unlistenecr( )

disables echo cancellation on a specified voice channel and disconnects the voice channel from
the echo-referenced signal (SVP mode)

1.20 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

1.21 Syntellect License Automated Attendant Functions

These functions are used with Intel® Dialogic® products that are licensed for specific telephony
patents held by Syntellect Technology Corporation. For more information, see the Voice API
Programming Guide.

li_attendant( )

performs the actions of an automated attendant

li_islicensed_syntellect( )

verifies Syntellect patent license on board

1.22 Extended Attribute Functions

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

ATDX_ANSRSIZ( )

returns the duration of the answer detected during call progress analysis

ATDX_BDNAMEP( )

returns a pointer to the board device name string

ATDX_BDTYPE( )

returns the board type for the device
Function Summary by Category

ATDX_BUFDIGS()
returns the number of digits in the firmware since the last dx_getdig() for a given channel

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_CRTNID()
returns the identifier of the tone that caused the most recent call progress analysis termination

ATDX_DEVTYPE()
returns device type (board or channel)

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

ATDX_FRQDUR()
returns the duration of the first special information tone (SIT) frequency

ATDX_FRQDUR2()
returns the duration of the second special information tone (SIT) frequency

ATDX_FRQDUR3()
returns the duration of the third special information tone (SIT) frequency

ATDX_FRQHZ()
returns the frequency of the first detected SIT

ATDX_FRQHZ2()
returns the frequency of the second detected SIT

ATDX_FRQHZ3()
returns the frequency of the third detected SIT

ATDX_FRQOUT()
returns the percentage of frequency out of bounds detected during call progress analysis

ATDX_FWVER()
returns the firmware version

ATDX_HOOKST()
returns the current hook state of the channel

ATDX_LINEST()
returns the current line status of the channel

ATDX_LONGLOW()
returns the duration of longer silence detected during call progress analysis
ATDX_PHYADDR()  
returns the physical address of board

ATDX_SHORTLOW()  
returns the duration of shorter silence detected during call progress analysis

ATDX_SIZEHI()  
returns the duration of non-silence detected during call progress analysis

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

1.23 Voice Function Support by Platform

Table 1 provides an alphabetical listing of voice API functions. The table indicates which platforms (DM3 or Springware) are supported for each of the functions.

DM3 boards refers to products based on the Intel® DM3 mediastream architecture. Typically DM3 board names have the prefix “DM,” such as Intel® NetStructure™ DM/V2400A-PCI. Springware boards refer to boards based on earlier-generation architecture. Typically Springware board names have the prefix “D,” such as Intel® Dialogic® D/240JCT-T1.

Although a function may be supported on both DM3 and Springware boards, there may be some restrictions on its use. For example, some parameters or parameter values may not be supported. For details, see the function reference descriptions in Chapter 2, “Function Information”.

Table 1. Voice Function Support by Platform

<table>
<thead>
<tr>
<th>Function</th>
<th>DM3</th>
<th>Springware</th>
</tr>
</thead>
<tbody>
<tr>
<td>ag_getctinfo()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>ag_getxmitslot()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>ag_listen()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>ag_unlisten()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>ai_close()</td>
<td>S</td>
<td>NS</td>
</tr>
<tr>
<td>ai_getxmitslot()</td>
<td>S</td>
<td>NS</td>
</tr>
</tbody>
</table>

NS = Not supported
S = Supported
* = Variances between platforms; refer to the function reference for more information.
† = Asynchronous and synchronous mode supported (all other functions support synchronous mode only)
‡ = On DM3, call progress analysis is available directly through dx_dial().
### Table 1. Voice Function Support by Platform (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>DM3</th>
<th>Springware</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATDX_ANSRSIZ()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_BNAMEP()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_BDTYPE()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_BUFDIGS()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_CHNAMES()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_CHNUM()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_CONNTYPE()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_CPERROR()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_CPTERM()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_CRTNID()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_DEVTYPE()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_DTNFAIL()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_FRQDUR()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_FRQDUR2()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_FRQDUR3()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_FRQHZ()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_FRQHZ2()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_FRQHZ3()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_FRQOUT()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_FWVER()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_HOOKST()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_LINEST()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_LONGLOW()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_PHYADDR()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_SHORTLOW()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_SIZEHI()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_STATE()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_TERMSK()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_TONEID()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>ATDX_TRCOUNT()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_addspddig()</td>
<td>S*</td>
<td>S</td>
</tr>
<tr>
<td>dx_addtone()</td>
<td>S*</td>
<td>S</td>
</tr>
</tbody>
</table>

NS = Not supported  
S = Supported  
* = Variances between platforms; refer to the function reference for more information.  
† = Asynchronous and synchronous mode supported (all other functions support synchronous mode only)  
‡ = On DM3, call progress analysis is available directly through dx_dial().
### Table 1. Voice Function Support by Platform (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>DM3</th>
<th>Springware</th>
</tr>
</thead>
<tbody>
<tr>
<td>dx_addvoldig( )</td>
<td>S *</td>
<td>S</td>
</tr>
<tr>
<td>dx_adjsv( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_blddt( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_blddtcad( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_bldst( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_bldstcad( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_bidtngen( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_cacheprompt( ) †</td>
<td>S</td>
<td>NS</td>
</tr>
<tr>
<td>dx_chgdur( )</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>dx_chgfreq( )</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>dx_chgexpcnt( )</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>dx_close( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_CloseStreamBuffer( )</td>
<td>S</td>
<td>NS</td>
</tr>
<tr>
<td>dx_clrcap( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_cirldigbuf( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_clrsvcond( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_cirpt( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_createtone( ) †</td>
<td>S</td>
<td>NS</td>
</tr>
<tr>
<td>dx_deletetone( ) †</td>
<td>S</td>
<td>NS</td>
</tr>
<tr>
<td>dx_deltones( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_dial( ) †</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_distone( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_enbtone( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_fileclose( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_fileopen( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_fileerrno( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_fileerror( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_fileread( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_fileseek( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_filewrite( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_getcachesize( )</td>
<td>S</td>
<td>NS</td>
</tr>
<tr>
<td>dx_getctinfo( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_getcursv( )</td>
<td>S</td>
<td>S</td>
</tr>
</tbody>
</table>

NS = Not supported  
S = Supported  
* = Variances between platforms; refer to the function reference for more information.  
† = Asynchronous and synchronous mode supported (all other functions support synchronous mode only)  
‡ = On DM3, call progress analysis is available directly through `dx_dial()`.
Table 1. Voice Function Support by Platform (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>DM3</th>
<th>Springware</th>
</tr>
</thead>
<tbody>
<tr>
<td>dx_getdig()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_GetDllVersion()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_getevt()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_getfeaturelist()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_getparm()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_GetRscStatus()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>dx_GetStreamInfo()</td>
<td>S</td>
<td>NS</td>
</tr>
<tr>
<td>dx_getsvmt()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_getxmitslot()</td>
<td>S *</td>
<td>S</td>
</tr>
<tr>
<td>dx_getxmitslotecr()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>dx_gtcallid()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>dx_getxtcallid()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>dx_getsernum()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_initcallp()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>dx_libinit()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_listen()</td>
<td>S *</td>
<td>S</td>
</tr>
<tr>
<td>dx_listenecr()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>dx_listenecrex()</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>dx_open()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_OpenStreamBuffer()</td>
<td>S</td>
<td>NS</td>
</tr>
<tr>
<td>dx_mreciottdata()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_pause()</td>
<td>S</td>
<td>NS</td>
</tr>
<tr>
<td>dx_play()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_playf()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_playiottdata()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_playtone()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_playtoneEx()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_playwav()</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_PutStreamData()</td>
<td>S</td>
<td>NS</td>
</tr>
<tr>
<td>dx_querytone()</td>
<td>S</td>
<td>NS</td>
</tr>
<tr>
<td>dx_rec()</td>
<td>S *</td>
<td>S</td>
</tr>
<tr>
<td>dx_recf()</td>
<td>S</td>
<td>S</td>
</tr>
</tbody>
</table>

NS = Not supported
S = Supported
* = Variances between platforms; refer to the function reference for more information.
† = Asynchronous and synchronous mode supported (all other functions support synchronous mode only)
‡ = On DM3, call progress analysis is available directly through dx_dial().
### Table 1. Voice Function Support by Platform (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>DM3</th>
<th>Springware</th>
</tr>
</thead>
<tbody>
<tr>
<td>dx_reciottdata( ) †</td>
<td>S *</td>
<td>S</td>
</tr>
<tr>
<td>dx_recvox( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_recwav( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_ResetStreamBuffer( )</td>
<td>S</td>
<td>NS</td>
</tr>
<tr>
<td>dx_resume( )</td>
<td>S</td>
<td>NS</td>
</tr>
<tr>
<td>dx_RxIottData( ) †</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_sendvt( )</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>dx_setchxfercnt( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_setdevuio( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_setdigbuf( )</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>dx_setdigtyp( )</td>
<td>S *</td>
<td>S</td>
</tr>
<tr>
<td>dx_setevtmrk( )</td>
<td>S *</td>
<td>S</td>
</tr>
<tr>
<td>dx_setgtdamp( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_sethook( ) †</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>dx_setparm( )</td>
<td>S *</td>
<td>S</td>
</tr>
<tr>
<td>dx_SetRecordNotifyBeepTone( )</td>
<td>S</td>
<td>NS</td>
</tr>
<tr>
<td>dx_setsvcond( )</td>
<td>S *</td>
<td>S</td>
</tr>
<tr>
<td>dx_setsvmt( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_settonelen( )</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>dx_setuio( )</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_SetWaterMark( )</td>
<td>S</td>
<td>NS</td>
</tr>
<tr>
<td>dx_stopch( ) †</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_TSFStatus( )</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>dx_TxIottData( ) †</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_TxRxIottData( ) †</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>dx_unlisten( )</td>
<td>S *</td>
<td>S</td>
</tr>
<tr>
<td>dx_unlistenecr( )</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>dx_wink( ) †</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>dx_wtcallid( )</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>dx_wtring( )</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>li_attendant( )</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>li_islicensed_syntellect( )</td>
<td>NS</td>
<td>S</td>
</tr>
</tbody>
</table>

NS = Not supported  
S = Supported  
* = Variances between platforms; refer to the function reference for more information.  
† = Asynchronous and synchronous mode supported (all other functions support synchronous mode only)  
‡ = On DM3, call progress analysis is available directly through dx_dial().
### Table 1. Voice Function Support by Platform (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>DM3</th>
<th>Springware</th>
</tr>
</thead>
<tbody>
<tr>
<td>nr_scrount( )</td>
<td>S *</td>
<td>S</td>
</tr>
<tr>
<td>nr_scunroute( )</td>
<td>S *</td>
<td>S</td>
</tr>
<tr>
<td>r2_creatfsig( )</td>
<td>NS</td>
<td>S</td>
</tr>
<tr>
<td>r2_playbsig( ) †</td>
<td>NS</td>
<td>S</td>
</tr>
</tbody>
</table>

**Legend:**
- **NS** = Not supported
- **S** = Supported
- † = Variances between platforms; refer to the function reference for more information.
- † = Asynchronous and synchronous mode supported (all other functions support synchronous mode only)
- ‡ = On DM3, call progress analysis is available directly through dx_dial().
This chapter provides an alphabetical reference to the functions in the voice library.

2.1 Function Syntax Conventions

The voice functions use the following syntax:

```
data_type voice_function(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
ag_getctinfo( ) — get information about an analog device

ag_getctinfo( )

Name: int ag_getctinfo(chdev, ct_devinfop)

Inputs:
- int chdev • valid analog 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: Routing
Mode: synchronous
Platform: Springware

Description

The ag_getctinfo( ) function returns information about an analog channel on an analog device. This information is contained in a CT_DEVINFO structure.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid analog channel handle obtained when the channel was</td>
</tr>
<tr>
<td></td>
<td>opened using dx_open( )</td>
</tr>
<tr>
<td>ct_devinfop</td>
<td>specifies a pointer to the data structure CT_DEVINFO</td>
</tr>
</tbody>
</table>

Cautions

This function will fail if an invalid 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 library index number

EDX_SH_BADTYPE
Invalid 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; use dx_fileerrno( ) to obtain error value

■ Example

#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 (ag_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

• dt_getctinfo( ) in the Digital Network Interface Software Reference
• dx_getctinfo( )
ag_getxmitslot( ) — get TDM bus time slot number of analog transmit channel

### Description

The `ag_getxmitslot()` function provides the TDM bus time slot number of the analog transmit channel. This information is contained in an SC_TSINFO structure that also includes the number of the time slot connected to the analog transmit channel. For more information on this structure, see SC_TSINFO, on page 529.

**Note:** Routing convenience function `nr_scroute()` includes `ag_getxmitslot()` functionality.

#### Parameter Description

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

An analog channel can transmit on only one TDM bus time slot.

### Cautions

This function fails if 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
get TDM bus time slot number of analog transmit channel — ag_getxmitslot()

EDX_SH_BADINDX
Invalid Switch Handler library index number

EDX_SH_BADLCTS
Invalid channel number

EDX_SH_BADMODE
Function is not supported in current bus configuration

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

EDX_SH_CMDBLOCK
Blocking command is in progress

EDX_SH_LCLDSCNCT
Channel is already disconnected from TDM bus time slot

EDX_SH_LIBBSY
Switch Handler library is busy

EDX_SH_LIBNOTINIT
Switch Handler library is uninitialized

EDX_SH_MISING
Switch Handler is not present

EDX_SH_NOCLK
Switch Handler clock fallback failed

EDX_SYSTEM
Error from operating system; use dx_fileerrno() to obtain error value

■ Example

```c
#include <windows.h>
#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 devices */
    if ((chdev = dx_open("dxxxB1C1", 0)) == -1) {
        /* process error */
    }

    /* 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);
    }

    printf("%s is transmitting on TDM bus time slot %d", ATDV_NAMEP(chdev), ...scts);
}```
See Also

- ag_listen()
- dt_listen() in the Digital Network Interface Software Reference
- dx_listen()
- fx_listen() in the Fax Software Reference
- ms_listen() in the Modular Station Interface API Library Reference
**ag_listen( )**

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

**Inputs:**
- int chdev • valid analog 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
dxxxlib.h

**Category:** Routing

**Mode:** synchronous

**Platform:** Springware

---

**Description**

The **ag_listen( )** function connects an analog receive channel to a TDM bus time slot. This function uses the information stored in the **SC_TSINFO** data structure to connect the analog receive (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 analog transmit channel.

Due to analog signal processing on voice boards with on-board analog devices, a voice device and its corresponding analog device (analog device 1 to voice device 1, etc.) comprise a single channel. At system initialization, default TDM bus routing is to connect these devices in full-duplex communications.

**Note:** Routing convenience function **nr_scroute( )** includes **ag_listen( )** functionality.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid analog channel device handle obtained when the channel was opened using <strong>dx_open( )</strong></td>
</tr>
<tr>
<td>sc_tsinfop</td>
<td>specifies a pointer to the <strong>SC_TSINFO</strong> structure. For more information on this structure, see <strong>SC_TSINFO</strong>, on page 529.</td>
</tr>
</tbody>
</table>

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

---

**Cautions**

This function will fail when an invalid channel handle or 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_SH_BADEXTTS
  TDM bus time slot is not supported at current clock rate

EDX_SH_BADINDEX
  Invalid Switch Handler index number

EDX_SH_BADLCLTTS
  Invalid channel number

EDX_SH_BADMODE
  Function is not supported in current bus configuration

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

EDX_SH_CMDBLOCK
  Blocking command is in progress

EDX_SH_LCLTSCNCT
  Channel is already connected to TDM bus time slot

EDX_SH_LIBBSY
  Switch Handler library is busy

EDX_SH_LIBNOTINIT
  Switch Handler library is uninitialized

EDX_SH_NOCLK
  Switch Handler clock fallback failed

EDX_SYSTEM
  System error

Example

#include <windows.h>
#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 devices */
if ((chdev = dx_open("dxxxB1Cl", 0)) == -1) {
    /* process error */
}

/* 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 voice channel 1 on board 1 */
if (dx_getxmitslot(chdev, &sc_tsinfo) == -1) {
    printf("Error message = %s", ATDV_ERRMSGP(chdev));
    exit(1);
}

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

See Also

- dx_getxmitslot()
- dt_getxmitslot() in the Digital Network Interface Software Reference
- fx_getxmitslot() in the Fax Software Reference
- ag_unlisten()
ag_unlisten( ) — disconnect analog receive channel from TDM bus

ag_unlisten( )

Name: int ag_unlisten(chdev)

Inputs: int chdev  • analog channel device handle

Returns: 0 on success
-1 on error

Includes: srllib.h
dxxxlib.h

Category: Routing

Mode: synchronous

Platform: Springware

Description

The ag_unlisten( ) function disconnects an analog receive channel from the TDM bus. This function disconnects the analog receive (listen) channel from the TDM bus time slot it was listening to.

Calling ag_listen( ) to connect to a different TDM bus time slot will automatically break an existing connection. Thus, when changing connections, you need not call the ag_unlisten( ) function first.

Note: Routing convenience function nr_scunroute( ) includes ag_unlisten( ) functionality.

<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

This function will fail when an invalid 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_BADCMD
Command is not supported in current bus configuration

EDX_SH_BADINDX
Invalid Switch Handler index number
EDX_SH_BADLCLCTS
Invalid channel number

EDX_SH_BADMODE
Function is not supported in current bus configuration

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

EDX_SH_CMDBLOCK
Blocking command is in progress

EDX_SH_LCLDSCNCT
Channel is already disconnected from TDM bus time slot

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
System error

**Example**

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

main()
{
    int chdev;  /* Voice Channel handle */

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

    /* Disconnect receive of board 1, channel 1 from TDM bus time slot */
    if (ag_unlisten(chdev) == -1) {
        printf("Error message = %s", ATDV_ERRMSGP(chdev));
        exit(1);
    }
}
```

**See Also**

- ag_listen()
ai_close() — close an audio input device

ai_close()

**Name:** int ai_close(devh)

**Inputs:** int devh • valid audio input device handle

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

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

**Category:** Audio Input

**Mode:** synchronous

**Platform:** DM3

---

**Description**

The `ai_close()` function closes an audio input device that was previously opened using `ai_open()`. This function releases the handle and breaks any link between the calling process and the device.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>devh</td>
<td>specifies the valid device handle obtained when an audio input device is opened using <code>ai_open()</code></td>
</tr>
</tbody>
</table>

**Cautions**

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

**Errors**

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.

**Example**

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

int main()
{
    int aidev; /* Audio input device handle */
    SC_TSINFO sc_tsinfo; /* Time slot information structure */
    long scts; /* TDM bus time slot */
```
/* Open audio input device aiB1 */
if ((aidev = ai_open("aiB1")) < 0) {
    /* process error */
}

/* 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 audio input device */
if (ai_getxmitslot(aidev, &sc_tsinfo) < 0) {
    /* process error */
} else {
    printf("%s is transmitting on TDM time slot %d", ATDV_NAMEP(aidev), scts);
}

/* Close audio input device */
if (ai_close(aidev) < 0) {
    /* process error */
}
return 0;

See Also

- ai_getxmitslot()
- ai_open()
**ai_getxmitslot( )**

**Name:** int ai_getxmitslot(devh, sc_tsinfop)

**Inputs:**
- int devh • valid audio input device handle
- SC_TSINFO *sc_tsinfop • pointer to TDM bus time slot information structure

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

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

**Category:** Audio Input

**Mode:** synchronous

**Platform:** DM3

---

### Description

The **ai_getxmitslot( )** function returns the TDM bus time slot number of the audio input transmit channel. The TDM bus time slot information is contained in an SC_TSINFO structure.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>devh</td>
<td>specifies the valid device handle obtained when the audio input device is opened using <strong>ai_open( )</strong></td>
</tr>
<tr>
<td>sc_tsinfop</td>
<td>specifies a pointer to the TDM bus time slot information structure, SC_TSINFO.</td>
</tr>
</tbody>
</table>

The sc_numts field of the SC_TSINFO structure must be initialized with the number of TDM bus time slots requested (1). The sc_tsarrayp field of the SC_TSINFO structure must be initialized with a valid pointer to a long variable. Upon successful return from the function, the long variable will contain the number of the time slot on which the audio input device transmits. For more information on this structure, see SC_TSINFO, on page 529.

### Cautions

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

### Errors

If the function returns -1, use the SRL Standard Attribute function **ATDV_LASTERR( )** to obtain the error code or use **ATDV_ERRMSGP( )** to obtain a descriptive error message.
Example

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

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

    /* Open audio input device aiB1 */
    if ((aidev = ai_open("aiB1")) < 0) {
        /* process error */
    }

    /* 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 audio input device */
    if (ai_getxmitslot(aidev, &sc_tsinfo) < 0) {
        /* process error */
    } else {
        printf("%s is transmitting on TDM time slot %d", ATDV_NAMEP(aidev), scts);
    }

    /* Close audio input device */
    if (ai_close(aidev) < 0) {
        /* process error */
    }
    return 0;
}
```

See Also

- `ai_open`
- `ai_close`
ai_open( )

Name:  int ai_open(namep)

Inputs:  const char *namep  • pointer to an ASCIIZ string that contains the name of a valid audio input device

Returns:  audio input device handle if successful
                        -1 if failure

Includes:  srllib.h
                dxxxlib.h

Category:  Audio Input

Mode:  synchronous

Platform:  DM3

Description

The ai_open( ) function opens an audio input device and returns a unique device handle to identify the device. Until the device is closed, all subsequent references to the opened device must be made using the handle.

<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 audio input device, in the form aiBn, where n is the audio input device number</td>
</tr>
</tbody>
</table>

Cautions

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

Errors

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.

Example

```
#include <windows.h>
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
```
int main()
{
    int         aidev;         /* Audio input device handle */
    SC_TSINFO   sc_tsinfo;     /* Time slot information structure */
    long        scts;          /* TDM bus time slot */

    /* Open audio input device aiB1 */
    if ((aidev = ai_open("aiB1")) < 0) {
        /* process error */
    }

    /* 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 audio input device */
    if (ai_getxmitslot(aidev, &sc_tsinfo) < 0) {
        /* process error */
    } else {
        printf("%s is transmitting on TDM time slot %d", ATDV_NAMEP(aidev), scts);
    }

    /* Close audio input device */
    if (ai_close(aidev) < 0) {
        /* process error */
    }

    return 0;
}

■ See Also

- ai_close()
- ai_getxmitslot()
- sr_getboardcnt()}
ATDX_ANSRSIZ( ) — return the duration of the answer

**Description**

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

See the Voice API Programming Guide for information about call progress analysis. Also see this guide for information about how cadence detection parameters affect a connect and are used to distinguish between a live voice and a voice recorded on an answering machine.

<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
/* Call Progress Analysis with user-specified parameters */

#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
```
main()
{
    int cares, chdev;
    DX_CAP capp;
    .
    /* open the channel using dx_open( ). Obtain channel device descriptor in */
    /* chdev */
    if ((chdev = dx_open("dxxxB1C1", NULL)) == -1) {
        /* process error */
    }
    /* take the phone off-hook */
    if (dx_sethook(chdev, DX_OFFHOOK, EV_SYNC) == -1) {
        /* process error */
    }
    /* Set the DX_CAP structure as needed for call progress analysis. Perform the */
    /* outbound dial with call progress analysis enabled */
    if ((cares = dx_dial(chdev,"5551212", &capp, DX_CALLP|EV_SYNC)) == -1) {
        /* perform error routine */
    }
    switch (cares) {
        case CR_CNCT:     /* Call Connected, get some additional info */
            printf("Duration of short low - \%ld ms\, ATDX_SHORTLOW(chdev)\*10\);
            printf("Duration of long low - \%ld ms\, ATDX_LONGLOW(chdev)\*10\);
            printf("Duration of answer    - \%ld ms\, ATDX_ANSRSIZ(chdev)\*10\);
            break;
        case CR_CEPT:     /* Operator Intercept detected */
            printf("Frequency detected - \%ld Hz\, ATDX_FRQHZ(chdev)\);
            printf("%% of Frequency out of bounds - \%ld Hz\, ATDX_FRQOUT(chdev)\);
            break;
        case CR_BUSY:
            .
            .
            .
        }

    See Also
    - dx_dial()
    - DX_CAP data structure
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 ASCIIZ string “Unknown device” if error

Includes: srllib.h
dxxxlib.h

Category: Extended Attribute

Mode: synchronous

Platform: DM3, Springware

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;
```
return a pointer to the board device name — ATDX_BDNAMEP()

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

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

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

■ See Also

None.
**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:** srllib.h
dxxxlsxlib.h

**Category:** Extended Attribute

**Mode:** synchronous

**Platform:** DM3, Springware

---

### 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>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:

**DL_D41BD**
- D/41 Board Device. This value represents the “dxxxBn type” devices (virtual boards).

**DL_D41CH**
- D/41 Channel Device. This value represents the “dxxxBnCm” type devices (channel device).

The values DL_D41BD and DL_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 <srllib.h>
#include <dxxxlib.h>

#define ON 1

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

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

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

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

**See Also**

None.
**ATDX_BUFDIGS()**

**Name:** long ATDX_BUFDIGS(chdev)

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

**Returns:**
- number of uncollected digits in the firmware buffer if successful
- AT_FAILURE if error

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

**Category:** Extended Attribute

**Mode:** synchronous

**Platform:** DM3, Springware

---

**Description**

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

This function is supported on DM3 boards but must be manually enabled. You must enable the function before the application is loaded in memory. To enable this function, set parameter SupportForSignalCounting to 1 in Key HKEY_LOCAL_MACHINE\SOFTWARE\Dialogic\Cheetah\CC.

To subsequently disable this function, set this parameter to 0.

<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**

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

**Errors**

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

**Example**

```c
#include <fcntl.h>
#include <srllib.h>
#include <dxxxlib.h>
```
main()
{
    int chdev;
    long bufdigs;
    DX_IOTT iott;
    DV_TPT tpt[2];

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

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

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

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

    /* Play a voice file. Terminate on receiving 4 digits, the digit "5" or
     * at end of file. */
    if (dx_play(chdev, &iott, tpt, EV_SYNC) == -1) {
        /* process error */
    }

    /* Check # of digits collected and continue processing. */
    if ((bufdigs = ATDX_BUFDIGS(chdev)) == AT_FAILURE) {
        /* process error */
    }

    ....
}

See Also

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

Name: char ** ATDX_CHNAMES(bddev)

Inputs: int bddev • valid board device handle

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

Includes: srllib.h
dxxxlib.h

Category: Extended Attribute

Mode: synchronous

Platform: DM3, Springware

### 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><code>bddev</code></td>
<td>specifies the valid board device handle obtained when the board was opened using <code>dx_open()</code></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) {
```
retrieve all channel names for a board — ATDX_CHNAMES()

    /* Process error */
    
    /* Display channels on board */
    chnames = ATDX_CHNAMES(bddev);
    subdevs = ATDV_SUBDEVS(bddev); /* number of sub-devices on board */
    printf("Channels on this board are:\n");
    for(cnt=0; cnt<subdevs; cnt++) {
        printf("%s\n", *(chnames + cnt));
    }
    /* Call dx_open( ) to open each of the
     * channels and store the device descriptors
     */
    
    See Also
    None.
ATDX_CHNUM( )

Name: long ATDX_CHNUM(chdev)

Inputs: int chdev • valid channel device handle

Returns: channel number if successful
         AT_FAILURE if error

Includes: srllib.h
dxxxlib.h

Category: Extended Attribute

Mode: synchronous

Platform: DM3, Springware

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.

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

```
#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 */
```
return the channel number — ATDX_CHNUM()

if((chno = ATDX_CHNUM(chdev)) == AT_FAILURE) {
    /* Process error */
}
/* Use chno for application-specific purposes */

See Also

None.
ATDX_CONNTYPE( ) — return the connection type for a completed call

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
dxxxlib.h

Category: Extended Attribute

Mode: synchronous

Platform: DM3, Springware

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.

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>
</tbody>
</table>

On DM3 boards, 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

On Springware boards, possible return values are the following:

CON_CAD
Connection due to cadence break

CON_LPC
Connection due to loop current

CON_PVD
Connection due to positive voice detection

CON_PAMD
Connection due to positive answering machine detection
return the connection type for a completed call — ATDX_CONNTYPE() 

- **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 );
    }
```

ATDX_CONNTYPE( ) — return the connection type for a completed call

```c
printf( "call progress analysis returned \n", cares );
if ( cares == CR_CNCT ) {
    switch ( ATDX_CONNTYPE( dxxxdev ) ) {
    case CON_CAD:
        printf( "Cadence Break\n" );
        break;
    case CON_LPC:
        printf( "Loop Current Drop\n" );
        break;
    case CON_PVD:
        printf( "Positive Voice Detection\n" );
        break;
    case CON_PAMD:
        printf( "Positive Answering Machine Detection\n" );
        break;
    default:
        printf( "Unknown connection type\n" );
        break;
    }
}
/*
 * Continue Processing
 * .
 * .
 * /

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

- See Also
  - `dx_dial()`
  - `ATDX_CPTERM()`
  - `DX_CAP` data structure
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

**Platform:** DM3, Springware

---

### 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_mxtimelfrq 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_mctimefreq 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( "Last error = %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 );
    }

    /*
     * 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
- **Platform:** DM3, Springware

### 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.

<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:

- **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_NODIALTONE**
  Timeout occurred while waiting for dial tone. This return value is not supported on DM3 boards.

- **CR_NORB**
  No ringback on called line.
CR_STOPD
Call progress analysis stopped due to dx_stopch().

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

■ Cautions

None.

■ Errors

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

■ Example

```c
/* Call progress analysis with user-specified parameters */
#include <srllib.h>
#include <dxxxlib.h>

main()
{
    int chdev;
    DX_CAP capp;
    ...
    /* open the channel using dx_open(). Obtain channel device descriptor
     * in chdev */
    if ((chdev = dx_open("dxxx81C1",NULL)) == -1) {
        /* process error */
    }
    /* take the phone off-hook */
    if (dx_sethook(chdev,DX_OFFHOOK,EV_SYNC) == -1) {
        /* process error */
    } else {
        /* Clear DX_CAP structure */
        dx_clrcap(&capp);
        /* Set the DX_CAP structure as needed for call progress analysis.
         * Allow 3 rings before no answer. */
        capp.ca_nbrdna = 3;
        /* Perform the outbound dial with call progress analysis enabled. */
        if (dx_dial(chdev,"5551212",&capp,DX_CALLP|EV_SYNC) == -1) {
            /* perform error routine */
        }
    }
    /* Examine last call progress termination on the device */
    switch (ATDX_CPTERM(chdev)) {
        case CR_CNCT:      /* Call Connected, get some additional info */
            ...
            break;
        case CR_CEPT:      /* Operator Intercept detected */
            ...
            break;
    }
}
```
ATDX_CPTERM( ) — return the last result of call progress analysis termination

break;

case AT_FAILURE: /* Error */

■ See Also

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

**Name:** long ATDX_CRTNID(chdev)

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

**Returns:** identifier of the tone that caused the most recent call progress analysis termination, if successful
AT_FAILURE if error

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

**Category:** Extended Attribute

**Mode:** synchronous

**Platform:** Springware

---

**Description**

The ATDX_CRTNID( ) function returns the last call progress analysis termination of the tone that caused the most recent call progress analysis termination of the channel device. See the Voice API Programming Guide for a description of 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>

Possible return values are the following:

TID_BUSY1
First signal busy

TID_BUSY2
Second signal busy

TID_DIAL_INTL
International dial tone

TID_DIAL_LCL
Local dial tone

TID_DIAL_XTRA
Special (“Extra”) dial tone

TID_DISCONNECT
Disconnect tone (post-connect)

TID_FAX1
First fax or modem tone

TID_FAX2
Second fax or modem tone
ATDX_CRTNID( ) — return the last call progress analysis termination

TID_RNGBK1
Ringback (detected as single tone)

TID_RNGBK2
Ringback (detected as dual tone)

■ Cautions

None.

■ Errors

This function fails and returns AT_FAILURE if an invalid device handle is specified.

■ Example

```c
#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 */
  }

  /*
   * 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 */
  }
```
return the last call progress analysis termination — ATDX_CRTNID()

/*
 * Dial
 */
printf("Dialing %s\n", dialstrg);
car = dx_dial(ddd,dialstrg,(DX_CAP *)&cap_s,DX_CALLP|EV_SYNC);
if (car == -1) {
    /* handle error */
}
switch( car ) {
    case CR_NODIALTONE:
        switch( ATDX_DTNFAIL(ddd) ) {
            case 'L':
                printf(" Unable to get Local dial tone\n");
                break;
            case 'I':
                printf(" Unable to get International dial tone\n");
                break;
            case 'X':
                printf(" Unable to get special eXtra dial tone\n");
                break;
        }
        break;
    case CR_BUSY:
        printf(" %s engaged - %s detected\n", dialstrg,
            (ATDX_CRTNID(ddd) == TID_BUSY1 ? "Busy 1" : "Busy 2") );
        break;
    case CR_CNCT:
        printf(" Successful connection to %s\n", dialstrg );
        break;
    default:
        break;
}
/*
 * Set on Hook
 */
if ((dx_sethook( ddd, DX_ONHOOK, EV_SYNC )) == -1) {
    /* handle error */
}
    dx_close( ddd );
}

See Also

None.
**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
- **Platform:** DM3, Springware

### 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><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:

- `DT_DXBD`
  - Board device (indicates virtual board)
- `DT_DXCH`
  - Channel device
- `DT_PHYBD`
  - Physical board 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("dxxxB1", NULL)) == -1) {
        /* Process error */
    }

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

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

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

See Also

None.
ATDX_DTNFAIL( )

**Name:** long ATDX_DTNFAIL(chdev)

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

**Returns:** code for the dial tone that failed to appear
AT_FAILURE if error

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

**Category:** Extended Attribute

**Mode:** synchronous

**Platform:** Springware

---

**Description**

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

### 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>
</tbody>
</table>

Possible return values are the following:

- **L**
  - Local dial tone

- **I**
  - International dial tone

- **X**
  - Special (“extra”) dial tone

**Cautions**

None.

**Errors**

This function fails and returns AT_FAILURE if an invalid device handle is specified.

**Example**

```c
#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 */
    }

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

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

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

default:
    break;
}

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

    dx_close( ddd );
}

See Also

None.
ATDX_FRQDUR( )

Name: long ATDX_FRQDUR(chdev)

Inputs: int chdev • valid channel device handle

Returns: first frequency duration in 10 msec units if success
AT_FAILURE if error

Includes: srllib.h
dxxxlib.h

Category: Extended Attribute

Mode: synchronous

Platform: Springware

Description

The ATDX_FRQDUR( ) function returns the duration of the first Special Information Tone (SIT) sequence in 10 msec units after dx_dial( ) terminated due to an Operator Intercept.

Termination due to Operator Intercept is indicated by ATDX_CPTERM( ) returning CR_CEPT. For information on SIT frequency detection, see the Voice API Programming Guide.

Parameter Description

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

Cautions

None.

Errors

This function fails and returns AT_FAILURE if an invalid channel device handle is specified.

Example

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

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

main()
{
    int cares, chdev;
    DX_CAP capp;
    ...
}
ATDX_FRQDUR( ) — return the duration of the first SIT sequence

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

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

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

switch (cares)    {
  case CR_CNCT:    /* Call Connected, get some additional info */
    printf("\nDuration of short low  - \d ms",ATDX_SHORTLOW(chdev)*10);
    printf("\nDuration of long low  - \d ms",ATDX_LONGLOW(chdev)*10);
    printf("\nDuration of answer    - \d ms",ATDX_ANSRSIZ(chdev)*10);
    break;
  case CR_CEPT:    /* Operator Intercept detected */
    printf("\nFirst frequency detected - \d Hz",ATDX_FRQHZ(chdev));
    printf("\nSecond frequency detected - \d Hz", ATDX_FRQHZ2(chdev));
    printf("\nThird frequency detected - \d Hz", ATDX_FRQHZ3(chdev));
    printf("\nDuration of first frequency - \d ms", ATDX_FRQDUR(chdev));
    printf("\nDuration of second frequency - \d ms", ATDX_FRQDUR2(chdev));
    printf("\nDuration of third frequency - \d ms", ATDX_FRQDUR3(chdev));
    break;
  case CR_BUSY:    /* Operator Busy */
    break;
  ...
}

See Also

- dx_dial()
- ATDX_CPTERM()
- DX_CAP data structure
- call progress analysis topic in the Voice API Programming Guide
- ATDX_FRQDUR2()
- ATDX_FRQDUR3()
- ATDX_FRQHZ()
- ATDX_FRQHZ2()
- ATDX_FRQHZ3()
The ATDX_FRQDUR2() function returns the duration of the second Special Information Tone (SIT) sequence in 10 msec units after dx_dial() terminated due to an Operator Intercept.

Termination due to Operator Intercept is indicated by ATDX_CPTERM() returning CR_CEPT. For information on SIT frequency detection, see the Voice API Programming Guide.

### 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>
</tbody>
</table>

### Cautions

None.

### Errors

This function fails and returns AT_FAILURE if an invalid channel device handle is specified.

### Example

See the example for ATDX_FRQDUR().

### See Also

- dx_dial()
- ATDX_CPTERM()
- DX_CAP data structure
- call progress analysis topic in the Voice API Programming Guide
- ATDX_FRQDUR()
ATDX_FRQDUR2( ) — return the duration of the second SIT sequence

- ATDX_FRQDUR3()
- ATDX_FRQHZ()
- ATDX_FRCHZ2()
- ATDX_FRQHZ3()
### ATDX_FRQDUR3()

**Name:** long ATDX_FRQDUR3(chdev)

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

**Returns:**
- third frequency duration in 10 msec units if success
- AT_FAILURE if error

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

**Category:** Extended Attribute

**Mode:** synchronous

**Platform:** Springware

---

#### Description

The ATDX_FRQDUR3() function returns the duration of the third Special Information Tone (SIT) sequence in 10 msec units after dx_dial() terminated due to an Operator Intercept.

Termination due to Operator Intercept is indicated by ATDX_CPTERM() returning CR_CEPT. For information on SIT frequency detection, 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 dx_open()</td>
</tr>
</tbody>
</table>

#### Cautions

None.

#### Errors

This function fails and returns AT_FAILURE if an invalid channel device handle is specified.

#### Example

See the example for ATDX_FRQDUR().

#### See Also

- dx_dial()
- ATDX_CPTERM()
- DX_CAP data structure
- call progress analysis topic in Voice API Programming Guide
- ATDX_FRQDUR()
**ATDX_FRQDUR3()** — return the duration of the third SIT sequence

- ATDX_FRQDUR2()
- ATDX_FRQHZ()
- ATDX_FRQHZ2()
- ATDX_FRQHZ3()
ATDX_FRQHZ( )

**Name:** long ATDX_FRQHZ(chdev)

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

**Returns:**
- first tone frequency in Hz if success
- AT_FAILURE if error

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

**Category:** Extended Attribute

**Mode:** synchronous

**Platform:** Springware

---

### Description

The ATDX_FRQHZ( ) function returns the frequency in Hz of the first Special Information Tone (SIT) sequence after dx_dial( ) has terminated due to an Operator Intercept.

Termination due to Operator Intercept is indicated by ATDX_CPTERM( ) returning CR_CEPT. For information on SIT frequency detection, 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 dx_open( )</td>
</tr>
</tbody>
</table>

### Cautions

None.

### Errors

This function fails and returns AT_FAILURE if an invalid channel device handle is specified.

### Example

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

```c
/* Call progress analysis with user-specified parameters */
#include <stdio.h>
#include <srllib.h>
#include <dxxlib.h>

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

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

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

switch (cares) {
    case CR_CNCT:      /* Call Connected, get some additional info */
        printf("\nDuration of short low - %ld ms, ATDX_SHORTLOW(\n\nDuration of long low - %ld ms, ATDX_LONGLOW(\n\nDuration of answer - %ld ms, ATDX_ANSRSIZ(\nbreak;
    case CR_CEPT:      /* Operator Intercept detected */
        printf("\nFirst frequency detected - %ld Hz, ATDX_FRQHZ(chdev));
        printf("\nSecond frequency detected - %ld Hz, ATDX_FRQHZ2(chdev));
        printf("\nThird frequency detected - %ld Hz, ATDX_FRQHZ3(chdev));
        printf("\nDuration of first frequency - %ld ms, ATDX_FRQDUR(chdev));
        printf("\nDuration of second frequency - %ld ms, ATDX_FRQDUR2(chdev));
        printf("\nDuration of third frequency - %ld ms, ATDX_FRQDUR3(chdev));
break;
    case CR_BUSY:
        break;
    case CR_BUSY:
        break;
    case CR_BUSY:
        break;
    case CR_BUSY:
        break;
    case CR_BUSY:
        break;
    case CR_BUSY:
        break;
    case CR_BUSY:
        break;
}
```

## See Also
- `dx_dial()`
- `ATDX_CPTERM()`
- `DX_CAP` data structure
- call progress analysis topic in the *Voice API Programming Guide*
- `ATDX_FRQHZ2()`
- `ATDX_FRQHZ3()`
- `ATDX_FRQDUR()`
- `ATDX_FRQDUR2()`
- `ATDX_FRQDUR3()`
return the frequency of the second SIT sequence — ATDX_FRQHZ2( )

ATDX_FRQHZ2( )

**Name:** long ATDX_FRQHZ2(chdev)

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

**Returns:**
- second tone frequency in Hz if success
- AT_FAILURE if error

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

**Category:** Extended Attribute

**Mode:** synchronous

**Platform:** Springware

---

### Description

The ATDX_FRQHZ2( ) function returns the frequency in Hz of the second Special Information Tone (SIT) sequence after dx_dial( ) has terminated due to an Operator Intercept.

Termination due to Operator Intercept is indicated by ATDX_CPTERM( ) returning CR_CEPT. For information on SIT frequency detection, 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 dx_open( )</td>
</tr>
</tbody>
</table>

### Cautions

None.

### Errors

This function fails and returns AT_FAILURE if an invalid channel device handle is specified.

### Example

See the example for ATDX_FRQHZ( ).

### See Also

- dx_dial( )
- ATDX_CPTERM( )
- DX_CAP data structure
- call progress analysis topic in the *Voice API Programming Guide*
- ATDX_FRQHZ( )
ATDX_FRQHZ2( ) — return the frequency of the second SIT sequence

- ATDX_FRQHZ3()
- ATDX_FRQDUR()
- ATDX_FRQDUR2()
- ATDX_FRQDUR3()
ATDX_FRQHZ3( )

Name: long ATDX_FRQHZ3(chdev)

Inputs: int chdev • valid channel device handle

Returns: third tone frequency in Hz if success
AT_FAILURE if error

Includes: srllib.h
dxxlib.h

Category: Extended Attribute

Mode: synchronous

Platform: Springware

Description

The ATDX_FRQHZ3( ) function returns the frequency in Hz of the third Special Information Tone (SIT) sequence after dx_dial( ) has terminated due to an Operator Intercept.

Termination due to Operator Intercept is indicated by ATDX_CPTERM( ) returning CR_CEPT. For information on SIT frequency detection, 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 dx_open( )</td>
</tr>
</tbody>
</table>

Cautions

None.

Errors

This function fails and returns AT_FAILURE if an invalid channel device handle is specified.

Example

See the example for ATDX_FRQHZ( ).

See Also

- dx_dial( )
- ATDX_CPTERM( )
- DX_CAP structure
- call progress analysis topic in the Voice API Programming Guide
- ATDX_FRQHZ( )
ATDX_FRQHZ3() — return the frequency of the third SIT sequence

- ATDX_FRQHZ2()
- ATDX_FRQDUR()
- ATDX_FRQDUR2()
- ATDX_FRQDUR3()
ATDX_FRQOUT( )

**Name:** long ATDX_FRQOUT(chdev)

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

**Returns:**
- percentage frequency out-of bounds
  - AT_FAILURE if error

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

**Category:** Extended Attribute

**Mode:** synchronous

**Platform:** Springware

### Description

The **ATDX_FRQOUT( )** function returns percentage of time SIT tone was out of bounds as specified by the range in the **DX_CAP structure**.

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

For information on SIT frequency detection, 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 <strong>dx_open( )</strong></td>
</tr>
</tbody>
</table>

### Cautions

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

### Errors

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

### Example

```c
/* Call progress analysis with user-specified parameters */
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
```
main()
{
    int cares, chdev;
    DX_CAP capp;
    /* open the channel using dx_open( ). Obtain channel device descriptor in
    * chdev
    */
    if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
        /* process error */
    }
    /* take the phone off-hook */
    if (dx_sethook(chdev,DX_OFFHOOK,EV_SYNC) == -1) {
        /* process error */
    }
    /* Set the DX_CAP structure as needed for call progress analysis. Perform the
    * outbound dial with call progress analysis enabled.
    */
    if ((cares = dx_dial(chdev,"5551212",&capp,DX_CALLP|EV_SYNC)) == -1) {
        /* perform error routine */
    }
    switch (cares) {
        case CR_CNCT:      /* Call Connected, get some additional info */
            printf("\nDuration of short low - %ld ms\n",ATDX_SHORTLOW(chdev)*10);
            printf("\nDuration of long low  - %ld ms\n",ATDX_LONGLOW(chdev)*10);
            printf("\nDuration of answer    - %ld ms\n",ATDX_ANSRSIZ(chdev)*10);
            break;
        case CR_CEPT:      /* Operator Intercept detected */
            printf("\nFrequency detected - %ld Hz\n",ATDX_FRQHZ(chdev));
            printf("\n%% of Frequency out of bounds - %ld Hz\n",ATDX_FRQOUT(chdev));
            break;
        case CR_BUSY:
            break;
        
    }
}

See Also

- dx_dial( )
- ATDX_CPTERM( )
- DX_CAP data structure
- call progress analysis topic in the Voice API Programming Guide
return the voice firmware version number — ATDX_FWVER()

ATDX_FWVER()

**Name:**
long ATDX_FWVER(bddev)

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

**Returns:**
D/4x Firmware version if successful
AT_FAILURE if error

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

**Category:**
Extended Attribute

**Mode:**
synchronous

**Platform:**
Springware

### Description

The ATDX_FWVER() function returns the voice firmware version number or emulated D/4x firmware version number.

<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>

This function returns a 32-bit value in the following format.

TTTT|MMMM|mmmmm|AAAAAAA|aaaaaaa

where each letter represents one bit of data with the following meanings:

<table>
<thead>
<tr>
<th>Letter</th>
<th>Description</th>
</tr>
</thead>
</table>
| T      | Type of Release. Decimal values have the following meanings (example: 0010 for Alpha release):
|       | • 0 – Production
|       | • 1 – Beta
|       | • 2 – Alpha
|       | • 3 – Experimental
|       | • 4 – Special |
| M      | Major version number for a production release in BCD format. Example: 0011 for version “3” |
| m      | Minor version number for a production release in BCD format. Example: 00000001 for “.10” |
| A      | Major version number for a non-production release in BCD format. Example: 00000100 for version “4” |
| a      | Minor version number for a non-production release in BCD format. Example: 00000010 for version “.02” |
Example: 0000 0010 0001 0101 0000 0000 0000 0000 (Production v2.15)

- **Cautions**
  
  None.

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

- **Example**

  ```c
  #include <stdio.h>
  #include <srllib.h>
  #include <dxxxlib.h>
  #include <windows.h>
  
  void GetFwlVersion(char *, long);
  
  main()
  {
    int bddev;
    char *bdname, FWVersion[50];
    long fwver;
    
    bdname = "dxxxB1";
    /*
     * Open board device
     */
    if (!bddev = dx_open(bdname, NULL)) == -1)
    { /* Handle error */
      
    if (fwver = ATDX_FWVER(bddev)) = AT_FAILURE)
    { /* Handle error */
      
    /*
     * Parse fw version
     */
    GetFwlVersion(FWVersion, fwver);
    
    printf("%s\n", FWVersion);
    }
  } /* end main */
  ```

- **See Also**

  None.
The **ATDX_HOOKST( )** function returns the current hook-switch state of the channel **chdev**.

**Note:** This function is not supported on digital interfaces.

---

### Description

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

**Note:** This function is not supported on digital interfaces.

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

Possible return values are the following:

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

---

### Cautions

None.

---

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

See Also

- `dx_sethook()`
- `DX_CST` structure
- `dx_setevtsmk()` for enabling hook state (call status transition events)
- `sr_getevt()` for synchronous call status transition event detection
- `DX_EBLK` for asynchronous call status transition event detection
- `sr_getevtdatap()` in the `Standard Runtime Library API Library Reference`
**ATDX_LINEST( )**

**Name:** long ATDX_LINEST(chdev)

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

**Returns:**
- current line status of channel if successful
- AT_FAILURE if error

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

**Category:** Extended Attribute

**Mode:** synchronous

**Platform:** Springware

---

### Description

The `ATDX_LINEST( )` function returns the current activity on the channel specified in `chdev`. The information is returned in a bitmap.

<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>

Possible return values are the following:

- **RLS_DTMF**
  - DTMF signal present

- **RLS_HOOK**
  - Channel is on-hook

- **RLS_LCSENSE**
  - Loop current not present

- **RLS_RING**
  - Ring not present

- **RLS_RINGBK**
  - Audible ringback detected

- **RLS_SILENCE**
  - Silence on the line

### Cautions

None.
ATDX_LINEST() — return the current activity on the channel

- **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 linest;

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

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

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

- **See Also**

  None.
ATDX_LONGLOW( )

Name: long ATDX_LONGLOW(chdev)

Inputs: int chdev

Returns: duration of longer silence if successful
AT_FAILURE if error

Includes: srllib.h
dxxxlib.h

Category: Extended Attribute
Mode: synchronous
Platform: Springware

Description

The ATDX_LONGLOW( ) function returns duration of longer silence duration in 10 msec units for the initial signal that occurred during call progress analysis on the channel chdev. This function can be used in conjunction with ATDX_SIZEHI( ) and ATDX_SHORTLOW( ) to determine the elements of an established cadence.

See the Voice API Programming Guide for more information on call progress analysis and cadence detection.

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

main()
{
    int cares, chdev;
    DX_CAP capp;
    ...
```
ATDX_LONGLOW( ) — return duration of longer silence detected

,  
/* open the channel using dx_open( ). Obtain channel device descriptor in  
* chdev */  
if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {  
  /* process error */  
}
/* take the phone off-hook */  
if (dx_sethook(chdev,DX_OFFHOOK,EV_SYNC) == -1) {  
  /* process error */  
}
/* Set the DX_CAP structure as needed for call progress analysis. Perform the  
* outbound dial with call progress analysis enabled */  
if ((cares = dx_dial(chdev,"5551212",&capp,DX_CALLP|EV_SYNC)) == -1) {  
  /* perform error routine */  
}  
switch (cares) {
  case CR_CNCT:       /* Call Connected, get some additional info */
    printf("\nDuration of short low  - %ld ms",ATDX_SHORTLOW(chdev)*10);
    printf("\nDuration of long low  - %ld ms",ATDX_LONGLOW(chdev)*10);
    break;
  case CR_CEPT:      /* Operator Intercept detected */
    printf("\nFrequency detected   - %ld Hz",ATDX_FRQHZ(chdev));
    printf("\n%% of Frequency out of bounds - %ld Hz",ATDX_FRQOUT(chdev));
    break;
  case CR_BUSY:   }  
          
See Also

- dx_dial(  
- ATDX_CPTERM(  
- ATDX_SIZEHI(  
- ATDX_SHORTLOW(  
- DX_CAP data structure  
- call progress analysis in the Voice API Programming Guide  
- cadence detection in the Voice API Programming Guide
The `ATDX_PHYADDR()` function returns the physical board address for the board device `bddev`.

**Parameters**

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

**Description**

None.

**Errors**

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

**Example**

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

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

    if ((phyaddr = ATDX_PHYADDR(bddev)) == AT_FAILURE) {
        /* Process error */
    }
}
ATDX_PHYADDR() — return the physical board address

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

- See Also

None.
ATDX_SHORTLOW( )

**Name:** long ATDX_SHORTLOW(chdev)

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

**Returns:**
- duration of shorter silence if successful
- AT_FAILURE if error

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

**Category:** Extended Attribute

**Mode:** synchronous

**Platform:** Springware

---

### Description

The ATDX_SHORTLOW( ) function returns duration of shorter silence in 10 msec units for the initial signal that occurred during call progress analysis on the channel chdev. This function can be used in conjunction with ATDX_SIZEHI( ) and ATDX_LONGLOW( ) to determine the elements of an established cadence.

See the Voice API Programming Guide for more information on call progress analysis and cadence detection.

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

- If the result of ATDX_SHORTLOW( ) is less than the ca_lo2rmin field, this indicates a double ring cadence.
- If the result of ATDX_SHORTLOW( ) is greater than the ca_lo2rmin field, this indicates a single ring.

<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 AT_FAILURE if an invalid channel device handle is specified in chdev.
**Example**

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

main()
{
    int cares, chdev;
    DX_CAP capp;
    .
    /* open the channel using dx_open(). Obtain channel device descriptor
     * in chdev
     */
    if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
        /* process error */
    }
    /* take the phone off-hook */
    if (dx_sethook(chdev,DX_OFFHOOK,EV_SYNC) == -1) {
        /* process error */
    }
    /* Set the DX_CAP structure as needed for call progress analysis. Perform the
     * outbound dial with call progress analysis enabled
     */
    if ((cares = dx_dial(chdev,"5551212", &capp, DX_CALLP|EV_SYNC)) == -1) {
        /* perform error routine */
    }
    switch (cares) {
        case CR_CNCT:      /* Call Connected, get some additional info */
            printf("\nDuration of short low - %ld ms, ATDX_SHORTLOW(chdev)*10);%
            printf("\nDuration of long low  - %ld ms, ATDX_LONGLOW(chdev)*10);%
            printf("\nDuration of answer    - %ld ms, ATDX_ANSRSIZ(chdev)*10);%
            break;
        case CR_CEPT:     /* Operator Intercept detected */
            printf("\nFrequency detected - %ld Hz, ATDX_FRQHZ(chdev));
            printf("\n%% of Frequency out of bounds - %ld Hz, ATDX_FRQOUT(chdev));
            break;
        case CR_BUSY:
            .
    }
}

**See Also**

- dy_dial()
- ATDX_LONGLOW()
- ATDX_SIZEHI()
- ATDX_CPTERM()
- DX_CAP data structure
- call progress analysis in the *Voice API Programming Guide*
- cadence detection in the *Voice API Programming Guide*
ATDX_SIZEHI( )

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

Description

The ATDX_SIZEHI( ) function returns duration of initial non-silence in 10 msec units that occurred during call progress analysis on the channel chdev. This function can be used in conjunction with ATDX_SHORTLOW( ) and ATDX_LONGLOW( ) to determine the elements of an established cadence.

See the Voice API Programming Guide for more information on call progress analysis and cadence detection.

<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 AT_FAILURE if an invalid channel device handle is specified in chdev.

Example

/* Call progress analysis with user-specified parameters */
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
main()
{
    int cares, chdev;
    DX_CAP capp;
    ...
}
ATDX_SIZEHI( ) — return duration of initial non-silence

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

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

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

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

}  

See Also

- dx_dial()
- ATDX_LONGLOW()
- ATDX_SHORTLOW()
- ATDX_CPTERM()
- DX_CAP data structure
- call progress analysis in the Voice API Programming Guide
- cadence detection in the Voice API Programming Guide
**Description**

The ATDX_STATE( ) function returns the current state of the channel `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>

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
ATDX_STATE() — return the current state of the channel

CS_WINK
   Wink state

When a VFX combined resource board is being used to send and receive faxes the following states may be returned:

CS_SENDFAX
   Channel is in a fax transmission state.

CS_RECVFAX
   Channel is in a fax reception state.

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

Cautions

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("dxxxB1C1",NULL)) == -1) {
        /* Process error */
    }

    /* Examine state of the channel. Perform application specific action based on state of the channel */
    if ((chstate = ATDX_STATE(chdev)) == AT_FAILURE) {
        /* Process error */
    }

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

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
dxxxlib.h
Category: Extended Attribute
Mode: synchronous
Platform: DM3, Springware

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.

Parameter Description

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

On DM3 boards, 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_MAXDATA
  - Maximum data reached; returned when the last I/O function terminates on DX_MAXDATA
- TM_MAXDTMF
  - Maximum DTMF count
- TM_MAXNOSIL
  - Maximum period of non-silence
- TM_MAXSIL
  - Maximum period of silence
- TM_MAXTIME
  - Maximum function time exceeded
ATDX_TERMMSK( ) — return the reason for the last I/O function termination

TM_NORMTERM
   Normal termination (for dx_dial( ), dx_sethook( ))

TM_TONE
   Tone-on/off event

TM_USRSTOP
   Function stopped by user

On Springware boards, 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_LCOFF
   Loop current off.

TM_MAXDATA
   Maximum data reached; returned when the last I/O function terminates on DX_MAXDATA

TM_MAXDTMF
   Maximum DTMF count

TM_MAXNOSIL
   Maximum period of non-silence

TM_MAXSIL
   Maximum period of silence

TM_MAXTIME
   Maximum function time exceeded

TM_NORMTERM
   Normal termination (for dx_dial( ), dx_sethook( ))

TM_PATTERN
   Pattern matched silence off

TM_TONE
   Tone-on/off event

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.
return the reason for the last I/O function termination — \texttt{ATDX\_TERMMSK()} \\

- On DM3 boards, when both DX\_MAXDTMF and DX\_DIGMASK termination conditions are specified in the DV\_TPT structure, and both conditions are satisfied, the \texttt{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 \texttt{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.

\section*{Errors}

This function will fail and return AT\_FAILURE if an invalid channel device handle is specified in \texttt{chdev}.

\section*{Example}

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

main()
{
    int chdev;
    long term;
    DX\_IOTT iott;
    DV\_TPT tpt[4];

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

    /* Record a voice file. Terminate on receiving a digit, silence, loop * current drop, max time, or reaching a byte count of 50000 bytes. */
    /* set up DX\_IOTT */
    iott.io\_type = IO\_DEV|IO\_EOT;
    iott.io\_bufp = 0;
    iott.io\_offset = 0;
    iott.io\_length = 50000;
    if((iott.io\_fhandle = dx\_fileopen("file.vox", O\_RDWR)) == -1) {
        /* Process error */
    }

    /* set up DV\_TPTs for the required terminating conditions */
    dx\_clrtpt(tpt,4);
    tpt[0].tp\_type = IO\_CONT;
    tpt[0].tp\_termno = DX\_MAXDTMF;            /* Maximum digits */
    tpt[0].tp\_length = 1;                      /* terminate on the first digit */
    tpt[0].tp\_flags = TF\_MAXDTMF;            /* Use the default flags */
    tpt[1].tp\_type = IO\_CONT;
    tpt[1].tp\_termno = DX\_MAXTIME;            /* Maximum time */
    tpt[1].tp\_length = 100;                    /* terminate after 10 secs */
    tpt[1].tp\_flags = TF\_MAXTIME;            /* Use the default flags */
    tpt[2].tp\_type = IO\_CONT;
    tpt[2].tp\_termno = DX\_DIGMASK;            /* Maximum digits */
    tpt[2].tp\_length = 1;                      /* terminate on the first digit */
    tpt[2].tp\_flags = TF\_DIGMASK;            /* Use the default flags */
    tpt[3].tp\_type = IO\_CONT;
```
ATDX_TERMMSK( ) — return the reason for the last I/O function termination

```c
#define ATDX_MAXSIL  DX_MAXSIL; /* Maximum Silence */
#define ATDX_MAXTIME  DX_MAXSIL; /* terminate on 3 sec silence */
#define ATDX_MAXFILT  IO_MAXFILT; /* last entry in the table */
#define ATDX_LCOFF  DX_LCOFF; /* terminate on loop current drop */
#define ATDX_LCTIME  DX_LCOFF; /* terminate on 1 sec silence */
#define ATDX_LCFILT  TF_LCFILT; /* 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
**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

**Platform:** DM3, Springware

---

### 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 dx_open( )</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>
#include <windows.h>

#define TID_1   101

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

    // Code...
}
```
ATDX_TONEID( ) — return user-defined tone ID that terminated I/O function

#include "dxfunc.h"

int main() {
    // Open the D/xxx Channel Device and Enable a Handler
    /*
    * 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.
    /*
    * 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
    /*
    * 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.
    /*
    * 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
    /*
    * 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, tpt, EV_SYNC ) == -1 ){
        printf( "Unable to Play the Tone\n" );
        printf( "Lasterror = %d  Err Msg = %s\n",
                ATDV_LASTERR( chdev ), ATDV_ERRMSGP( chdev ) );
        dx_close( chdev );
        exit( 1 );
    }

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

    return 0;
}

return user-defined tone ID that terminated I/O function — ATDX_TONEID()

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

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

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

- See Also

None.
ATDX_TRCOUNT( ) — return the byte count for the last I/O transfer

**Name:** long ATDX_TRCOUNT(chdev)

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

**Returns:**
- last play/record transfer count if successful
- AT_FAILURE if error

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

**Category:** Extended Attribute

**Mode:** synchronous

**Platform:** DM3, Springware

### Description

The ATDX_TRCOUNT( ) function returns the number of bytes transferred during the last play or record on the channel 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 dx_open()</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 <fcntl.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

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

    /* Open the channel device */
    if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
        /* Process error */
    }
```
return the byte count for the last I/O transfer — ATDX_TRCOUNT( )

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

/*
 * set up DX_IOTT */
io_type = IO_DEV|IO_EOT;
iott.io_bufp = 0;
iott.io_offset = 0L;
iott.io_length = 50000L;
if((iott.io_fhandle = dx_fileopen("file.vox", O_RDWR)) == -1) {
    /* process error */
}

/* set up DV_TPTs for the required terminating conditions */
dx_clrtpt(tpt, 2);
tpt[0].tp_type = IO_CONT;
tpt[0].tp_termno = DX_MAXDTMF;  /* Maximum digits */
tpt[0].tp_length = 1;            /* terminate on the first digit */
tpt[0].tp_flags = TF_MAXDTMF;    /* Use the default flags */
tpt[1].tp_type = IO_EOT;
tpt[1].tp_termno = DX_MAXTIME;   /* Maximum time */
tpt[1].tp_length = 100;          /* terminate after 10 secs */
tpt[1].tp_flags = TF_MAXTIME;    /* Use the default flags */

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

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

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

• See Also

None.
**dx_addspddig( ) — set a DTMF digit to adjust speed**

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

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

For more information about speed and volume control as well as speed and volume modification tables, see the *Voice API Programming Guide*. For information about speed and 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 <code>dx_open( )</code></td>
</tr>
</tbody>
</table>
set a DTMF digit to adjust speed — dx_addspddig( )

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>digit</td>
<td>specifies a DTMF digit (0-9, *,#) that will modify speed by the amount specified in adjval</td>
</tr>
<tr>
<td>adjval</td>
<td>specifies a speed adjustment value to take effect whenever the digit specified in digit occurs:</td>
</tr>
</tbody>
</table>

On DM3 boards, the following are valid values:

- SV_ADD10PCT – increase play speed by 10%
- SV_NORMAL – set play speed to origin (regular speed) when the play begins. digit must be set to NULL.
- SV_SUB10PCT – decrease play speed by 10%

On Springware boards, the following are valid values:

- SV_ADD10PCT – increase play speed by 10%
- SV_ADD20PCT – increase play speed by 20%
- SV_ADD30PCT – increase play speed by 30%
- SV_ADD40PCT – increase play speed by 40%
- SV_ADD50PCT – increase play speed by 50%
- SV_NORMAL – set play speed to origin (regular speed) when the play begins. digit must be set to NULL.
- SV_SUB10PCT – decrease play speed by 10%
- SV_SUB20PCT – decrease play speed by 20%
- SV_SUB30PCT – decrease play speed by 30%
- SV_SUB40PCT – decrease play speed by 40%

To start play speed at the origin, set digit to NULL and set adjval to SV_NORMAL.

■ Cautions

- On DM3 boards, 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.
- On DM3 boards, digits that are used for play adjustment may also be used as a terminating condition. If a digit is defined as both, then both actions are applied upon detection of that digit.
- On Springware boards, digits that are used for play adjustment will not be used as a terminating condition.
- Calls to this function are cumulative. To reset or remove any condition, you should clear all adjustment conditions with dx_clrsvcond( ), and reset if required. For example, if DTMF digit “1” has already been set to increase play speed by one step, a second call that attempts to redefine digit “1” to the origin will have no effect on speed or 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() or ATDX_BUFDIGS().
dx_addspddig( ) — set a DTMF digit to adjust speed

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_SVADJBLK
  Invalid number of play adjustment blocks

- EDX_SYSTEM
  Error from operating system; use dx_fileerrno( ) to obtain error value

Example

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

/*
 * Global Variables
 */

main()
{
    int  dxxxdev;

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

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

    /*
     * Continue Processing
     */
    .
    .
    .
    */
```
set a DTMF digit to adjust speed — dx_addspddig( )

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

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

See Also

- dx_addvoldig( )
- dx_adjsv( )
- dx_clrsvcond( )
- dx_getcursv( )
- dx_getsvmt( )
- dx_setsvcond( )
- dx_setsvmt( )
- speed and volume modification tables in the Voice API Programming Guide
- DX_SVMT data structure
- DX_SVCB data structure
dx_addtone( ) — add a user-defined tone

**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

**Platform:** DM3, Springware

---

### 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 <strong>dx_open( )</strong></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 DV_DIGIT digit buffer. These digits can be retrieved using <strong>dx_getdig( )</strong> (they can be used in the same way as DTMF digits, for example).</td>
</tr>
</tbody>
</table>

If you do not specify a digit, the tone will be indicated by a DE_TONEON event or DE_TONEOFF event.
add a user-defined tone — dx_addtone( )

Parameter | Description
-----------|----------------------------------
**digtype** | specifies the type of digit the channel will detect

On DM3 boards, the valid value is:
- DG_USER1

On Springware boards, valid values are:
- DG_USER1
- DG_USER2
- DG_USER3
- DG_USER4
- DG_USER5

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

**Note:** These types can be specified in addition to the digit types already defined for the voice library (DTMF, MF) which are specified using **dx_setdigtyp( )**.

### 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
dx_addtone( ) — add a user-defined tone

EDX_FREQDET
Invalid tone frequency

EDX_INVSUBCMD
Invalid sub-command

EDX_MAXTtmpl
Maximum number of user-defined tones for the board

EDX_SYSTEM
Error from operating system; use dx_fileerrno() to obtain error value

EDX_TONEID
Invalid tone template ID

Example

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

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

main()
{
  int  dxxxdev;

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

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

  /*
   * Bind the Tone to the Channel
   */
  if ( dx_addtone( dxxxdev, NULL, 0 ) == -1 ) {
    printf( "Unable to Bind the Tone \d\n", TID_1 );
    printf( "Lasterror = \d Err Msg = \s\n", ADV_LASTERR( dxxxdev ), ATRD_ERRMSGP( dxxxdev ));
    dx_close( dxxxdev );
    exit( 1 );
  }
}
```
add a user-defined tone — dx_addtone()

/*
 * 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_bldtdtcad( TID_2, 1000, 50, 500, 25, 20, 1, 100, 1, 3 ) == -1 ) {
    printf("Unable to build a Dual Tone Cadence Template\n");
}

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

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

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

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

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

/*
 * Continue Processing
 */
dx_addtone( ) — add a user-defined tone

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

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

See Also

- dx_blddt( ), dx_bldst( ), dx_blddtcad( ), dx_bldstcad( )
- dx_distone( )
- dx_enbtone( )
- global tone detection in the *Voice API Programming Guide*
- dx_getevt( )
- DX_CST data structure
- sr_getevtdatap( ) in the *Standard Runtime Library API Library Reference*
- dx_getdig( )
- dx_setdigtyp( )
- DV_DIGIT data structure
**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:** Speed and Volume

**Mode:** synchronous

**Platform:** DM3, Springware

---

**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 speed and volume control as well as speed and volume modification tables, see the Voice API Programming Guide. For information about speed and volume data structures, see the DX_SVMT and the DX_SVCB data structures.

**Parameter** | **Description**
--- | ---
chdev | specifies the valid channel device handle obtained when the channel was opened using `dx_open( )`
digit | specifies a DTMF digit (0-9, *, #) that will modify volume by the amount specified in `adjval`
dx_addvoldig() — set a DTMF digit to adjust volume

Parameter | Description
---|---
`adjval` | specifies a speed adjustment value to take effect whenever the digit specified in `digit` occurs

On DM3 boards, 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)

On Springware boards, the following are valid values:
- `SV_ADD2DB` – increase play volume by 2 dB
- `SV_ADD4DB` – increase play volume by 4 dB
- `SV_ADD6DB` – increase play volume by 6 dB
- `SV_ADD8DB` – increase play volume by 8 dB
- `SV_SUB2DB` – decrease play volume by 2 dB
- `SV_SUB4DB` – decrease play volume by 4 dB
- `SV_SUB6DB` – decrease play volume by 6 dB
- `SV_SUB8DB` – decrease play volume by 8 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.

- On DM3 boards, digits that are used for play adjustment may also be used as a terminating condition. If a digit is defined as both, then both actions are applied upon detection of that digit.

- On Springware boards, digits that are used for play adjustment will not be used as a terminating condition. If a digit is defined as both, then the play adjustment will take priority.

### 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; use dx_fileerrno() to obtain error value

Example

#include <stdio.h>
#include <srllib.h>
#include <dxclilib.h>
#include <windows.h>

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

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

    /*
    * Add a Speed Adjustment Condition - decrease the
    * playback volume by 2dB whenever DTMF key 2 is pressed.
    */
    if ( ( dxxxdev = dx_addvoldig( dxxxdev, '2', SV_SUB2DB ) ) == -1 ) {
        printf( "Unable to Add a Volume Adjustment Condition
" );
        printf( "Lasterror = %d  Err Msg = %s
",
                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_addspddig()
- dx_adjsv()
- dx_clrsvcond()
dx_addvoldig() — set a DTMF digit to adjust volume

- 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 (speed or volume)
- unsigned short action: how to adjust (absolute position, relative change, or toggle)
- unsigned short adjsize: adjustment size

Returns:
- 0 if successful
- -1 if failure

Includes: srllib.h
dxxxlib.h

Category: Speed and Volume
Mode: synchronous
Platform: DM3, Springware

Description

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

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

Notes:
1. This function is similar to dx_setsvcond(). Use dx_adjsv( ) to explicitly adjust the play immediately, and use dx_setsvcond( ) to adjust the play in response to specified conditions. See the description of dx_setsvcond( ) for more information.

2. Whenever a play is started, its speed and volume are 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</td>
</tr>
<tr>
<td></td>
<td>opened using dx_open( )</td>
</tr>
<tr>
<td>tabletype</td>
<td>specifies whether to modify the playback using a value from the speed or the</td>
</tr>
</tbody>
</table>
|               | volume modification table
|               | • SV_SPEEDTBL – use the speed modification table                            |
|               | • SV_VOLUMETBL – use the volume modification table                         |
dx_adjsv( ) — adjust speed or volume immediately

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

■ 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; use dx_fileerrno( ) to obtain error value
Example

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

main()
{
  int dxxxdev;

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

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

  /*
   * Continue Processing
   */

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

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

See Also

- `dx_setsvcond()`
- `dx_clrsvcond()`
- `dx_getcursv()`
- `dx_getsvmt()`
- speed and volume modification tables 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

**Platform:** DM3, Springware

---

**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(). 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.
- Do not use tone IDs 261, 262 and 263; they are reserved for library use.
- If you are using R2/MF tone detection, reserve the use of tone IDs 101 to 115 for the R2/MF tones. See r2_creatfsig() for further information.
- 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

For a list of error codes, see the Error Codes chapter.

■ Example

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

main()
{
  int dxxxdev;

  /*
   * Open the Voice Channel Device and Enable a Handler
   */
  if ( ( dxxxdev = dx_open( "dxxxB1C1", 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" );
  }
}
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( )
- r2_creatfsig( )
- r2_playbsig( )
**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:** srllib.h
dxxlib.h

**Category:** Global Tone Detection

**Mode:** synchronous

**Platform:** DM3, Springware

---

**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>
</tbody>
</table>
dy_blddtcad( ) — define a user-defined dual frequency cadenced tone

Parameter Description
ontime specifies the length of time for which the cadence is on (in 10 msec units)
ondev specifies the allowable deviation for on time (in 10 msec units)
offtime specifies the length of time for which the cadence is off (in 10 msec units)
offdev specifies the allowable deviation for off time (in 10 msec units)
repcnt specifies the number of repetitions for the cadence (that is, the number of times that an on/off signal is repeated)

Cautions

- Only one user-defined tone per process can be defined at any time. dy_blddtcad( ) will replace the previous user-defined tone definition.
- Do not use tone IDs 261, 262 and 263; they are reserved for library use.
- If you are using R2/MF tone detection, reserve the use of tone IDs 101 to 115 for the R2/MF tones. See r2_creatfsig( ) for further information.
- 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

```c
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
#define TID_2 102

main()
{
  int dxxxdev;

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

ontime specifies the length of time for which the cadence is on (in 10 msec units)
ondev specifies the allowable deviation for on time (in 10 msec units)
offtime specifies the length of time for which the cadence is off (in 10 msec units)
offdev specifies the allowable deviation for off time (in 10 msec units)
repcnt specifies the number of repetitions for the cadence (that is, the number of times that an on/off signal is repeated)
define a user-defined dual frequency cadenced tone — dx_blddtcad( )

/*----------------------------------------------------------------
 * 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_bldtt( )
- dx_bldstcad( )
- dx_addtone( )
- dx_distone( )
- dx_enbtone( )
- r2creatfsig( )
- r2_playbsig( )
**dx_bldst( ) — define a user-defined single-frequency tone**

**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
- **Platform:** DM3, Springware

## 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.
- Do not use tone IDs 261, 262 and 263; they are reserved for library use.
define a user-defined single-frequency tone — dx_bldst()

- If you are using R2/MF tone detection, reserve the use of tone IDs 101 to 115 for the R2/MF tones. See `r2_creatfsig()` for further information.
- 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**

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

#define TID_3   103

main()
{
    int dxxxdev;

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

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

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

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

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

**See Also**

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

- dx_blddt()
- dx_bldstcad()
- dx_addtone()
- dx_distone()
- dx_enbtone()
- r2_creatfsig()
- r2_playbsig()
define a user-defined single-frequency cadenced tone — dx_bldstcad( )

dx_bldstcad( )

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

Inputs:
- unsigned int tid: tone ID to assign
- unsigned int freq: frequency in Hz
- unsigned int fqdev: frequency deviation in Hz
- unsigned int ontime: tone on time in 10 msec
- unsigned int ontdev: on time deviation in 10 msec
- unsigned int offtime: tone off time in 10 msec
- unsigned int offtdev: 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
Platform: DM3, Springware

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>
dx_bldstcad( ) — define a user-defined single-frequency cadenced tone

### Cautions

- Only one tone per application may be defined at any time. **dx_bldstcad( )** will replace the previous user-defined tone definition.
- Do not use tone IDs 261, 262 and 263; they are reserved for library use.
- If you are using R2/MF tone detection, reserve the use of tone IDs 101 to 115 for the R2/MF tones. See the **r2_creatfsig( )** function for further information.
- 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

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

#define TID_4   104

main()
{
  int  dxxxdev;

  /*
  * Open the Voice Channel Device and Enable a Handler
  */
  if ( ( dxxxdev = dx_open( "dxxxB1C1", 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
  */
  .
  .
  .
  .
  */
```
/**
 * 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_bldttcad( )
- dx_blddt( )
- dx_bldst( )
- dx_addtone( )
- dx_distone( )
- dx_enbtone( )
- r2_creatfsig( )
- r2_playbsig( )
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

**Platform:** DM3, Springware

### 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>
#include <windows.h>

main()
{
    TN_GEN   tngen;
    int      dxxxdev;

    /*
    * Open the Voice Channel Device and Enable a Handler
    */
    if ( ( dxxxdev = dx_open( "dxxxB1C1", 0 ) ) == -1 ) {
        perror( "dxxxB1C1" );
        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*
- `r2_creatfsig()`
- `r2_playbsig()`
**dx_cacheprompt()**

**Name:** int dx_cacheprompt(brdhdl, iottp, prompthdl, mode)

**Inputs:**
- int brdhdl • valid physical board device handle
- DX_IOTT *iottp • pointer to I/O Transfer Table
- int prompthdl • pointer to return the cached prompt handle
- unsigned short mode • cached prompt mode

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

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

**Category:** Cached Prompt Management

**Mode:** asynchronous or synchronous

**Platform:** DM3

---

### Description

The **dx_cacheprompt()** function downloads voice data from multiple sources to the on-board memory. On successful completion the function returns a handle to the single cached prompt. This cached prompt handle can then be used in subsequent calls to a play function such as **dx_playioitdata()**.

For more information about cached prompt management and extended example code, see the *Voice API Programming Guide*.

**Parameter** | **Description**
---|---
brdhdl | specifies a valid physical board device handle (of the format brdBn) obtained by a call to **dx_open()**
iottp | points to the I/O Transfer Table structure, DX_IOTT, which specifies the location of voice data and the order in which data is downloaded. See **DX_IOTT**, on page 509, for information about this data structure.
prompthdl | points to an integer that represents the cached prompt handle
mode | specifies the mode in which the function will run. Valid values are:
  - EV_ASYNC – asynchronous mode
  - EV_SYNC – synchronous mode

---

### Cautions

- Before using **dx_cacheprompt()**, call **dx_getcachesize()** to determine the amount of on-board memory available for storing cached prompts.
- Closing the physical board device handle using **dx_close()** flushes the prompts from the on-board cache.
dx_cacheprompt( ) — download a cached prompt from multiple sources

- If the function is called in asynchronous mode (mode = EV_ASYNC), then the cached prompt handle returned should be used only after the TDX_CACHEPROMPT event is received.
- When iott parameter points to an array of DX_IOTT data structures (voice data being specified from multiple sources), the cached prompt handle that is returned refers to the beginning of the combined set of voice data that is downloaded. It is not possible to select an individual data item for playing from the cached prompt.
- WAVE files cannot be played from on-board cache memory.
- When dx_cacheprompt() is issued on a physical 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**

In asynchronous mode, the function returns immediately and a TDX_CACHEPROMPT event is queued upon completion. Check the extended attribute function ATDX_TERMMSK() for the termination reason. If a failure occurs, then a TDX_ERROR event will be queued. Use the Standard Runtime Library (SRL) Standard Attribute function ATDV_LASTERR() to determine the reason for error.

In synchronous mode, if this function returns -1 to indicate failure, call 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 <windows.h>
#include <stdio.h>
#include "srllib.h"
#include "dxxxlib.h"

int main()
{
    int brdhdl;         /* physical board device handle */
    int promptHandle;   /* Handle of the prompt to be downloaded */
    int fd1;            /* First file descriptor for file to be downloaded */
    int fd2;            /* Second file descriptor for file to be downloaded */
    DX_IOTT iott[2];    /* I/O transfer table to download cached prompt */

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

    /* Open first VOX file to cache */
    if ((fd1 = dx_fileopen("HELLO.VOX",O_RDONLY|O_BINARY)) == -1) {
        printf("File open error\n");
        exit(2);
    }

    /* Open second VOX file to cache */
    if ((fd2 = dx_fileopen("GREETING.VOX",O_RDONLY|O_BINARY)) == -1) {
        printf("File open error\n");
        exit(2);
    }

    return 0;
}
```
download a cached prompt from multiple sources — dx_cacheprompt()

```c
/* Set up DX_IOTT */
/*This block specifies the first data file */
iott[0].io_fhandle = fd1;
iott[0].io_offset = 0;
iott[0].io_length = -1;
iott[0].io_type = IO_DEV | IO_CONT;

/*This block specifies the second data file */
iott[1].io_fhandle = fd2;
iott[1].io_offset = 0;
iott[1].io_length = -1;
iott[1].io_type = IO_DEV | IO_EOT

/* Download the prompts to the on-board memory */
int promptHandle;
int result = dx_cacheprompt(brdhdl, iott, &promptHandle, EV_SYNC);
}

See Also

- dx_getcachesize()
- dx_open()
- dx_playiottdata()
- dx_setuio()
dx_chgdur( ) — change the duration definition for a tone

**dx_chgdur( )**

**Name:** int dx_chgdur(tonetype, ontime, ondev, offtime, offdev)

**Inputs:**
- int tonetype • tone to modify
- int ontime • on duration
- int ondev • ontime deviation
- int offtime • off duration
- int offdev • offtime deviation

**Returns:**
- 0 on success
- -1 if tone does not have cadence values
- -2 if unknown tone type

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

**Category:** Call Progress Analysis

**Mode:** synchronous

**Platform:** Springware

---

**Description**

The **dx_chgdur( )** function changes the standard duration definition for a call progress analysis tone, identified by **tonetype**. The voice driver comes with default definitions for each of the call progress analysis tones. The **dx_chgdur( )** function alters the standard definition of the duration component.

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

For more information on default tone templates as well as the call progress analysis feature, see the *Voice API Programming Guide*.
### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tonetype</td>
<td>specifies the identifier of the tone whose definition is to be modified. It may be one of the following:</td>
</tr>
<tr>
<td></td>
<td>- TID_BUSY1 – Busy signal</td>
</tr>
<tr>
<td></td>
<td>- TID_BUSY2 – Alternate busy signal</td>
</tr>
<tr>
<td></td>
<td>- TID_DIAL_INTL – International dial tone</td>
</tr>
<tr>
<td></td>
<td>- TID_DIAL_LCL – Local dial tone</td>
</tr>
<tr>
<td></td>
<td>- TID_DIAL_XTRA – Special (extra) dial tone</td>
</tr>
<tr>
<td></td>
<td>- TID_DISCONNECT – Disconnect tone (post-connect)</td>
</tr>
<tr>
<td></td>
<td>- TID_FAX1 – Fax or modem tone</td>
</tr>
<tr>
<td></td>
<td>- TID_FAX2 – Alternate fax or modem tone</td>
</tr>
<tr>
<td></td>
<td>- TID_RNGBK1 – Ringback (detected as single tone)</td>
</tr>
<tr>
<td></td>
<td>- TID_RNGBK2 – Ringback (detected as dual tone)</td>
</tr>
<tr>
<td>ontime</td>
<td>specifies the length of time that the tone is on (10 msec units)</td>
</tr>
<tr>
<td>ondev</td>
<td>specifies the maximum permissible deviation from ontime (10 msec units)</td>
</tr>
<tr>
<td>offtime</td>
<td>specifies the length of time that the tone is off (10 msec units)</td>
</tr>
<tr>
<td>offdev</td>
<td>specifies the maximum permissible deviation from offtime (10 msec units)</td>
</tr>
</tbody>
</table>

### Cautions

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

### Errors

For a list of error codes, see the Error Codes chapter.

### Example

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

main()
{
    DX_CAP  csp_c;
    int     ddd, car;
    char    *chnam, *dialstrg;
    chnam   = "dxxxB1C1";
    dialstrg = "L1234";
    
    /*
    * Open channel
    */
    if ((ddd = dx_open( chnam, NULL )) == -1 ) {
        /* handle error */
    }
}```
dx_chgdur( ) — change the duration definition for a tone

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

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

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

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

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

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

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

switch( car ) {
    case CR_NODIALTONE:
        printf("Unable to get dial tone\n");
        break;
    case CR_BUSY:
        printf("%s engaged\n", dialstrg );
        break;
    case CR_CNCT:
        printf("Successful connection to %s\n", dialstrg );
        break;
    default:
        break;
}
change the duration definition for a tone — dx_chgdur()

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

See Also

- dx_chgfreq()
- dx_chgrepcnt()
- dx_deltones()
- dx_initcallp()
dx_chgfreq( ) — change the frequency definition for a tone

**dx_chgfreq( )**

**Name:** int dx_chgfreq(tonetype, freq1, freq1dev, freq2, freq2dev)

**Inputs:**
- int tonetype  • tone to modify
- int freq1    • frequency of first tone
- int freq1dev • frequency deviation for first tone
- int freq2    • frequency of second tone
- int freq2dev • frequency deviation of second tone

**Returns:**
- 0 on success
- -1 on failure due to bad parameter(s) for tone type
- -2 on failure due to unknown tone type

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

**Category:** Call Progress Analysis

**Mode:** synchronous

**Platform:** Springware

---

**Description**

The **dx_chgfreq( )** function changes the standard frequency definition for a call progress analysis tone, identified by **tonetype**. The voice driver comes with default definitions for each of the call progress analysis tones. The **dx_chgfreq( )** function alters the standard definition of the frequency component.

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

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

For more information on default tone templates as well as the call progress analysis feature, see the **Voice API Programming Guide**.
### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| **tonetype** | specifies the identifier of the tone whose definition is to be modified. It may be one of the following:  
  - TID_BUSY1 – Busy signal  
  - TID_BUSY2 – Alternate busy signal  
  - TID_DIAL_INTL – International dial tone  
  - TID_DIAL_LCL – Local dial tone  
  - TID_DIAL_XTRA – Special (extra) dial tone  
  - TID_DISCONNECT – Disconnect tone (post-connect)  
  - TID_FAX1 – Fax or modem tone  
  - TID_FAX2 – Alternate fax or modem tone  
  - TID_RNGBK1 – Ringback (detected as single tone)  
  - TID_RNGBK2 – Ringback (detected as dual tone) |
| **freq1** | specifies the frequency of the first tone (in Hz) |
| **freq1dev** | specifies the maximum permissible deviation (in Hz) from **freq1** |
| **freq2** | specifies the frequency of the second tone, if any (in Hz). If there is only one frequency, **freq2** is set to 0. |
| **freq2dev** | specifies the maximum permissible deviation (in Hz) from **freq2** |

### Cautions

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

### Errors

For a list of error codes, see the Error Codes chapter.

### Example

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

main()
{
  DX_CAP  cap_s;
  int     ddd, car;
  char    *chnam, *dialstrg;
  chnam   = "dxxxB1C1";
  dialstrg = "L1234";

  /*
   * Open channel
   */
  if ((ddd = dx_open( chnam, NULL )) == -1 ) { /* handle error */
  }
```
/*
 *  Delete any previous tones
 * /
if (dx_deltones(ddd) < 0) {
    /* handle error */
}

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

/*
 *  Change Enhanced call progress default busy cadence
 * /
if (dx_chgdur(TID_BUSY1, 550, 400, 550, 400) < 0) {
    /* handle error */
}
if (dx_chgrep(TID_BUSY1, 4) < 0) {
    /* handle error */
}

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

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

/*
 *  Dial
 * /
if ((car = dx_dial(ddd, dialstrg,(DX_CAP *)&cap_s, DX_CALLP|EV_SYNC))==-1) {
    /* handle error */
}
switch(car) {
    case CR_NODIALTONE:
        printf(" Unable to get dial tone\n");
        break;
    case CR_BUSY:
        printf(" %s engaged\n", dialstrg);
        break;
    case CR_CNCT:
        printf(" Successful connection to %s\n", dialstrg);
        break;
    default:
        break;
}
/*
 * Set on Hook
 *
 if ((dx_sethook( ddd, DX_ONHOOK, EV_SYNC )) == -1) {
     /* handle error */
 }

dx_close( ddd );
}

• dx_chgdur( )
• dx_chgrepcnt( )
• dx_deltones( )
• dx_initcallp( )
dx_chgrepcnt( ) — change the repetition definition for a tone

**dx_chgrepcnt( )**

**Name:** int dx_chgrepcnt(tonetype, repcnt)

**Inputs:**
- int tonetype  • tone to modify
- int repcnt  • repetition count

**Returns:**
- 0 if success
- -1 if tone does not have a repetition value
- 2 if unknown tone type

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

**Category:** Call Progress Analysis

**Mode:** synchronous

**Platform:** Springware

---

**Description**

The `dx_chgrepcnt( )` function changes the standard repetition definition for a call progress analysis tone, identified by `tonetype`. The repetition count component refers to the number of times that the signal must repeat before being recognized as valid. The voice driver comes with default definitions for each of the call progress analysis tones. The `dx_chgrepcnt( )` function alters the standard definition of the repetition count component.

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.

Changing a tone definition has no immediate effect on the behavior of an application. The `dx_initcallp( )` function takes the tone definitions and uses them to initialize a channel. Once a channel is initialized, subsequent changes to the tone definitions have no effect on that channel. For these changes to take effect, you must first call `dx_deltones( )` followed by `dx_initcallp( )`. 
change the repetition definition for a tone — dx_chgrepent()

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| tonetype  | specifies the identifier of the tone whose definition is to be modified. It may be one of the following:  
  • TID_BUSY1 – Busy signal  
  • TID_BUSY2 – Alternate busy signal  
  • TID_DIAL_INTL – International dial tone  
  • TID_DIAL_LCL – Local dial tone  
  • TID_DIAL_XTRA – Special (extra) dial tone  
  • TID_DISCONNECT – Disconnect tone (post-connect)  
  • TID_FAX1 – Fax or modem tone  
  • TID_FAX2 – Alternate fax or modem tone  
  • TID_RINGBK1 – Ringback (detected as single tone)  
  • TID_RINGBK2 – Ringback (detected as dual tone) |
| repcnt    | the number of times that the signal must repeat |

## Cautions

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

## Errors

For a list of error codes, see the Error Codes chapter.

## Example

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

main()
{
  DX_CAP cap_s;
  int     ddd, car;
  char    *chnam, *dialstrg;
  chnam   = "dxxxB1C1";
  dialstrg = "L1234";

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

  /*
  * Delete any previous tones
  */
  if ( dx_deltones(ddd) < 0 ) {
    /* handle error */
  }
```
dx_chgrepcnt( ) — change the repetition definition for a tone

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

/*
 * Change Enhanced call progress default busy cadence
 */
if (dx_chgdur( TID_BUSY1, 550, 400, 550, 400 ) < 0) {
    /* handle error */
}
if (dx_chgrepcnt( TID_BUSY1, 4 ) < 0) {
    /* handle error */
}

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

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

/*
 * Dial
 */
if ((car = dx_dial( ddd, dialstrg,(DX_CAP *)&cap_s, DX_CALLP|EV_SYNC))==-1) {
    /* handle error */
}
switch( car ) {
    case CR_NODIALTONE:
        printf(" Unable to get dial tone\n");
        break;
    case CR_BUSY:
        printf(" %s engaged\n", dialstrg );
        break;
    case CR_CNCT:
        printf(" Successful connection to %s\n", dialstrg );
        break;
    default:
        break;
}
    /*
    * Set on Hook
    */
    if ((dx_sethook( ddd, DX_ONHOOK, EV_SYNC )) == -1) {
        /* handle error */
    }
    dx_close( ddd );
}

See Also

- dx_chgdur()
change the repetition definition for a tone — \textit{dx\_chgrepcnt()} 

- \textit{dx\_chgfreq( )}
- \textit{dx\_deltones( )}
- \textit{dx\_initcallp( )}
dx_close() — close a channel or board device handle

The `dx_close()` function closes a channel device handle or board device handle that was previously opened using `dx_open()`. On DM3 boards, the `dx_close()` function also closes a physical board device 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.

### Description

The `dx_close()` function closes a channel device handle or board device handle that was previously opened using `dx_open()`. On DM3 boards, the `dx_close()` function also closes a physical board device 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>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>

### 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 Windows `close()` function to close a voice device; unpredictable results will occur.
- The `dx_close()` function discards any outstanding events on that handle.
- On DM3 boards, if you close a device via `dx_close()` after modifying speed and volume table values using `dx_setsvmt()`, the `dx_getcursv()` function may return incorrect speed and volume settings for the device. This is because the next `dx_open()` resets the speed and volume tables to their default values.
close a channel or board device handle — dx_close()

■ Errors

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.

■ Example 1

This example illustrates how to close a channel device handle.

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

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

■ Example 2

This example illustrates how to close a physical board device handle when using cached prompts.

```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);
  }
  .
  .
  dx_close(brdhdl);
}
```
`dx_close( ) — close a channel or board device handle`

- **See Also**
  - `dx_open( )`
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

**Platform:** DM3

### 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.
dx_CloseStreamBuffer( ) — delete a circular stream buffer

Example

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

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

    if ((hBuffer = dx_OpenStreamBuffer(nBuffSize)) < 0)
        { printf("Error opening stream buffer \n"); exit(1); }
    if (dx_PutStreamData(hBuffer, pData, 1024, STREAM_CONT) < 0)
        { printf("Error in dx_PutStreamData \n"); exit(2); }

    if ((vDev = dx_open("dxxxB1C1", 0)) < 0)
        { printf("Error opening voice device\n"); exit(3); }

    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_playiottdata(vDev, &iott, NULL, EV_SYNC) < 0)
        { printf("Error in dx_play() \n", ATDV_LASTERR(vDev)); }
    if (dx_close(vDev) < 0)
        { printf("Error closing voice device\n"); }
    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: srlib.h 
dxxxlib.h
Category: Structure Clearance
Mode: synchronous
Platform: DM3, Springware

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.

Parameter Description

<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 496.</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 <srlib.h>
#include <dxxxlib.h>
#include <windows.h>

define 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); 
} 
```
dx_clrcap() — clear all fields in a DX_CAP structure

```c
    cap.ca_nbrdna = 5;     /* 5 rings before no answer */
    /* continue with call progress analysis */
    };
```

- **See Also**

  - `dx_dial()`
  - DX_CAP data structure
  - call progress analysis topic in the *Voice API Programming Guide*
The **dx_clrdigbuf( )** function clears all digits in the firmware digit buffer of the channel specified by `chdev`.

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>chdev</code></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; use `dx_fileerrno( )` to obtain error value

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

---

**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
- **Platform:** DM3, Springware

---

**Description**

The `dx_clrdigbuf( )` function clears all digits in the firmware digit buffer of the channel specified by `chdev`. The function will fail and return -1 if the channel device handle is invalid or the channel is busy.

**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; use `dx_fileerrno( )` to obtain error value

**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>
#include <windows.h>
```
dx_clrdigbuf() — clear all digits in the firmware digit buffer

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( )

**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:** Speed and Volume

**Mode:** synchronous

**Platform:** DM3, Springware

---

**Description**

The `dx_clrsvcond( )` function clears all speed or volume adjustment conditions that have been previously set using `dx_setsvcond( )` or the convenience functions `dx_addspddig( )` and `dx_addvoldig( )`.

Before resetting an adjustment condition, you must first clear all current conditions by using this function, and then reset conditions using `dx_setsvcond( )`, `dx_addspddig( )`, 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; use `dx_fileerrno( )` to obtain error value
Example

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

main()
{
    int dxxxdev;

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

    /*
     * Clear all Speed and Volume Conditions
     */
    if ( dx_clrsvcond(dxxxdev) == -1 ) {
        printf("Unable to Clear the Speed/Volume");
        printf("Conditions\n", 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_addspddig()`
- `dx_addvoldig()`
- Speed and volume modification tables in Voice API Programming Guide
- DX_SVCB data structure
dx_clrtpt( )

Name: int dx_clrtpt(tpt, size)

Inputs: DV_TPT *tpt • pointer to Termination Parameter Table structure
    int size • number of entries to clear

Returns: 0 if success
         -1 if failure

Includes: srllib.h
dxxlib.h

Category: Structure Clearance

Mode: synchronous

Platform: DM3, Springware

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.

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tpt</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 485.</td>
</tr>
</tbody>
</table>

Notes: 1. The DV_TPT is defined in srllib.h rather than dxxlib.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.
dx_clrtpt() — clear all fields in a DV_TPT structure

- **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>
#include <windows.h>

main()
{
    DV_TPT tpt1[2];
    DV_TPT tpt2[2];

    /* Set up the links in the DV_TPTs */
    tpt1[0].tp_type = IO_CONT;
    tpt1[1].tp_type = IO_LINK;
    tpt1[1].tp_nextp = &tpt2[0];
    tpt2[0].tp_type = IO_CONT;
    tpt2[1].tp_type = IO_EOT;
    /* set up the other DV_TPT fields as required for termination */
    ...
    /* play a voice file, get digits, etc. */
    ...
    /* clear out the DV_TPT structures if required */
    dx_clrtpt(&tpt1[0], 4);
    /* now set up the DV_TPT structures for the next play */
    ...
}
```

- **See Also**

- DV_TPT data structure
dx_createtone( )

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

**Inputs:**
- int brdhdl • a valid physical 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
dxxlib.h

**Category:** Call Progress Analysis

**Mode:** Asynchronous or synchronous

**Platform:** DM3

---

**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.
dx_createtone( ) — create a new tone definition for a specific call progress tone

Parameter | Description
---|---
**brdhdl** | specifies a valid physical board device handle (of the format `brdBn`) obtained by a call to `dx_open( )`
**toneid** | specifies the tone ID of the call progress tone whose definition needs to be created. Valid values are:
- TID_DIAL_LCL
- TID_DIAL_INTL
- TID_BUSY1
- TID_RNGBK1
- TID_BUSY2
- TID_RNGBK2
- TID_DISCONNECT
- TID_FAX1
- TID_FAX2
- TID_SIT_NC (no circuit found)
- TID_SIT_IC (operator intercept)
- TID_SIT_VC (vacant circuit)
- TID_SIT_RO (reorder)
**tonedata** | specifies a pointer to the TONE_DATA data structure which contains the tone information to be created for the call progress tone identified by `toneid`
**mode** | specifies how the function should be executed, either EV_ASYNC (asynchronous) or EV_SYNC (synchronous)

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 physical 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.
create a new tone definition for a specific call progress tone — dx_createtone( )

## 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 */

    /* 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_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_max = 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 */
} 
```
dx_createtone( ) — create a new tone definition for a specific call progress tone

```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( )**

**Name:** int dx_deletetone(brdhdl, toneid, mode)

**Inputs:**
- int brdhdl • a valid physical 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

**Platform:** DM3

---

**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 <strong>physical board device handle</strong> (not a virtual board device) of the format <strong>brdBn</strong> obtained by a call to <code>dx_open( )</code></td>
</tr>
</tbody>
</table>
dx_deletetone() — delete a specific call progress tone

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

- **EDX_BADPARM**
  - invalid parameter
- **EDX_SYSTEM**
  - error from operating system
- **EDX_TONEID**
  - bad tone template ID

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

| mode | specifies how the function should be executed, either EV_ASYNC (asynchronous) or EV_SYNC (synchronous) |

Parameter Description
delete a specific call progress tone — dx_deletetone( )

- **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);
    }

    /* 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);
    }
}
```

- **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

**Platform:** DM3, Springware

---

### 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_bldt()` , `dx_bldst()` , `dx_bldstcad()` , or `dx_bldtdtcad()` .

<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; use `dx_fileerrno()` to obtain error value
delete all user-defined tones — dx_deltones( )

## Example

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

main()
{
  int  dxxxdev;

  /*
   * Open the Voice Channel Device and Enable a Handler
   */
  if ( ( dxxxdev = dx_open( "dxxxB1C1", 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_bldttcad( )`
dx_dial( ) — dial an ASCIIZ string

**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

**Platform:** DM3, Springware

---

**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*. For DM3 boards, 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 <strong>dx_open( )</strong></td>
</tr>
<tr>
<td>dialstrp</td>
<td>points to the ASCII dial string. <strong>dialstrp</strong> must contain a null-terminated string of ASCII characters. For a list of valid dialing and control characters, see Table 2 and Table 3.</td>
</tr>
</tbody>
</table>
dial an ASCII string — dx_dial()

<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>
<tr>
<td>mode</td>
<td>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) On Springware boards, to enable call progress analysis (PerfectCall), you must call dx_initcallp() prior to calling dx_dial(). Otherwise, dx_dial() uses basic call progress analysis. 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. On DM3 boards, call progress analysis (PerfectCall) is enabled directly through dx_dial(). The dx_initcallp() function is not supported.</td>
</tr>
</tbody>
</table>

### Valid Dial String Characters

On DM3 boards, the following is a list of valid dialing and control characters.

#### Table 2. Valid Dial String Characters (DM3)

<table>
<thead>
<tr>
<th>Characters</th>
<th>Description</th>
<th>DTMF</th>
<th>MF</th>
</tr>
</thead>
<tbody>
<tr>
<td>On Keypad</td>
<td></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>
<td>Yes</td>
</tr>
<tr>
<td>*</td>
<td>asterisk or star</td>
<td>Yes</td>
<td>Yes (KP)</td>
</tr>
<tr>
<td>#</td>
<td>pound, hash, number, or octothorpe</td>
<td>Yes</td>
<td>Yes (ST)</td>
</tr>
<tr>
<td>Not on Keypad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>Yes</td>
<td>Yes (ST1)</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>Yes</td>
<td>Yes (ST2)</td>
</tr>
<tr>
<td>c</td>
<td></td>
<td>Yes</td>
<td>Yes (ST3)</td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Special Control</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>pause for 2.5 seconds (comma)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>T</td>
<td>Dial Mode: Tone (DTMF) (default)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>M</td>
<td>Dial Mode: MF</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

When using dx_dial() on DM3 boards, 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, it becomes the new default and that dialing mode remains in effect for all dialing until a new dialing mode is specified or the system is restarted. For this reason, we recommend that you always put “T” in the dialing string for DTMF tone dialing after using the M (MF) dial mode. The `dx_close( )` and `dx_open( )` do not reset the default dialing mode to DTMF tone dialing.
• 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”.

On Springware boards, the following is a list of valid dialing and control characters.

### Table 3. Valid Dial String Characters (Springware)

<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>DTMF</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 (comma)</td>
<td>Yes</td>
</tr>
<tr>
<td>&amp;</td>
<td>flash (ampersand)</td>
<td>Yes</td>
</tr>
<tr>
<td>T</td>
<td>Dial Mode: Tone (DTMF) (default)</td>
<td>Yes</td>
</tr>
<tr>
<td>P</td>
<td>Dial Mode: Pulse</td>
<td>Yes</td>
</tr>
<tr>
<td>M</td>
<td>Dial Mode: MF</td>
<td>Yes</td>
</tr>
<tr>
<td>L</td>
<td>call progress analysis: local dial tone</td>
<td>Yes</td>
</tr>
<tr>
<td>I</td>
<td>call progress analysis: international dial tone</td>
<td>Yes</td>
</tr>
<tr>
<td>X</td>
<td>call progress analysis: special dial tone</td>
<td>Yes</td>
</tr>
</tbody>
</table>

When using `dx_dial( )` on Springware boards, 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 P, M, or T control characters, it becomes the new default and that dialing mode remains in effect for all dialing until a new dialing mode is specified or the system is restarted. For this reason, we recommend that you always put “T” in the dialing string for DTMF tone dialing after using the P (pulse) or M (MF) dial modes. The `dx_close()` and `dx_open()` do not reset the default dialing mode to DTMF tone dialing.

- Intel® TDM bus boards do not support pulse digit dialing using `dx_dial()`.
- The L, I, and X control characters function only when dialing with PerfectCall call progress analysis.
- MF dialing is only available on systems with MF capability.
- The pause character “,” and the flash character “&” are not available in MF dialing mode. To send these characters with a string of MF digits, switch to DTMF or pulse mode before sending “,” or “&”, and then switch back to MF mode by sending an “M”. For example: M*1234T,M5678a
- Dialing parameter default values can be set or retrieved using `dx_getparm()` and `dx_setparm()`; see the 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”.

**Asynchronous Operation**

Set the `mode` field to EV_ASYNC, using a bitwise OR. When running asynchronously, the function will return 0 to indicate it has initiated successfully, and will generate one of the following termination events to indicate completion:

- TDX_CALLP
  - termination of dialing (with call progress analysis)
- TDX_DIAL
  - termination of dialing (without call progress analysis)

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 will return a 0 to indicate that it has completed successfully.

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

**Call Progress Analysis**

Call progress analysis provides information about the call. If it is enabled using the `mode` parameter, it runs on the call after dialing completes. The function can be set to run using default
call progress analysis parameters, or by using the call progress analysis parameter structure
(DX_CAP).

Call progress analysis results can be retrieved using ATDX_CPTERM(). If dx_dial() is running
synchronously, the call progress analysis results will also be returned by the dx_dial() function.

Possible call progress analysis termination reasons are:

- CR_BUSY
  - line was busy
- CR_CEPT
  - operator intercept
- CR_CNCT
  - call connected
- CR_ERROR
  - call progress analysis error
- CR_FAXTONE
  - fax machine or modem
- CR_NOANS
  - no answer
- CR_NODIALTONE
  - no dial tone
- CR_NORB
  - no ringback
- CR_STOPD
  - call progress analysis stopped due to dx_stopch() 

On DM3 boards, if call progress analysis is enabled, additional information about the call can be
obtained using the following extended attribute functions:

- 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

**Note:** On DM3 boards, the extended attribute functions that provide call progress analysis information do
not return information related to functionality that is not supported; for example,
ATDX_CONNTYPE() connection type CON_LPC and ATDX_CPTERM() termination reason
CR_NODIALTONE.

On Springware boards, if call progress analysis is enabled, additional information about the call
can be obtained using the following extended attribute functions:

- ATDX_ANSRSIZ() 
  - Returns duration of answer
ATDX_CONNTYPE()       
Returns the connection type for a completed call

ATDX_CPERERROR()       
Returns call progress analysis error

ATDX_CPTERM()          
Returns last call progress analysis termination

ATDX_CRTNID()          
Returns tone identifier

ATDX_DTNFAIL()         
Returns dial tone fail character

ATDX_FRQDUR()          
Returns duration of first frequency detected

ATDX_FRQDUR2()         
Returns duration of second frequency detected

ATDX_FRQDUR3()         
Returns duration of third frequency detected

ATDX_FRQHZ()           
Returns frequency of first detected tone in Hz

ATDX_FRQHZ2()          
Returns frequency of second detected tone in Hz

ATDX_FRQHZ3()          
Returns frequency of third detected tone in Hz

ATDX_FRQOUT()          
Returns percent of frequency out of bounds

ATDX_LONGLOW()         
Returns duration of longer silence

ATDX_SHORTLOW()        
Returns duration of shorter silence

ATDX_SIZEHI()          
Returns duration of non-silence

**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 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.
dx_dial() — dial an ASCIIZ string

- 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; use dx_fileerrno() to obtain error value

**Example**

This example demonstrates how to use dx_dial() and call progress analysis (synchronous mode) on Springware boards. On DM3 boards, dx_dial() supports call progress analysis directly; you do not use dx_initcallp() to initialize call progress analysis.

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

main()
{
    DX_CAP   cap_s;
    int      ddd, car;
    char     *chnam, *dialstrg;
    chnam    = "dxxxB1C1";
    dialstrg = "L1234";

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

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

    /*
     * Change call progress analysis default local dial tone
     */
    if (dx_chgfreq(TID_DIAL_LCL, 425, 150, 0, 0) < 0) {
        /* handle error */
    }
}```
dial an ASCIIZ string — `dx_dial()`

```c
/*
 * Change call progress analysis default busy cadence
 */
if (dx_chgdur( TID_BUSY1, 550, 400, 550, 400 ) < 0) {
    /* handle error */
}
if (dx_chgrepcent( 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\n", dialstrg );
        break;
    case CR_CNCT:
        printf(" Successful connection to %s\n", dialstrg );
        break;
    default:
        break;
}

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

dx_close( ddd );
```

**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_ANSRSIZ()
- ATDX_CPERRORE( )
- ATDX_FRQDUR()
- ATDX_FRQDUR2()
- ATDX_FRQDUR3()
- ATDX_FRQHZ()
- ATDX_FRQHZ2()
- ATDX_FRQHZ3()
- ATDX_FRQOUT()
- ATDX_LONGLOW()
- ATDX_SHORTLOW()
- ATDX_SIZEHI()
**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:**
- srllib.h
- dxxxlib.h

**Category:** Global Tone Detection

**Mode:** synchronous

**Platform:** DM3, Springware

---

**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. To disable detection of all user-defined tones on the channel, set <code>toneid</code> to <code>TONEALL</code>.</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.
**dx_distone( ) — disable detection of a user-defined 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_BADPROD**
  - Function not supported on this board

- **EDX_SYSTEM**
  - Error from operating system; use dx_fileerrno( ) to obtain error value

- **EDX_TNMSGSTATUS**
  - Invalid message status setting

- **EDX_TONEID**
  - Bad tone ID

### Example

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

main()
{
    int dxxxdev;

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

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

    /*
     * Bind the Tone to the Channel
     */
    if ( dx_addtone( dxxxdev, NULL, 0 ) == -1 ) {
        printf( "Unable to Bind the Tone \d\n", TID_1 );
        printf( "Lasterror = %d  Err Msg = %s\n",
                ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
        dx_close( dxxxdev );
        exit( 1 );
    }
}
```
/*
 * 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_bldttcad( ), dx_bldstcad( )
- dx_enbtone( )
- global tone detection topic in the Voice API Programming Guide
- dx_getev( )
- DX_CST data structure
- sr_getevtdatap( ) in the Standard Runtime Library API Library Reference
dx_enbtone( ) — enable detection of a user-defined tone

dx_enbtone( )

**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

**Platform:** DM3, Springware

---

### 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</td>
</tr>
<tr>
<td></td>
<td>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>
<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 will remain in effect until <code>dx_enbtone()</code> or <code>dx_distone()</code> is called again to reset it.</td>
</tr>
</tbody>
</table>
enable detection of a user-defined tone — **dx_enbtone()**

### 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; use **dx_fileerrno()** to obtain error value
- **EDX_TONEID**
  - Bad tone ID
- **EDX_TNMSGSTATUS**
  - Invalid message status setting

### Example

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

main()
{
    int  dxxxdev;

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

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

    /*
     * Bind the Tone to the Channel
     */
    if ( dx_addtone( dxxxdev, NULL, 0 ) == -1 ) {
        printf( "Unable to Bind the Tone %d\n", TID_1 );
    }
```

---

*Voice API for Windows Operating Systems Library Reference — November 2003*
dx_enbtone( ) — enable detection of a user-defined tone

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

/*
 * Enable Detection of ToneId TID_1
 */
if ( dx_enbtone( dxxxdev, TID_1, DM_TONEON | DM_TONEOFF ) == -1 ) {
    printf( "Unable to Enable Detection of Tone %d\n", TID_1 );
    printf( "Lasterror = %d  Err Msg = %s\n",  
        ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
dx_close( dxxxdev );
exit( 1 );
}

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

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

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

**See Also**

- dx_addtone( )
- dx_blddt( ), dx_bldst( ), dx_blddtcad( ), dx_bldstcad( )
- dx_distone( )
- global tone detection in *Voice API Programming Guide*
- dx_getevt( )
- DX_CST data structure
- sr_getevtdatap( ) 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
dxxxlil.h
- **Category:** File Manipulation
- **Mode:** synchronous
- **Platform:** DM3, Springware

### Description

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; use `dx_fileerrno( )` to obtain the system error value. Refer to the `dx_fileerrno( )` function for a list of the possible system error values.

### Example

```c
/*
 * Play a voice file. Terminate on receiving 4 digits or at end of file
 */
#include <fcntl.h>
#include <srllib.h>
#include <dxxxlil.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_fileseek()
- dx_fileread()
- dx_filewrite()
return the system error value — _dx_fileerrno( )

_dx_fileerrno( )

**Name:** int dx_fileerrno(void)

**Inputs:** none

**Returns:** system error value

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

**Category:** File Manipulation

**Mode:** synchronous

**Platform:** DM3, Springware

---

**Description**

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 4 lists the system error values that may be returned by dx_fileerrno( ).

**Table 4. 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. This error signifies that an attempt was made to access a file (or, in some cases, a directory) in a way that is incompatible with the file’s attributes. For example, the error can occur when an attempt is made to read from a file that is not open, to open an existing read-only file for writing, or to open a directory instead of a file. The error can also occur in an attempt to rename a file or directory or to remove an existing directory.</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>
dx_fileerrno( ) — return the system error value

Table 4. 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
- **Platform:** DM3, Springware

## Description

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_recioffdata( )** 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; use **dx_fileerrno( )** to obtain the system error value. Refer to the **dx_fileerrno( )** function for a list of the possible system error values.

## 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_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
  - handle returned from **dx_fileopen( )**
- void *buffer
  - storage location for data
- unsigned int count
  - maximum number of bytes

**Returns:**
- number of bytes if success
- -1 if failure

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

**Category:** File Manipulation

**Mode:** synchronous

**Platform:** DM3, Springware

---

**Description**

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; use **dx_fileerrno( )** to obtain the system error value. Refer to the **dx_fileerrno( )** function for a list of the possible system error values.

**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;
dx_fileread( ) — read data from a file

/*This block uses my I/O functions */
iotpp++;
iott->io_type = IO_DEV|IO_UIO|IO_CONT
iott->io_fhandle = vodat_fd;
iott->io_offset = 20001;
iott->io_length = 20000;

/*This block uses standard I/O functions */
iotpp++
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;
dehandle = dx_open("dxxxB1C1", 0);
dx_sethook(devhandle, DX-ONHOOK, EV_SYNC)
dx_wstring(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_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

**Platform:** DM3, Springware

---

**Description**

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; use dx_fileerrno( ) to obtain the system error value. Refer to the dx_fileerrno( ) function for a list of the possible system error values.

**Example**

```c
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
```
dx_fileseek() — move a file pointer

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 */
iott++;
iott->io_type = IO_DEV|IO_UIO|IO_CONT
iott->io_fhandle = vodat_fd;
iott->io_offset = 20001;
iott->io_length = 20000;

/This block uses standard I/O functions */
iott++
iott->io_type = IO_DEV|IO_CONT
iott->io_fhandle = vodat_fd;
iott->io_offset = 20002;
iott->io_length = 20000;

/This block uses my I/O functions */
iott->io_type = IO_DEV|IO_UIO|IO_EOT
iott->io_fhandle = vodat_fd;
iott->io_offset = 10003;
iott->io_length = 20000;
devhandle = dx_open("dxxxB1C1", NULL);
dx_sethook(devhandle, DX_ONHOOK,EV_SYNC)
dx_wtring(devhandle,1,DX_OFFHOOK,EV_SYNC);
dx_clrdigbuf;
if(dx_rec(devhandle,iott,(DX_TPT*)NULL,RM_TONE|EV_SYNC) == -1) {
    perror("*");
    exit(1);
}
dx_clrdigbuf(devhandle);
if(dx_play(devhandle,iott,(DX_TPT*)EV_SYNC) == -1 {
    perror("*");
    exit(1);
}
dx_close(devhandle);

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
  - void *buffer
  - unsigned int count
- **Returns:**
  - number of bytes if success
  - -1 if failure
- **Includes:** srllib.h
dxxxlib.h
- **Category:** File Manipulation
- **Mode:** synchronous
- **Platform:** DM3, Springware

---

### Description

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; use `dx_fileerrno( )` to obtain the system error value. Refer to the `dx_fileerrno( )` function for a list of the possible system error values.

### 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 */
```
```c
/*
 * 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_filewrite( ) — write data from a buffer into a file

/* This block uses standard I/O functions */
iott++;
iott->io_type = IO_DEV|IO_CONT
iott->io_fhandle = vodat_fd;
iott->io_offset = 20002;
iott->io_length = 20000;

/* This block uses my I/O functions */
iott->io_type = IO_DEV|IO_UIO|IO_EOT
iott->io_fhandle = vodat_fd;
iott->io_offset = 10003;
iott->io_length = 20000;
devhandle = dx_open("dxxxB1C1", NULL);
dx_sethook(devhandle, DX-ONHOOK, EV_SYNC)
dx_wtring(devhandle,1,DX_OFFHOOK, EV_SYNC);
dx_clrdigbuf;
    if(dx_rec(devhandle, iott, (DX_TPT*)NULL,RM_TONE|EV_SYNC) == -1) {
        perror("*");
        exit(1);
    }
dx_clrdigbuf(devhandle);
    if(dx_play(devhandle, iott, (DX_TPT*)EV_SYNC) == -1 {
        perror("*");
        exit(1);
    }
dx_close(devhandle);

See Also

- dx_fileopen()
- dx_fileclose()
- dx_fileseek()
- dx_fileread()
**dx_getcachesize( )**

**Name:** int dx_getcachesize(brdhdl, cachesize, flag)

**Inputs:**
- int brdhdl  • valid physical board device handle
- int *cachesize  • pointer to current cache size
- unsigned short flag  • flag for type of cache size

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

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

**Category:** Cached Prompt Management

**Mode:** synchronous

**Platform:** DM3

---

**Description**

The `dx_getcachesize( )` function returns the size of the on-board memory used to store cached prompts for a board specified by the board handle.

If the flag specified is DX_CACHETOTAL, the function returns the total size of the memory available for the board. If the flag specified is DX_CACHEREMAINING, the function returns the remaining size of the cache that can be used to store additional prompts.

For more information about Cached Prompt Management and extended example code, see the *Voice API Programming Guide*.

**Parameter** | **Description**
--- | ---
brdhdl | specifies a valid physical board device handle (of the format brdBn) obtained by a call to `dx_open( )`
cachesize | points to an integer that represents the current cache size in bytes
flag | flag that indicates the type of cache size. Valid values are:
- DX_CACHETOTAL – total size of memory available on the board
- DX_CACHEREMAINING – remaining size of cache that can be used to store additional prompts

---

**Cautions**

None.

---

**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
dx_getcachesize( ) — get size of on-board memory for cached prompts

obtain a descriptive error message. For a list of error codes returned by ATDV_LASTERR( ), see the Error Codes chapter.

■ Example

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

main()
{
    int brdhdl;         /* board handle */
    int cachetotal;     /* Total size of the on-board memory for storing cache prompts */
    int cacheremaining; /* Remaining size of on-board memory */
    .
    .
    .
    /* Open board */
    if ((brdhdl = dx_open("brdB1",0)) == -1) {
        printf("Cannot open board\n");
        /* Perform system error processing */
        exit(1);
    }
    /* Find the total available size of the on-board memory */
    if (!dx_getcachesize(brdhdl, &cachetotal, DX_CACHETOTAL) == -1) {
        printf("Error while getting cache size \n");
        /* Perform system error processing */
        exit(1);
    }
    .
    .
    .
    /* Download prompts to the on-board memory */
    .
    .
    .
    /* Check available size remaining for additional downloads */
    if (!dx_getcachesize(brdhdl, &cacheremaining, DX_CACHEREMAINING) == -1) {
        printf("Error while getting cache size \n");
        /* Perform system error processing */
        exit(1);
    }
}

■ See Also

- dx_cacheprompt( )
- dx_open( )
- dx_playiottdata( )
dx_getctinfo( )

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

**Inputs:** int chdev
CT_DEVINFO *ct_devinfop

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

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

**Category:** TDM Routing

**Mode:** synchronous

**Platform:** DM3, Springware, Intel® NetStructure™ IPT Series

### Description

The **dx_getctinfo( )** function returns information about a voice channel of a voice device. This information is contained in a 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 <strong>dx_open( )</strong></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; use dx_fileerrno() to obtain error value

Example

#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 */
        exit(1);
    }

    /* 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

- ag_getctinfo()
- dt_getctinfo() in the Digital Network Interface Software 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 • pointer to current absolute speed setting

Returns: 0 if success
          -1 if failure

Includes: srlib.h
dxxlib.h

Category: Speed and Volume

Mode: synchronous

Platform: DM3, Springware

---

Description

The dx_getcursv( ) function returns the specified current speed and volume settings on a channel. For example, use dx_getcursv( ) to determine the speed and volume level set interactively by a listener using DTMF digits during a play. DTMF digits are set as play adjustment conditions using the dx_setsvcond( ) function, or by one of the convenience functions, dx_addspddig( ) 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 dx_open( )</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>points to an integer that represents the current absolute speed setting for the channel. This value will be between -50% and +50%.</td>
</tr>
</tbody>
</table>

Cautions

On DM3 boards, if you close a device via dx_close( ) after modifying speed and volume table values using dx_setsvmt( ), the dx_getcursv( ) function may return incorrect speed and volume settings for the device. This is because the next dx_open( ) resets the speed and 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

- **EDX_SYSTEM**
  - Error from operating system; use `dx_fileerrno( )` to obtain error value

**Example**

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

/*
* Global Variables
*/

int dx xxxdev;
int curspeed, curvolume;

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

/*
* Get the Current Volume and Speed Settings
*/
if ( dx_getcursv( dx xxxdev, &curvolume, &curspeed ) == -1 ) {
    printf( "Unable to Get the Current Speed and" );
    printf( " Volume Settings
" );
    printf( "Lasterror = %d  Err Msg = %s
",
            ATDV_LASTERR( dx xxxdev ), ATDV_ERRMSGP( dx xxxdev ) );
    dx_close( dx xxxdev );
    exit( 1 );
} else {
    printf( "Volume = %d   Speed = %d\n", curvolume, curspeed );
}

/*
* Continue Processing
*/
* ...
* ...
*/
```
return the specified current speed and volume settings — dx_getcursv()


**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 NULL) if successful (synchronous)
- -1 if failure

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

**Category:** I/O

**Mode:** asynchronous or synchronous

**Platform:** DM3, Springware

---

**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.

<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 this function. For a list of possible termination conditions, see DV_TPT, on page 485.</td>
</tr>
<tr>
<td>digitp</td>
<td>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 483.</td>
</tr>
</tbody>
</table>

For more information about creating user-defined digits, see `dx_addtone( )`.

<table>
<thead>
<tr>
<th>mode</th>
<th>specifies whether to run <code>dx_getdig( )</code> asynchronously or synchronously. Specify one of the following:</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>
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). Use ATDX_TERMMSK() to determine the reason for termination.

The channel’s digit buffer contains up to 31 digits, collected on a First-In First-Out (FIFO) basis. Since the digits remain in the channel’s digit buffer until they are overwritten or cleared using dx_clrdigbuf(), the digits in the channel’s buffer may have been received prior to this function call. The DG_MAXDIGS define in dxxxlib.h specifies the maximum number of digits (that is, 31) that can be returned by a single call to dx_getdig().

Note: By default, after the thirty-first digit, all subsequent digits will be discarded. You can use the dx_setdigbuf() function with the mode parameter set to DX_DIGCYCLIC, which will cause all incoming digits to overwrite the oldest digit in the buffer.

If the function is operating synchronously and there are no digits in the buffer, the return value from this function will be 1, which indicates the NULL terminator.

Cautions

- On DM3 boards, 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 539). 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.
- 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; use **dx_fileerrno()** to obtain error value

### Example 1

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

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

int main()
{
    DV_TPT tpt[3];
    DV_DIGIT digp;
    int chdev, numdigs, cnt;

    /* open the channel with dx_open() and obtain channel device descriptor
       in chdev */
    if ((chdev = dx_open("dxxxB1C1", NULL)) == -1) {
        /* process error */
    }

    /* initiate the call */
    ...

    /* Set up the DV_TPT and get the digits */
    dx_clrtpt(tpt, 3);
    tpt[0].tp_type   = IO_CONT;
    tpt[0].tp_termno = DX_MAXDTMF;       /* Maximum number of digits */
    tpt[0].tp_length = 4;                /* terminate on 4 digits */
    tpt[0].tp_flags  = TF_MAXDTMF;       /* terminate if already in buf. */
    tpt[1].tp_type   = IO_CONT;
    tpt[1].tp_termno = DX_LCOFF;         /* LC off termination */
    tpt[1].tp_length = 3;                /* Use 30 msec (10 msec resolution timer) */
    tpt[1].tp_flags  = TF_LCOFF|TF_10MS;  /* level triggered, clear history,
```
collect digits from a channel digit buffer — dx_getdig()

collect digits from a channel digit buffer — dx_getdig()

#define MAXCHAN 24

int digit_handler();
DV_TPT stpt[3];
DV_DIGIT digp[256];

main()
{
    int i, chdev[MAXCHAN];
    char *chnamep;
    int srlmode;

    /* Set SRL to run in polled mode. */
    srlmode = SR_POLLMODE;
    if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1) {
        /* process error */
    }
    for (i=0; i<MAXCHAN; i++) {
        /* Set chnamep to the channel name - e.g., dxxxB1C1 */
        /* open the channel with dx_open( ). Obtain channel device */
        /* descriptor in chdev[i] */
        if ((chdev[i] = dx_open(chnamep, NULL)) == -1) {
            /* process error */
        }
        /* Using sr_enbhdlr(), set up handler function to handle dx_getdig() */
        /* completion events on this channel. */
        if (sr_enbhdlr(chdev[i], TDX_GETDIG, digit_handler) == -1) {
            /* process error */
        }
        dv_tpt tpt[3];
        dv_digit digp;

        tpt[0].tp_type = IO_INIT;
        tpt[0].tp_termno = DX_MAXTIME; /* Function Time */
        tpt[0].tp_length = 100; /* 10 seconds (100 msec resolution timer) */
        tpt[0].tp_flags = TF_MAXTIME; /* Edge-triggered */

        /* clear previously entered digits */
        if (dx_clearbuf(chdev) == -1) {
            /* process error */
        }
        if ((numdigs = dx_getdig(chdev, tpt, &digp, EV_SYNC)) == -1) {
            /* process error */
        }
        for (cnt=0; cnt < numdigs; cnt++) {
            printf("Digit received = \%c, digit type = \%d",
                   digp.dg_value[cnt], digp.dg_type[cnt]);
        }
        /* go to next state */
    }
}

Example 2

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

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

int digit_handler();
DV_TPT stpt[3];
DV_DIGIT digp[256];
dx_getdig() — collect digits from a channel digit buffer


dx_getdig( )

/* process error */
} /* initiate the call */

/* Set up the DV_TPT and get the digits */

dx_clrtpt(tpt,3);

tpt[0].tp_type = IO_CONT;
tpt[0].tp_termno = DX_MAXDTMF; /* Maximum number of digits */
tpt[0].tp_length = 4; /* terminate on 4 digits */
tpt[0].tp_flags = TF_MAXDTMF; /* terminate if already in buf */

tpt[1].tp_type = IO_CONT; /* LC off termination */
tpt[1].tp_termno = DX_LCOFF; /* Use 30 msec (10 msec resolution timer) */
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; /* Function Time */
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()
- dx_setdigtyp()
- DV_DIGIT data structure
• `dx_sethook()`
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

**Platform:** DM3, Springware

---

**Description**

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**

```c
/*$ 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
• dt_GetDllVersion( ) in the Digital Network Interface Software Reference
**dx_getevt( ) — monitor channel events synchronously**

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.

### 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 <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>
</tbody>
</table>
| timeout   | specifies the maximum amount of time in seconds to wait for an event to occur. `timeout` can have one of the following values:  
  - number of seconds – maximum length of time `dx_getevt( )` will wait for an event. When the time specified has elapsed, the function will terminate and return an error.  
  - -1 – `dx_getevt( )` will block until an event occurs; it will not time out.  
  - 0 – The function will return -1 immediately if no event is present. |

**Note:** 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.

### Cautions

It is recommended that you enable only one process per channel. The event that `dx_getevt( )` is waiting for may change if another process sets a different event for that channel. See `dx_setevtmsk( )` for more information.
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; use `dx_fileerrno()` to obtain error value

- **EDX_TIMEOUT**
  - Timeout time limit is reached

Example

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

main() {
    int chdev; /* channel descriptor */
    int timeout; /* timeout for function */
    DX_EBLK eblk; /* Event Block Structure */
    .
    .
    .

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

    /* Set RINGS or WINK as events to wait on */
    if (dx_setevtmsk(chdev,DM_RINGS|DM_WINK) == -1) {
        /* process error */
    }

    /* Set timeout to 5 seconds */
    timeout = 5;
    if (dx_getevt(chdev,&eblk,timeout) == -1) {
        /* process error */
        if (ATDV_LASTERR(chdev) == EDX_TIMEOUT) {    /* check if timed out */
            printf("Timed out waiting for event.\n");
        } else {
            /* further error processing */
            .
            .
        }
    }

    switch (eblk.ev_event) {
    case DE_RINGS:
        printf("Ring event occurred.\n");
        break;
    case DE_WINK:
        printf("Wink event occurred.\n");
        break;
    }
}
```
dx_getevt( ) — monitor channel events synchronously

break;
}

See Also

- dx_setevtmsk( )
- 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

**Platform:** DM3, Springware

---

**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: Specifying a board device handle is not supported on Springware boards.  
|               | Note: On high-density DM3 boards, 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 525. |

**Cautions**

- This function fails if an invalid device handle is specified.
- On DM3 analog boards, use `dx_getctinfo( )` to return information about the type of front end or network interface on the board. The network interface information is contained in the ct_nettype field of CT_DEVINFO.
dx_getfeaturelist( ) — retrieve feature support information for the device

### 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.)

- **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; use dx_fileerrno() to obtain error value

### Example

```c
#include <stdio.h>
#include <windows.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 \\
               \"%s\n\n", chname);
        exit(1);
    }

    if (dx_getfeaturelist(dev, &feature_table) == -1) {
        printf("\%s: Error \%d getting featurelist\n", chname, ATDV_LASTERR(dev));
        exit(2);
    }
}
```
retrieve feature support information for the device — dx_getfeaturelist( )

```c
printf("\n\n%s: Play Features: \n", chname);
if (feature_table.ft_play & FT_ADPCM) {
    printf("ADPCM ");
}
if (feature_table.ft_play & FT_PCM) {
    printf("PCM ");
}
if (feature_table.ft_play & FT_ALAW) {
    printf("ALAW ");
}
if (feature_table.ft_play & FT_ULAW) {
    printf("ULAW ");
}
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%s: 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 ");
}
```
dx_getfeaturelist() — retrieve feature support information for the device

if (feature_table.ft_record & FT_DRT8KHZ) {
    printf("DRT8KHZ ");
}
if (feature_table.ft_record & FT_DRT11KHZ) {
    printf("DRT11KHZ");
}
printf("\n\n\n%s: Tone Features:\n", chname);
if (feature_table.ft_tone & FT_GTDENABLED) {
    printf("GTDENABLED ");
}
if (feature_table.ft_tone & FT_GTGENABLED) {
    printf("GTGENABLED ");
}
if (feature_table.ft_tone & FT_CADENCE_TONE) {
    printf("CADENCE_TONE");
}
printf("\n\n\n%s: E2P 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");
}
printf("\n\n\n%s: FAX 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");
}
if (feature_table.ft_fax & FT_FAX_EXT_TBL)
    && (feature_table.ft_send & FT_SENDFAX_TXFILE.ASCII) {
    printf("SOFTFAX \n");
}
printf("\n\n\n%s: FrontEnd 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");
}
printf("\n\n\n%s: Miscellaneous Features:\n", chname);
retrieve feature support information for the device — `dx_getfeaturelist()`

```c
if (feature_table.ft_misc & FT_CALLERID) {
    printf("CALLERID\n");
}
printf("\n");
dx_close(dev);
```

**See Also**

- `dx_getctinfo()`
dx_getparm( ) — get the current parameter settings

dx_getparm( )

Name: int dx_getparm(dev, parm, valuep)

Inputs: int dev 
   • valid channel or board device handle
   unsigned long parm 
   • parameter type to get value of
   void *valuep 
   • pointer to variable for returning parameter value

Returns: 0 if success
         -1 if failure

Includes: srlib.h
          dxxxlib.h

Category: Configuration

Mode: synchronous

Platform: DM3, Springware

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 dx_open( )</td>
</tr>
</tbody>
</table>
get the current parameter settings — dx_getparm( )

<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 valuep. 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 dx_setparm( ) function description. For DM3 boards, board parameter defines are described in Table 12, “Voice Board Parameters (DM3)”, on page 390. For Springware boards, board parameter defines are described in Table 13, “Voice Board Parameters (Springware)”, on page 390. For DM3 boards, channel parameter defines are described in Table 14, “Voice Channel Parameters (DM3)”, on page 391. For Springware boards, channel parameter defines are described in Table 15, “Voice Channel Parameters (Springware)”, on page 394.</td>
</tr>
<tr>
<td>valuep</td>
<td>Points to the variable where the value of the parameter specified in parm should be returned. Note: You must use a void* cast on the returned parameter value, as demonstrated in the Example section code for this function. Note: valuep 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 PM_SHORT, PM_BYTE, PM_INT, PM_LONG, PM_FLSTR (fixed length string), and PM_VLSTR (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_BADPARAM
  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; use dx_fileerrno( ) to obtain error value
Example

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

main()
{
    int bddev;
    unsigned short parmval;

    /* open the board using dx_open(). Obtain board device descriptor in
     * bddev
     */
    if ((bddev = dx_open("dxxxB1",NULL)) == -1) {
        /* process error */
    }

    parmval = 0;    /* CLEAR parmval */

    /* get the number of channels on the board. DXBD_CHNUM is of type
     * unsigned short as specified by the PM_SHORT define in the definition
     * for DXBD_CHNUM in dxxxlib.h. The size of the variable parmval is
     * sufficient to hold the value of DXBD_CHNUM.
     */
    if (dx_getparm(bddev, DXBD_CHNUM, (void *)&parmval) == -1) {
        /* process error */
    }

    printf("Number of channels on board = %d",parmval);
    
}
```

See Also

- `dx_setparm()`
**dx_GetRscStatus( )**

**Name:** int dx_GetRscStatus(chdev, rsctype, status)

**Inputs:**
- int chdev • valid channel device handle
- int rsctype • type of resource
- int *status • pointer to assignment status

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

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

**Category:** Configuration

**Mode:** synchronous

**Platform:** Springware

---

### Description

The `dx_GetRscStatus( )` function returns the assignment status of the shared resource for the specified channel.

<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>rsctype</td>
<td>specifies the type of shared resource:</td>
</tr>
<tr>
<td></td>
<td>• RSC_FAX – shared fax resource (DSP-based Group 3 Fax, also known as Softfax)</td>
</tr>
<tr>
<td>status</td>
<td>points to the data that represents the assignment status of the resource:</td>
</tr>
<tr>
<td></td>
<td>• RSC_ASSIGNED – a shared resource of the specified <code>rsctype</code> is assigned to the channel</td>
</tr>
<tr>
<td></td>
<td>• RSC_NOTASSIGNED – a shared resource of the specified <code>rsctype</code> is not assigned to the channel</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
dx_GetRscStatus() — return assignment status of a shared resource

EDX_SYSTEM
Error from operating system; use dx_fileerrno() to obtain error value

■ Example

```c
/* Check whether a shared Fax resource is assigned to the voice channel */
#include <windows.h>
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <faxlib.h>

main()
{
    int chdev; /* Fax channel device handle */
    int status;

    /*Open the Voice channel resource (device) using dx_open(). */

    /*Open the FAX channel resource (device) */
    if((chdev = fx_open("dxxxB1C1", NULL)) == -1) {
        /*Error opening device */
        /* Perform system error processing */
        exit(1);
    }

    /*Get current Resource Status*/
    if(dx_GetRscStatus(chdev, RSC_FAX, &status) == -1) {
        printf("Error - %s (error code %d)\n", ATDV_ERRMSGP(chdev), ATDV_LASTERR(chdev));
        if(ATDV_LASTERR(chdev) == EDX_SYSTEM) {
            /* Perform system error processing */
        }
    } else {
        printf("The resource status ::%d\n", status);
    }
}
```

■ See Also

- DSP Fax topic in the Fax Software Reference
**dx_GetStreamInfo( )**

**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.

**Parameter** | **Description**
--- | ---
`hBuffer` | specifies the circular stream buffer handle
`StreamStatStruct` | specifies a pointer to the `DX_STREAMSTAT` data structure. For more information on this structure, see `DX_STREAMSTAT`, on page 512.

**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;
}```
dx_GetStreamInfo() — retrieve information about the circular stream buffer

```
if ((hBuffer = dx_OpenStreamBuffer(nBuffSize)) < 0)
{
    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()

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

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

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

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

**Category:** Speed and Volume

**Mode:** synchronous

**Platform:** DM3, Springware

---

**Description**

The `dx_getsvmt()` function returns the current speed or volume modification table to the DX_SVMT structure.

<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>
| tabletype | specifies whether to retrieve the speed or the volume modification table:  
- SV_SPEEDTBL – retrieve the speed modification table values  
- SV_VOLUMETBL – retrieve the volume modification table values |
| svmtp     | points to the DX_SVMT structure that contains the speed and volume modification table entries |

**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
dx_getsvmt( ) — return the current speed or volume modification table

EDX_SPDVOL
Must specify either SV_SPEEDTBL or SV_VOLUMETBL

EDX_SYSTEM
Error from operating system; use dx_fileerrno() to obtain error value

Example

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

/*
 * Global Variables
*/

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

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

    /*
     * Get the Current Volume Modification Table
     */
    memset( &svmt, 0, sizeof( DX_SVMT ) );
    if ( dx_getsvmt( dxxxdev, SV_VOLUMETBL, &svmt ) == -1 ){
        printf( "Unable to Get the Current Volume" );
        printf( " Modification Table\n" );
        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
", 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 );

See Also

- dx_addspddig()
- dx_addvoldig()
- dx_adjsv()
- dx_clrsvcond()
- dx_getcursv()
- dx_setsvcond()
- dx_setsvmt()
- speed and volume modification tables in Voice API Programming Guide
- DX_SVMT data structure
dx_getxmitslot( ) — get TDM bus time slot number of voice transmit channel

---

**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:** srlib.h
dxxxlib.h
- **Category:** TDM routing
- **Mode:** synchronous
- **Platform:** DM3, Springware

---

**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 529.

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

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

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.
- On DM3 boards, this function is supported in a flexible routing configuration but not a fixed routing configuration. This document assumes that a flexible routing configuration is the configuration of choice. For more information on API restrictions in a fixed routing configuration, see the Voice API Programming Guide.
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_BADINDEX
Invalid Switch Handler index number

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; use dx_fileerrno() to obtain error value

Example

```
#include <windows.h>
#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 */
```
dx_getxmitslot() — get TDM bus time slot number of voice transmit channel

/* 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_numts = 1;
sc_tsinfo.sc_tsarrayp = &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()
- dt_listen() in the Digital Network Interface Software Reference
- fx_listen() in the Fax Software Reference
### dx_getxmitslotecr( )

**Name:** int dx_getxmitslotecr(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
dxxxlib.h

**Category:** Echo Cancellation Resource

**Mode:** synchronous

**Platform:** Springware

---

#### Description

The `dx_getxmitslotecr()` function returns the transmit time slot number assigned to the echo cancellation resource (ECR) of the specified voice channel device. The time slot information is contained in an SC_TSINFO structure. For more information on this structure, see SC_TSINFO, on page 529.

**Note:** The ECR functions have been replaced by the continuous speech processing (CSP) API functions. CSP provides enhanced echo cancellation. For more information, see the Continuous Speech Processing API Programming Guide and Continuous Speech Processing API Library Reference.

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

#### Cautions

This function fails when:
- An invalid channel device handle is specified.
- The ECR feature is not enabled on the board specified.
- The ECR feature is not supported on the board 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
**Example**

```
#include <stdio.h>
#include <windows.h>
#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 devices */
    if ((chdev = dx_open("dxxxB1C1", 0)) == -1) {
        /* Perform system error processing */
        exit(1);
    }

    /* Fill in the TDM bus time slot information */
    sc_tsinfo.sc_nums = 1;
    sc_tsinfo.sc_tsarrayp = &scts;
```
get echo cancellation resource transmit time slot number — dx_getxmitslotecr()

/* Get TDM bus time slot on which the echo-cancelled signal will be transmitted */
if (dx_getxmitslotecr(chdev, &sc_tsinfo) == -1) {
    printf("Error message = \%s\n", ATDV_ERRMSGP(chdev));
    exit(1);
}

printf("\%s transmits the echo cancelled signal on \%d", ATDV_NAMEP(chdev),scts);
return(0);

See Also
- dx_listenecr()
- dx_listenecrex()
- dx_unlistenecr()
- dx_getxmitslot()
**dx_gtcallid( ) — return the calling line Directory Number**

**dx_gtcallid( )**

**Name:** int dx_gtcallid (chdev, bufferp)

**Inputs:**
- int chdev • valid channel device handle
- unsigned char *bufferp • pointer to buffer where calling line Directory Number is returned

**Returns:**
- 0 success
- -1 error return code

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

**Category:** Caller ID

**Mode:** synchronous

**Platform:** Springware

**Description**

The **dx_gtcallid( )** function returns the calling line Directory Number (DN) sent by the Central Office (CO). On successful completion, a NULL-terminated string containing the caller’s phone number (DN) is placed in the buffer. Non-numeric characters (punctuation, space, dash) may be included in the number string; however, the string may not be suitable for dialing without 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 a channel was opened using <strong>dx_open( )</strong></td>
</tr>
<tr>
<td>bufferp</td>
<td>pointer to buffer where calling line Directory Number (DN) is returned</td>
</tr>
</tbody>
</table>

*Note:* Make sure to allocate a buffer size large enough to accommodate the DN returned by this function.

Caller ID information is available for the call from the moment the ring event is generated and until the call is either disconnected (for answered calls) or until rings are no longer received from the CO (for unanswered calls). If the call is answered before caller ID information has been received from the CO, caller ID information will not be available.

If the call is not answered and the ring event is received before the caller ID information has been received from the CO, caller ID information will not be available until the beginning of the second ring (CLASS, ACLIP) or the beginning of the first ring (CLIP, JCLIP).

To determine if caller ID information has been received from the CO before issuing a **dx_gtcallid( )** or **dx_gtextcallid( )** caller ID function, check the event data in the event block. When the ring event is received, the event data field in the event block is bitmapped and indicates that caller ID information is available when bit 0 (LSB) is set to a 1.
Based on the caller ID options provided by the CO and for applications that require only the calling line Directory Number (DN), issue the `dx_gtcallid()` function to get the calling line DN.

Based on the caller ID options provided by the CO and for applications that require additional caller ID information, issue the `dx_gtextcallid()` function for each type of caller ID message required.

### Cautions

- If caller ID is enabled, on-hook digit detection (DTMF, MF, and global tone detection) will not function.
- This function does not differentiate between a checksum error and no caller ID.

### 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_CLIDBLK**
  - Caller ID is blocked or private or withheld (other information may be available using `dx_gtextcallid()`)
- **EDX_CLIDINFO**
  - Caller ID information is not sent or caller ID information is invalid
- **EDX_CLIDOOA**
  - Caller ID is out of area (other information may be available using `dx_gtextcallid()`)
- **EDX_SYSTEM**
  - Error from operating system; use `dx_fileerrno()` to obtain error value

### Example

```c
/*$ dx_gtcallid() example $*/
#include <windows.h>
#include <sys/types.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
/* Intel(r) Dialogic(r) Includes */
#include "srllib.h"
#include "dxxxlib.h"
```
int main()
{
    int numRings = 2; /* In the US */
    int ringTimeout = 20; /* 20 seconds */
    int chdev; /* Channel descriptor */
    unsigned short parmval;
    unsigned char buffer[81];

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

    /* Enable the caller ID functionality */
    parmval = DX_CALLIDENABLE;
    if (dx_setparm(chdev, DXCH_CALLID, (void *) &parmval) == -1) {
        /* process error */
        exit(0);
    }

    /******************************************************************
    * Set the number of rings required for a RING event to permit
    * receipt of the caller ID information. In the US, caller ID
    * information is transmitted between the first and second rings
    *-------------------------------------------------------------------*/
    parmval = numRings; /* 2 in the US */
    if (dx_setparm(chdev, DXCH_RINGCNT, &parmval) == -1) {
        /* process error */
        exit(0);
    }

    /* Put the channel onhook */
    if (dx_sethook(chdev, DX_ONHOOK, EV_SYNC) == -1) {
        /* process error */
        exit(0);
    }

    /* Wait for 2 rings and go offhook (timeout after 20 seconds) */
    if (dx_wtring(chdev, numRings, DX_OFFHOOK, ringTimeout) == -1) {
        /* process error */
    }

    /* Get just the caller ID */
    if (dx_gtcallid(chdev, buffer) == -1) {
        /* Can check the specific error code */
        if (ATDV_LASTERR(chdev) == EDX_CLIDBLK) {
            printf("Caller ID information blocked \n");
        } else if (ATDV_LASTERR(chdev) == EDX_CLIDOOGA) {
            printf("Caller out of area \n");
        } else {
            /* Or print the pre-formatted error message */
            printf("Error: %s \n", ATDV_ERRMSGP(chdev));
        }
    } else {
        printf("Caller ID = \%s\n", buffer);
    }

    /******************************************************************
    * If the message is an MDM (Multiple Data Message), then
    * additional information is available.
    * First get the frame and check the frame type. If Class MDM,
    * get and print additional information from submessages.
    *-------------------------------------------------------------------*/
if (dx_gtextcallid(chdev, CLIDINFO_FRAMETYPE, buffer) != -1) {
    if (buffer[0] == CLASSFRAME_MDM) {
        /* Get and print the date and time */
        if (dx_gtextcallid(chdev, MCLASS_DATETIME, buffer) == -1) {
            /* process error */
            printf("Error: %s\n", ATDV_ERRMSGP(chdev));
        } else {
            printf("Date/Time = %s\n", buffer);
        }
        /* Get and print the caller name */
        if (dx_gtextcallid(chdev, MCLASS_NAME, buffer) == -1) {
            /* process error */
            printf("Error: %s\n", ATDV_ERRMSGP(chdev));
        } else {
            printf("Caller Name = %s\n", buffer);
        }
        /* Get and print the Dialed Number */
        if (dx_gtextcallid(chdev, MCLASS_DDN, buffer) == -1) {
            /* process error */
            printf("Error: %s\n", ATDV_ERRMSGP(chdev));
        } else {
            printf("Dialed Number = %s\n", buffer);
        }
    } else {
        printf("Submessages not available - not an MDM message\n");
    }
}
dx_close(chdev);
return(0);

See Also

- dx_gtextcallid()
- dx_wtcallid()
- dx_setparm()
- dx_setevtmsk()
- dx_getevt()
- DX_EBLK data structure
dx_gtextcallid( )

- **Name:** int dx_gtextcallid (chdev, infotype, bufferp)
- **Inputs:**
  - int chdev • channel device handle
  - int infotype • message type ID
  - unsigned char *bufferp • pointer to where to return the requested caller ID message
- **Returns:**
  - 0 success
  - -1 error return code
- **Includes:** srllib.h
dxxxlib.h
- **Category:** Caller ID
- **Mode:** synchronous
- **Platform:** Springware

### Description

The dx_gtextcallid( ) function returns the requested caller ID message by specifying the Message Type ID. The application can issue this function as many times as required to get the desired caller ID messages (such as date and time, calling line subscriber name, reason why caller ID is not available).

The format and content of the caller ID messages are based on published telecommunication standards. The actual formatting and content of the data returned depend on the implementation and level of service provided by the originating and destination Central Offices.

**Note:** For CLASS and ACLIP, do not use Multiple Data Message Type IDs with caller ID information in Single Data Message format.

<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 a channel was opened using dx_open( )</td>
</tr>
<tr>
<td>infotype</td>
<td>the Message Type ID for the specific caller ID information to receive. Message Type IDs for CLASS, ACLIP, JCLIP and CLIP are listed on the following pages. See Table 5, “Caller ID Common Message Types”, on page 264, Table 6, “Caller ID CLASS Message Types (Multiple Data Message)”, on page 265, Table 7, “Caller ID ACLIP Message Types (Multiple Data Message)”, on page 265, Table 8, “Caller ID CLIP Message Types”, on page 266, and Table 9, “Caller ID JCLIP Message Types (Multiple Data Message)”, on page 266.</td>
</tr>
<tr>
<td>bufferp</td>
<td>pointer to buffer where the requested caller ID message is to be stored. All returns are NULL terminated.</td>
</tr>
</tbody>
</table>
**Cautions**

- To allow the reception of caller ID information from the central office before answering a call (application channel goes off-hook):
  - For CLASS and ACLIP, set the ring event to occur on or after the second ring.
  - For CLIP and JCLIP, set the ring event to occur on or after the first ring.

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

- CLASS and ACLIP: Do not use Multiple Data Message Type IDs with caller ID information in Single Data Message format.
- Make sure the buffer size is large enough to hold the caller ID message(s) returned by this function.
- JCLIP operation requires that the Japanese country-specific parameter file be installed and configured (select Japan in the country configuration).
- If the application program performs a **dx_sethook( )** on an on-hook channel device during the short period before the first ring and when the channel is receiving JCLIP caller ID information, the function will return an error.

**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_CLIDBLK**
  - Caller ID is blocked or private or withheld (**infotype = CLIDINFO_CALLID**)
- **EDX_CLIDINFO**
  - Caller ID information not sent, sub-message(s) requested not available or caller ID information invalid
- **EDX_CLIDOAA**
  - Caller ID is out of area (**infotype = CLIDINFO_CALLID**)
- **EDX_SYSTEM**
  - Error from operating system; use **dx_fileerrno( )** to obtain error value

All Message Types (**infotype**) can produce an **EDX_CLIDINFO** error. Message Type **CLIDINFO_CALLID** can also produce **EDX_CLIDOAA** and **EDX_CLIDBLK** errors.

**Common Message Types**

The following standard Message Types are available for:

- **CLASS** (Single Data Message)
- **CLASS** (Multiple Data Message)
dx_gtextcallid( ) — retrieve a caller ID message

- ACLIP (Single Data Message)
- ACLIP (Multiple Data Message)
- CLIP
- JCLIP

All returns are NULL terminated.

### Table 5. Caller ID Common Message Types

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition/Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLIDINFO_CMPLT</td>
<td>All caller ID information as sent from the CO (maximum of 258 bytes; includes header and length byte at the beginning). Can produce EDX_CLIDINFO error.</td>
</tr>
<tr>
<td>CLIDINFO_GENERAL</td>
<td>Date and time (20 bytes - formatted with / and : characters; padded with spaces). Caller phone number or reason for absence (20 bytes; padded with spaces). Caller name or reason for absence (variable length ≥0; not padded). Can produce EDX_CLIDINFO error. See Figure 1.</td>
</tr>
<tr>
<td>CLIDINFO_CALLID</td>
<td>Caller ID (phone number). Can produce EDX_CLIDINFO, EDX_CLIDOAA, and EDX_CLIDBLK errors.</td>
</tr>
<tr>
<td>CLIDINFO_FRAMETYPE</td>
<td>Indicates caller ID frame. Does not apply to CLIP. Can produce EDX_CLIDINFO error. Values (depending upon service type):</td>
</tr>
<tr>
<td></td>
<td>• CLASSFRAME_SDH</td>
</tr>
<tr>
<td></td>
<td>• CLASSFRAME_MDM</td>
</tr>
<tr>
<td></td>
<td>• ACLIPFRAME_SD</td>
</tr>
<tr>
<td></td>
<td>• ACLIPFRAME_MDM</td>
</tr>
<tr>
<td></td>
<td>• JCLIPFRAME_MDM</td>
</tr>
</tbody>
</table>

**Figure 1. Format of General Caller ID Information**

<table>
<thead>
<tr>
<th>Date and Time (20 bytes)</th>
<th>Phone Number (20 bytes)</th>
<th>Name (variable length ≥0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>012345678901234567890123456789012345678901234567890123456789</td>
<td>bbbbbbbbbbbbbb</td>
<td>2019933000bbbbbbbbbbbbbbbbb</td>
</tr>
<tr>
<td>04/04b10:11bbbbbbbbbb</td>
<td>2019933000bbbbbbbbbbbbbbbbb</td>
<td>JOHNBDOE</td>
</tr>
<tr>
<td>04/04b10:11bbbbbbbbbb</td>
<td>2019933000bbbbbbbbbbbbbbbbb</td>
<td>2019933000bbbbbbbbbbbbbbbbb</td>
</tr>
<tr>
<td>04/04b10:11bbbbbbbbbb</td>
<td>2019933000bbbbbbbbbbbbbbbbb</td>
<td>2019933000bbbbbbbbbbbbbbbbb</td>
</tr>
<tr>
<td>04/04b10:11bbbbbbbbbb</td>
<td>2019933000bbbbbbbbbbbbbbbbb</td>
<td>2019933000bbbbbbbbbbbbbbbbb</td>
</tr>
</tbody>
</table>

**Legend:**

- b = Blank
- n = Null
- O = Out of Area
- P = Private

### Message Types for CLASS (Multiple Data Message)

See Table 5 for the standard Message Types that can also be used. Table 6 lists Message Types that can produce an EDX_CLIDINFO error. All returns are NULL terminated.
Table 6. Caller ID CLASS Message Types (Multiple Data Message)

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition/Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCLASS_DATETIME</td>
<td>Date and Time (as sent by CO without format characters / and :)</td>
</tr>
<tr>
<td>MCLASS_DN</td>
<td>Calling line directory number (digits only)</td>
</tr>
<tr>
<td>MCLASS_DDN</td>
<td>Dialed number (digits only)</td>
</tr>
<tr>
<td>MCLASS_ABSENCE1</td>
<td>Reason for absence of caller ID (only available if caller name is absent): O = out of area, P = private</td>
</tr>
<tr>
<td>MCLASS_REDIRECT</td>
<td>Call forward: 0 = universal; 1 = busy; 2 = unanswered</td>
</tr>
<tr>
<td>MCLASS_QUALIFIER</td>
<td>L = long distance call</td>
</tr>
<tr>
<td>MCLASS_NAME</td>
<td>Calling line subscriber name</td>
</tr>
<tr>
<td>MCLASS_ABSENCE2</td>
<td>Reason for absence of name (only available if caller name is absent): O = out of area, P = private</td>
</tr>
</tbody>
</table>

Message Types for ACLIP (Multiple Data Message)

See Table 5, “Caller ID Common Message Types”, on page 264 for the standard Message Types that can also be used. Table 7 lists Message Types that can produce an EDX_CLIDINFO error. All returns are NULL terminated.

Table 7. Caller ID ACLIP Message Types (Multiple Data Message)

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition/Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>MACLIP_DATETIME</td>
<td>Date and Time (as sent by CO without format characters / and :)</td>
</tr>
<tr>
<td>MACLIP_DN</td>
<td>Calling line directory number (digits only)</td>
</tr>
<tr>
<td>MACLIP_DDN</td>
<td>Dialed number (digits only)</td>
</tr>
<tr>
<td>MACLIP_ABSENCE1</td>
<td>Reason for absence of caller ID (only available if caller name is absent): O = out of area, P = private</td>
</tr>
<tr>
<td>MACLIP_REDIRECT</td>
<td>Call forward: 0 = universal; 1 = busy; 2 = unanswered</td>
</tr>
<tr>
<td>MACLIP_QUALIFIER</td>
<td>L = long distance call</td>
</tr>
<tr>
<td>MACLIP_NAME</td>
<td>Calling line subscriber name</td>
</tr>
<tr>
<td>MACLIP_ABSENCE2</td>
<td>Reason for absence of name (only available if caller name is absent): O = out of area, P = private</td>
</tr>
</tbody>
</table>

Message Types for CLIP

See Table 5, “Caller ID Common Message Types”, on page 264 for the standard Message Types that can also be used. Table 8 lists Message Types that can produce an EDX_CLIDINFO error. All returns are NULL terminated.
Table 8. Caller ID CLIP Message Types

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition/Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLIP_DATETIME</td>
<td>Date and Time (as sent by CO without format characters / and :)</td>
</tr>
<tr>
<td>CLIP_DN</td>
<td>Calling line directory number (digits only)</td>
</tr>
<tr>
<td>CLIP_DDNR</td>
<td>Dialed number (digits only)</td>
</tr>
<tr>
<td>CLIP_ABSENCE1</td>
<td>Reason for absence of caller ID (only available if caller name is absent): O = out of area, P = private</td>
</tr>
<tr>
<td>CLIP_NAME</td>
<td>Calling line subscriber name</td>
</tr>
<tr>
<td>CLIP_ABSENCE2</td>
<td>Reason for absence of name (only available if caller name is absent): O = out of area, P = private</td>
</tr>
<tr>
<td>CLIP_CALLTYPE</td>
<td>1 = voice call, 2 = ring back when free call, 129 = message waiting call</td>
</tr>
<tr>
<td>CLIP_NETMSG</td>
<td>Network Message System status: number of messages waiting</td>
</tr>
</tbody>
</table>

### Message Types for JCLIP (Multiple Data Message)

See Table 5, “Caller ID Common Message Types”, on page 264 for the standard Message Types that can also be used. Table 9 lists Message Types that can produce an EDX_CLIDINFO error. All returns are NULL terminated.

Table 9. Caller ID JCLIP Message Types (Multiple Data Message)

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition/Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>JCLIP_DN</td>
<td>Calling line directory number (digits only)</td>
</tr>
<tr>
<td>JCLIP_DDNR</td>
<td>Dialed number (digits only)</td>
</tr>
<tr>
<td>JCLIP_ABSENCE1</td>
<td>Reason for absence of caller ID (only available if caller name is absent): O = out of area or unknown reason, P = private (denied by call originator), C = public phone, S = service conflict (denied by call originator’s network)</td>
</tr>
<tr>
<td>JCLIP_ABSENCE2</td>
<td>Reason for absence of name (only available if caller name is absent): O = out of area or unknown reason, P = private (denied by call originator), C = public phone, S = service conflict (denied by call originator’s network)</td>
</tr>
</tbody>
</table>

By passing the proper Message Type ID, the dx_gtextcallid() function can be used to retrieve the desired message(s). For example:

- CLIDINFO_CMPLT can be used to get the complete caller ID frame including header, length, sub-message(s) as sent by the CO
- CLIDINFO_GENERAL can be used to get messages including date and time (formatted), caller’s Directory Number (DN), and name
- CLIDINFO_CALLID can be used to get caller’s Directory Number (DN)
- CLIDINFO_FRAMETYPE can be used to determine the type of caller ID frame (for example: CLASS SDM or CLASS MDM, ACLIP SDM or ACLIP MDM, JCLIP MDM)
- MCLASS_DDNR can be used to get the dialed number for CLASS MDM (digits only)
- MACLIP_DDNR can be used to get the dialed number for ACLIP MDM (digits only)
- CLIP_NAME can be used to get the calling line subscriber name for CLIP
MACLIP_NAME can be used to get the calling line subscriber name for ACLIP

Caller ID information is available for the call from the moment the ring event is generated (if the
ring event is set to occur on or after the second ring (CLASS, ACLIP) or set to occur on or after the
first ring (CLIP, JCLIP)) until either of the following occurs:

- If the call is answered (the application channel goes off-hook), the caller ID information is
  available to the application until the call is disconnected (the application channel goes on-
  hook).
- If the call is not answered (the application channel remains on-hook), the caller ID information
  is available to the application until rings are no longer received from the Central Office
  (signaled by ring off event, if enabled).

**Example**

```c
/*$ dx_gtextcallid() example to obtain all available caller ID information $*/

#include <windows.h>
#include <sys/types.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
/* Intel Dialogic Includes */
#include "srllib.h"
#include "dxnxlib.lib"

int main()
{
    int numRings = 2;          /* In the US */
    int ringTimeout = 20;      /* 20 seconds */
    int chdev;                 /* Channel descriptor */
    unsigned short parmval;
    unsigned char buffer[81];
    /* Open channel */
    if ((chdev=dx_open("dxxxB1C1", NULL)) == -1) {
        /* process error */
        exit(0);
    }
    /* Enable the caller ID functionality */
    parmval = DX_CALLIDENABLE;
    if (dx_setparm(chdev, DXCH_CALLID, (void *) &parmval) == -1) {
        /* process error */
        exit(0);
    }
    /******************************************************************
    * Set the number of rings required for a RING event to permit
    * receipt of the caller ID information.  In the US, caller ID
    * information is transmitted between the first and second rings
    ******************************************************************/
    parmval = numRings;        /* 2 in the US */
    if (dx_setparm(chdev, DXCH_RINGCNT, &parmval) == -1) {
        /* process error */
        exit(0);
    }
}
```

**Example (continued)**

```c
#include <windows.h>
#include <sys/types.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
/* Intel Dialogic Includes */
#include "srllib.h"
#include "dxnxlib.lib"

int main()
{
    int numRings = 2;          /* In the US */
    int ringTimeout = 20;      /* 20 seconds */
    int chdev;                 /* Channel descriptor */
    unsigned short parmval;
    unsigned char buffer[81];
    /* Open channel */
    if ((chdev=dx_open("dxxxB1C1", NULL)) == -1) {
        /* process error */
        exit(0);
    }
    /* Enable the caller ID functionality */
    parmval = DX_CALLIDENABLE;
    if (dx_setparm(chdev, DXCH_CALLID, (void *) &parmval) == -1) {
        /* process error */
        exit(0);
    }
    /******************************************************************
    * Set the number of rings required for a RING event to permit
    * receipt of the caller ID information.  In the US, caller ID
    * information is transmitted between the first and second rings
    ******************************************************************/
    parmval = numRings;        /* 2 in the US */
    if (dx_setparm(chdev, DXCH_RINGCNT, &parmval) == -1) {
        /* process error */
        exit(0);
    }
}
```
/* Put the channel onhook */
if (dx_sethook(chdev, DX_ONHOOK, EV_SYNC) == -1) {
    /* process error */
    exit (0);
}

/* Wait for 2 rings and go offhook (timeout after 20 seconds) */
if (dx_wtring(chdev, numRings, DX_OFFHOOK, ringTimeout) == -1) {
    /* process error */
}

/**************************************************************************/
/* If the message is an MDM (Multiple Data Message), then */
/* individual submessages are available. */
/* First get the frame and check the frame type. If Class MDM, */
/* get and print information from submessages. */
/**************************************************************************/
if (dx_gtextcallid(chdev,CLIDINFO_FRAMETYPE, buffer) != -1) {
    if(buffer[0] == CLASSFRAME_MDM) {
        /* Get and print the caller ID */
        if (dx_gtextcallid(chdev, MCLASS_DN, buffer) != -1) {
            printf("Caller ID = %s\n", buffer);
        }
        /* This is another way to obtain caller ID (regardless of frame type)*/
    } else if (dx_gtextcallid(chdev, CLIDINFO_CALLID, buffer) != -1) {
        printf("Caller ID = %s\n", buffer);
    } else {
        /* print the reason for the Absence of caller ID */
        printf("Caller ID not available: %s\n", ATDV_ERRMSGP(chdev));
    }
    /* Get and print the Caller Name */
    if (dx_gtextcallid(chdev, MCLASS_NAME, buffer) != -1) {
        printf("Caller Name = %s\n", buffer);
    }
    /* Get and print the Date and Time */
    if (dx_gtextcallid(chdev, MCLASS_DATETIME, buffer) != -1) {
        printf("Date/Time = %s\n", buffer);
    }
    /* Get and print the Dialed Number */
    if (dx_gtextcallid(chdev, MCLASS_DDN, buffer) != -1) {
        printf("Dialed Number = %s\n", buffer);
    } else {
        printf("Submessages not available - not an MDM message\n");
    }
}
else {
    printf("Submessages not available - not an MDM message\n");
    /* Get just the caller ID */
    if (dx_gtextcallid(chdev, CLIDINFO_CALLID, buffer) != -1) {
        printf("Caller ID = %s\n", buffer);
    } else {
        /* print the reason for the absence of caller ID */
        printf("Caller ID not available: %s\n", ATDV_ERRMSGP(chdev));
    }
}

/**************************************************************************/
/* If desired, the date/time, caller name, and caller ID can */
/* be obtained together. */
/**************************************************************************/
if (dx_gtextcallid(chdev, CLIDINFO_GENERAL, buffer) != -1) {
    printf("Date/Time, Caller Number, and Caller ID = %s\n", buffer);
} else {
    /* Print out the error message */
    printf("Error: %s\n", ATDV_ERRMSGP(chdev));
}
void dx_gtextcallid()
{
    dx_close(chdev);
    return(0);
}

See Also

- dx_gtcallid()
- dx_wtcallid()
**dx_gtsernum( )** — retrieve the board serial number

The **dx_gtsernum( )** function returns the board serial number, either the standard serial number or the silicon serial number, where supported. When available, the silicon serial number is the preferred method for uniquely identifying boards.

The board serial number consists of eight ASCII characters and is printed on the serial number sticker on the board. The silicon serial number consists of an additional six-byte serial number encoded into the board and can include non-printable characters.

The serial number and silicon serial number can be used for developing software security in an application program. For example, an application program can be “locked” to an Intel® telecom board as part of the application installation procedure, by getting and saving the serial number in a secure place within the application. From then on, when the application is executed, it can check for the presence of the board and match it with the board serial number secured within the application program.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>devd</strong></td>
<td>specifies a valid board device handle</td>
</tr>
<tr>
<td><strong>subfcn</strong></td>
<td>specifies one of the following sub-functions:</td>
</tr>
<tr>
<td></td>
<td>• GS_SN – returns the standard board serial number, consisting of eight ASCII characters followed by a NULL byte. This number is printed on the serial number sticker attached to the board.</td>
</tr>
<tr>
<td></td>
<td>• GS_SSN – returns the board silicon serial number (if supported), consisting of six bytes of any value, including 0x00. An EDX_BADPROD error is returned if the specified board does not support the silicon serial number.</td>
</tr>
<tr>
<td><strong>buffp</strong></td>
<td>pointer to buffer where the serial number is returned</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_SYSTEM
   Error from operating system; use `dx_fileerrno()` to obtain error value. The board device is busy.

EDX_BADPARM
   Invalid device handle or sub-function.

EDX_BADPROD
   The board does not support GS_SSN (silicon serial number).

Example

```c
/*$ dx_gtsernum() example $*/
#include "stdio.h"
#include "srllib.h"
#include "dxxxlib.h"

void main(int argc, char **argv)
{
    int dev;
    char serial[10];
    /* open the board device */
    if ((dev=dx_open("dxxxB1",0 )) == -1) {
        printf("Error opening dxxxB1\n");
        exit(1);
    }
    /* get the board serial number and display it */
    if (dx_gtsernum(dev, GS_SN, serial) == 0) {
        printf("dxxxB1: %s\n", serial);
    } else {
        printf("Error %d, %s\n", ATDV_LASTERR(dev), ATDV_ERRMSGP(dev));
        dx_close(dev);
        exit(0);
    }
}
```

See Also

- `dx_open()`
dx_initcallp() — initialize and activate call progress analysis

On Springware boards, the **dx_initcallp()** function initializes and activates call progress analysis on the channel identified by **chdev**. In addition, this function adds all tones used in call progress analysis to the channel’s global tone detection (GTD) templates.

On DM3 boards, call progress analysis is enabled directly through the **dx_dial()** function.

On Springware boards, to use call progress analysis, **dx_initcallp()** must be called prior to using **dx_dial()** on the specified channel. If **dx_dial()** is called before initializing the channel with **dx_initcallp()**, then call progress analysis will operate in basic mode only for that channel.

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

Call progress analysis allows the application to detect three different types of dial tone, two busy signals, ringback, and two fax or modem tones on the channel. It is also capable of distinguishing between a live voice and an answering machine when a call is connected. Parameters for these capabilities are downloaded to the channel when **dx_initcallp()** is called.

The voice driver comes equipped with useful default definitions for each of the signals mentioned above. The application can change these definitions through the **dx_chgdur()**, **dx_chgfreq()**, and **dx_chgrepcnt()** functions. The **dx_initcallp()** function takes whatever definitions are currently in force and uses these definitions to initialize the specified channel.

Once a channel is initialized with the current tone definitions, these definitions cannot be changed for that channel without deleting all tones (via **dx_deltones()**) and re-initializing with another call to **dx_initcallp()**. **dx_deltones()** also disables call progress analysis. Note, however, that **dx_deltones()** will erase all user-defined tones from the channel (including any global tone detection information), and not just the call progress analysis tones.
initialize and activate call progress analysis — dx_initcallp()

- **Cautions**

  When you issue this function, the channel must be idle.

- **Errors**

  For a list of error codes, see the Error Codes chapter.

- **Example**

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

  main()
  { 
    DX_CAP   cap_s;
    int      ddd, car;
    char     *chnam, *dialstrg;
    chnam    = "dxxxB1C1";
    dialstrg = "L1234";

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

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

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

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

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

    /*  
 *  Now enable Enhanced call progress with above changed settings.
 */
    if ( dx_initcallp( ddd ) ) {
      /* handle error */
    }
  }
```
/ *  Set off Hook
 */
if ((dx_sethook( ddd, DX_OFFHOOK, EV_SYNC ) == -1) {
    /* handle error */
}

/*  *  Dial  */
if ((car = dx_dial( ddd, dialstrg,(DX_CAP *)&cap_s, DX_CALLP|EV_SYNC)) == -1) { /* handle error */
    switch( car ) {
    case CR_NODIALTONE:
        printf(" Unable to get dial tone\n");
        break;
    case CR_BUSY:
        printf(" %s engaged\n", dialstrg );
        break;
    case CR_CNCT:
        printf(" Successful connection to %s\n", dialstrg );
        break;
    default:
        break;
    }
    /*  *  Set on Hook  */
*/
    if ((dx_sethook( ddd, DX_ONHOOK, EV_SYNC ) == -1) {
        /* handle error */
    }
    dx_close( ddd );
}

■ See Also

- dx_chgdur()
- dx_chgfreq()
- dx_chgrepcnt()
- dx_deltones()
- dx_TSFStatus()
**dx_libinit( )**

**Name:** dx_libinit ( flags )

**Inputs:**
- unsigned short flags
  - specifies the programming model

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

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

**Category:** Configuration

**Mode:** synchronous

**Platform:** DM3, Springware

---

**Description**

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

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

**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 <dxxlib.h>
#include <dxxxlib.h>
int InitDevices( )
```

---

dx_libinit( ) — initialize the voice library DLL

{
    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( )**

**Name:** `int dx_listen(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`
- `dxxxlib.h`

**Category:** TDM Routing

**Mode:** synchronous

**Platform:** DM3, Springware

---

### 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.
- On DM3 boards, this function is supported in a flexible routing configuration but not a fixed routing configuration. This document assumes that a flexible routing configuration is the configuration of choice. For more information on API restrictions in a fixed routing configuration, see the *Voice API Programming Guide.*
dx_listen() — connect a voice listen channel to TDM bus 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_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
   - **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**
     Windows system error

- **Example**

```c
#include <windows.h>
#include <srllib.h>
#include <dx.dll.h>

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

    if (dx_listen(&chdev, &sc_tsinfo, &scts) != 0)
    {
        switch_handler_error(sc_tsinfo.err_code); /* show error message */
    }
    return 0; /* return */
}
```

connect a voice listen channel to TDM bus time slot — dx_listen()

/* Open board 1 channel 1 device */
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_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

- ag_getxmitslot()
- dt_getxmitslot() in the Digital Network Interface Software Reference
- fx_getxmitslot() in the Fax Software Reference
- dx_unlisten()
- dx_unlistencr()
dx_listenecr( ) — enable echo cancellation resource mode

dx_listenecr( )

Name: int dx_listenecr(chdev, sc_tsinfop)

Inputs:
- int chdev: handle of voice channel device on which echo cancellation is to be performed
- SC_TSINFO *sc_tsinfop: pointer to TDM bus time slot information structure

Returns:
- 0 on success
- -1 on error

Includes:
srllib.h
dxxlib.h

Category: Echo Cancellation Resource
Mode: synchronous
Platform: Springware

Description

The dx_listenecr( ) function enables echo cancellation resource (ECR) mode on a specified voice channel and connects the voice channel to the echo-referenced signal on the specified TDM bus time slot. The TDM bus time slot information is contained in the SC_TSINFO data structure. For more information on this structure, see SC_TSINFO, on page 529.

Note: The ECR functions have been replaced by the continuous speech processing (CSP) API functions. CSP provides enhanced echo cancellation. For more information, see the Continuous Speech Processing API Programming Guide and Continuous Speech Processing API Library Reference.

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

Note: For this function, NLP is enabled by default. If you do not want NLP enabled, use dx_listenecrex() with NLP disabled.

The sc_numts field of the SC_TSINFO structure must be set to 1. The sc_tsarrayp field of the SC_TSINFO structure must be initialized with a pointer to a valid array. The first element of this array must contain a valid TDM bus time-slot number (between 0 and 1023) which was obtained by issuing a call to xx_getxmitslot( ) (where xx_ is ag_, dt_, dx_, fx_, or ms_) or dx_getxmitslotecr( ), depending on the application of the function. Upon return from the dx_listenecr( ) function, the echo canceller of the specified voice channel is connected to the TDM bus time slot specified, and it uses the signal carried on the TDM bus time slot as the echo-reference signal for echo cancellation.
enable echo cancellation resource mode — dx_listenecr()

## Cautions

This function fails when:
- An invalid channel device handle is specified.
- An invalid TDM bus time-slot number is specified.
- The ECR feature is not enabled on the board specified.
- The ECR feature is not supported on the board 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:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDX_BADPARM</td>
<td>Parameter error</td>
</tr>
<tr>
<td>EDX_SH_BADCMD</td>
<td>Function is not supported in current bus configuration</td>
</tr>
<tr>
<td>EDX_SH_BADEXTTS</td>
<td>TDM bus time slot is not supported at current clock rate</td>
</tr>
<tr>
<td>EDX_SH_BADINDEX</td>
<td>Invalid Switch Handler index number</td>
</tr>
<tr>
<td>EDX_SH_BADLCLTS</td>
<td>Invalid channel number</td>
</tr>
<tr>
<td>EDX_SH_BADMODE</td>
<td>Function is not supported in current bus configuration</td>
</tr>
<tr>
<td>EDX_SH_CMDBLOCK</td>
<td>Blocking function is in progress</td>
</tr>
<tr>
<td>EDX_SH_LCLTSCNCT</td>
<td>Channel is already connected to TDM bus</td>
</tr>
<tr>
<td>EDX_SH_LIBBSY</td>
<td>Switch Handler library is busy</td>
</tr>
<tr>
<td>EDX_SH_LIBNOTINIT</td>
<td>Switch Handler library is uninitialized</td>
</tr>
<tr>
<td>EDX_SH_MISSING</td>
<td>Switch Handler is not present</td>
</tr>
<tr>
<td>EDX_SH_NOCLK</td>
<td>Switch Handler clock fallback failed</td>
</tr>
<tr>
<td>EDX_SYSTEM</td>
<td>Error from operating system; use dx_fileerrno() to obtain error value</td>
</tr>
</tbody>
</table>
## dx_listenevr( ) — enable echo cancellation resource mode

### Example

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

main()
{
    int msdev1, chdev2;        /* MSI/SC Station, and Voice Channel device handles */
    SC_TSINFO sc_tsinfo;      /* Time slot information structure */
    long scts;                 /* TDM bus time slot */

    /* Open MSI/SC board 1 station 1 device */
    if ((msdev1 = ms_open("msiB1C1", 0)) == -1) {
        /* Perform system error processing */
        exit(1);
    }

    /* Open board 1 channel 2 device */
    if ((chdev2 = dx_open("dxxxB1C2", 0)) == -1) {
        /* Perform system error processing */
        exit(1);
    }

    /* 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 MSI/SC station 1 on board 1 */
    if (ms_getxmitslot(msdev1, &sc_tsinfo) == -1) {
        printf("Error message = %s", ATDV_ERRMSGP(msdev1));
        exit(1);
    }

    /* Connect the echo-reference receive of voice channel 2 on board 1 to
    the transmit signal of msdev1 */
    if (dx_listenevr(chdev2, &sc_tsinfo) == -1) {
        printf("Error message = %s", ATDV_ERRMSGP(chdev2));
        exit(1);
    }

    /* Continue
    .
    .
    */
    /* Then perform xx_unlisten()s and dx_unlistenevr(), plus all xx_close()s */
    return(0);
}
```

### See Also
- `dx_getxmitslotecr( )`
- `dx_listen( )`
- `dx_listenevre( )`
- `dx_unlistenevr( )`
- `xx_getxmitslot( )`, where xx refers to the type of device such as ag_ (analog), dt_ (digital network interface), dx_ (voice), fx_ (fax), and ms_ (modular station interface)
modify characteristics of the echo canceller — dx_listeneccrex() 

dx_listeneccrex() 

**Name:** `int dx_listeneccrex(chdev, sc_tsinfop, ecrctp)`

**Inputs:**
- `int chdev` • handle of voice channel device on which echo cancellation will be performed
- `SC_TSINFO *sc_tsinfop` • pointer to TDM bus time slot information structure
- `DX_ECRCT void *ecrctp` • pointer to ECR characteristic structure

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

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

**Category:** Echo Cancellation Resource

**Mode:** synchronous

**Platform:** Springware

### Description

The `dx_listeneccrex()` function performs identically to `dx_listeneccr()` and also modifies the characteristics of the echo canceller. The characteristics of the echo canceller can be set using the `DX_ECRCT` structure. For more information on this structure, see `DX_ECRCT`, on page 508.

**Note:** The ECR functions have been replaced by the continuous speech processing (CSP) API functions. CSP provides enhanced echo cancellation. For more information, see the Continuous Speech Processing API Programming Guide and Continuous Speech Processing API Library Reference.

<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>
<tr>
<td><code>ecrctp</code></td>
<td>specifies a pointer to the <code>DX_ECRCT</code> structure cast to a (void *)</td>
</tr>
</tbody>
</table>

One characteristic of the echo canceller that can be set using `dx_listeneccrex()` is non-linear processing (NLP). When NLP is activated, the output of the echo canceller is replaced with an estimate of the background noise. The NLP provides full echo suppression as long as the echo-reference signal contains speech signals and the echo-carrying signal does not. In this case, the echo canceller cancels the echo and maintains the full duplex connection.

**Note:** Disable NLP when using the echo canceller output for voice recognition algorithms as NLP may clip the beginning of speech.

### Cautions

This function fails when:

---

*Voice API for Windows Operating Systems Library Reference — November 2003*
dx_listenecrex( ) — modify characteristics of the echo canceller

- An invalid channel device handle is specified.
- The ECR feature is not enabled on the board specified.
- The ECR feature is not supported on the board specified.
- The characteristic table contains invalid fields.

### 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**
  - Function is not supported in current bus configuration
- **EDX_SH_BADEXTTS**
  - TDM bus time slot is not supported at current clock rate
- **EDX_SH_BADINDEX**
  - Invalid Switch Handler index number
- **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 function 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; use `dx_fileerrno()` to obtain error value
modify characteristics of the echo canceller — \texttt{dx\_listenecrex()}

### Example

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

main()
{
    int       msdev1, chdev2;   /* MSI/SC Station and Voice Channel device handles */
    SC_TSINFO sc_tsinfo;        /* Time slot information structure */
    DX_ECRCT  dx_ecrct;         /* ECR Characteristic Table */
    long      scts;             /* TDM bus time slot */

    /* Open MSI/SC board 1 station 1 device */
    if ((msdev1 = ms_open("msiB1C1", 0)) == -1) {
        /* Perform system error processing */
        exit(1);
    }

    /* Open board 1 channel 2 device */
    if ((chdev2 = dx_open("dxxxB1C2", 0)) == -1) {
        /* Perform system error processing */
        exit(1);
    }

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

    /* Fill in the ECR Characteristic Table : with NLP turned off */
    dx_ecrct.ct_length = size_of_ecr_ct;
    dx_ecrct.ct_NLPflag = ECR_CT_DISABLE;

    /* Get TDM bus time slot connected to transmit of MSI/SC station 1 on board 1 */
    if (ms_getxmitslot(msdev1, &sc_tsinfo) == -1) {
        printf("Error message = %s", ATDV_ERRMSGP(msdev1));
        exit(1);
    }

    /* Connect the echo-reference receive of voice channel 2 on board 1 to
    the transmit signal of msdev1 */
    if (dx_listenecrex(chdev2, &sc_tsinfo, (void *)&dx_ecrct) == -1) {
        printf("Error message = %s", ATDV_ERRMSGP(chdev2));
        exit(1);
    }

    /* Continue... */

    return(0);
}
```

### See Also
- \texttt{dx\_listenecr()}
dx_mreciottdata( ) — record voice data from two TDM bus time slots

The **dx_mreciottdata( )** function records voice data from two TDM bus time slots. The data may be recorded to a combination of data files, memory or custom devices.

This function is used for the transaction record feature, which allows you to record two TDM bus time slots from a single channel. Voice activity on two channels can be summed and stored in a single file, device, and/or memory.

**Note:** This function is not supported on HDSI (High Density Station Interface) products, because HDSI products do not support routable voice resources.
record voice data from two TDM bus time slots — dx_mreciottdata()

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>devd</td>
<td>specifies the valid channel device handle on which the recording is to occur. The channel descriptor may be that associated with either of the two TDM bus transmit time slots or a third device also connected to the TDM bus.</td>
</tr>
<tr>
<td>iotp</td>
<td>points to the I/O Transfer Table Structure, DX_IOTT, which specifies the order of recording and the location of voice data. For more information on this structure, see DX_IOTT, on page 509.</td>
</tr>
<tr>
<td>tptp</td>
<td>points to the Termination Parameter Table Structure, DV_TPT, which specifies the termination conditions for recording. For more information on this structure, see DV_TPT, on page 485.</td>
</tr>
<tr>
<td>xpb</td>
<td>points to a DX_XPB structure, which specifies the file format, data format, sampling rate, and resolution for I/O data transfer. For more information on this structure, see DX_XPB, on page 520.</td>
</tr>
</tbody>
</table>
| mode      | specifies the attributes of the recording mode. One or more of the following values can be specified:  
  • 0 – standard play mode  
  • RM_TONE – transmits a 200 msec tone before initiating record. If this mode is not selected, no tone is transmitted (default). |
| sc_tsinfop | points to the SC_TSINFO structure and specifies the TDM bus transmit time slot values of the two time slots being recorded. In the SC_TSINFO structure, sc_numts should be set to 2 for channel recording and sc_tsarrayp should point to an array of two long integers, specifying the two TDM bus transmit time slots from which to record. |

**Note:** When using RM_TONE bit for tone-initiated record, each time slot must be “listening” to the transmit time slot of the recording channel; the alert tone can only be transmitted on the recording channel’s transmit time slot.

After `dx_mreciottdata()` is called, recording continues until one of the following occurs:

• `dx_stopch()` is called on the channel whose device handle is specified in the `devd` parameter
• the data requirements specified in the DX_IOTT structure are fulfilled
• one of the conditions for termination specified in the DV_TPT structure is satisfied

### Cautions

• All files specified in the DX_IOTT structure are of the file format specified in DX_XPB.
• All files recorded will have the same data encoding and rate as DX_XPB.
• When recording VOX files, the data format is specified in DX_XPB rather than through the `dx_setparm()` function.
• Voice data files that are specified in the DX_IOTT structure must be opened with the O_BINARY flag.
• When using MSI/SC stations for transaction recording, make sure a full duplex connection is established. You must issue an `ms_listen()` even though the MSI/SC station is used only for transmitting.
On Springware boards, because the DSP sums the PCM values of the two TDM bus time slots before processing them during transaction recording, all voice-related terminating conditions or features such as DTMF detection, Automatic Gain Control (AGC), and sample rate change will apply to both time slots. In other words, for terminating conditions specified by a DTMF digit, either time slot containing the DTMF digit will stop the recording. Also, maximum silence length requires simultaneous silence from both time slots to meet the specification.

If both time slots transmit a DTMF digit at the same time, the recording will contain an unintelligible result.

Since this API uses `dx_listen()` to connect the channel to the first specified time slot, any error returned from `dx_listen()` will terminate the API with the error indicated.

The API will connect the channel to the time slot specified in the SC_TSINFO data structure `sc_tsarray[0]` field and remain connected after the function has been completed. Both `sc_tsarray[0]` and `sc_tsarray[1]` must be within the range 0 to 1023. No checking is done to verify that `sc_tsarray[0]` or `sc_tsarray[1]` has been connected to a valid channel.

Upon termination of the `dx_mreciottdata()` function, the recording channel continues to listen to the first time slot (pointed to by `sc_tsarray[0]`).

The application should check for a TDX_RECORD event with T_STOP event data after executing a `dx_stopch()` function during normal and transaction recording. This will ensure that all data is written to the disk.

On Springware boards, the recording channel can only detect a loop current drop on a physical analog front end that is associated with that channel. If you have a configuration where the recording channel is not listening to its corresponding front end, you will have to design the application to detect the loop current drop and issue a `dx_stopch()` to the recording device. The recording channel hook state should be off-hook while the recording is in progress.

The transaction record feature may not detect a DTMF digit over a dial tone.

When using `dx_mreciottdata()` and a dial tone is present on one of the time slots, digits will not be detected until dial tone is no longer present. This is because the DSP cannot determine the difference between dial tone and DTMF tones.

On DM3 boards, tone termination conditions such as DTMF and TONE apply only to the primary input of the function; that is, the TDM time slot specified in the SC_TSINFO data structure `sc_tsarray[0]` field.

### 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 handle

- **EDX_BADIOTT**
  - Invalid DX_IOTT entry

- **EDX_BADPARM**
  - Invalid parameter passed

- **EDX_BADTPT**
  - Invalid DV_TPT entry
**EDX_BUSY**

Busy executing I/O function

**EDX_SYSTEM**

Error from operating system; use `dx_fileerrno()` to obtain error value

---

### Example

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

#define MAXLEN 10000

main()
{
    int devh1, devh2, devh3;
    short fd;
    DV_TPT tpt;
    DX_IOTT iott[2];
    DX_XPB xpb;
    SC_TSINFO tsinfo;
    long scts;
    long tslots[32];
    char basebufp[MAXLEN];

    /* open two voice channels */
    if ((devh1 = dx_open("dxxxB1C1", NULL)) == -1) {
        printf("Could not open dxxxB1C1\n");
        exit (1);
    }

    if ((devh2 = dx_open("dxxxB1C2", NULL)) == -1) {
        printf("Could not open dxxxB1C2\n");
        exit (1);
    }

    if ((devh3 = dx_open("dxxxB1C3", NULL)) == -1) {
        printf("Could not open dxxxB1C3\n");
        exit (1);
    }

    if ((fd = dx_fileopen("file.vox", O_CREAT | O_RDWR | O_BINARY)) == -1){
        printf("File open error\n");
        exit (1);
    }

    /* Get channels' external time slots
    * and fill in tslots[] array
    */
    tsinfo.sc_numts = 1;
    tsinfo.sc_tsarrayp = &scts;

    if ((dx_getxmitslot (devh1, &tsinfo) == -1 ){
        /* Handle error */
    }
    tslots[0] = scts;

    if ((dx_getxmitslot (devh2, &tsinfo) == -1 ){
        /* Handle error */
    }
    tslots[1] = scts;
```
/* Set up SC_TSINFO structure */
tsinfo.sc_nums = 2;
tsinfo.sc_tsarrayp = &tslots[0];

/* Set up DX_XPB structure */
xpb.wFileFormat = FILE_FORMAT_VOX;
xpb.wDataFormat = 0;
xpb.nSamplesPerSec = 0L;
xpb.wBitsPerSample = 0;

/* Set up DX_TPT structure */
dx_clrtpt (&tpt,1);
tpt.tp_type = IO_EOT;
tpt.tp_termno = DX_MAXDTMF;
tpt.tp_length = 1;
tpt.tp_flags = TF_MAXDTMF;

/* Set up DX_IOTT structure */
iott[0].io_fhandle = fd;
iott[0].io_type = IO_DEV;
iott[0].io_offset = 0;
iott[0].io_length = MAXLEN;
iott[0].io_offset = IO_EOT;

/* And record from both voice channels */
if (dx_mreciottdata(devh3, &iott[0], &tpt, &xpb, RM_TONE, &tsinfo) == -1) {
    printf("Error recording from dxxxB1C1 and dxxxB1C2\n\n");
    printf("error = %s\n", ATDV_ERRMSGP(devh1));
    exit(2);
}

/* Display termination condition value */
printf("The termination value = %d\n", ATDX_TERMMSK(devh1));

/* And close three voice channels */
if (dx_close(devh3) == -1) {
    printf("Error closing devh3\n\n");
    /* Perform system error processing */
    exit(3);
}

if (dx_close(devh2) == -1) {
    printf("Error closing devh2\n\n");
    /* Perform system error processing */
    exit(3);
}

if (dx_close(devh1) == -1) {
    printf("Error closing devh1\n\n");
    /* Perform system error processing */
    exit(3);
}

if (dx_fileclose(fd) == -1) {
    printf("File close error\n\n");
    exit(1);
}
/* And finish */
return;

See Also

- dx_rec()
- dx_play()
record voice data from two TDM bus time slots — dx_mreciottdata()
**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

**Platform:** DM3, Springware

---

**Description**

The `dx_open( )` function opens a voice board device, channel device, or physical board 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. A device can be opened more than once by any number of processes.

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.

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, including a list of all physical boards, all virtual boards on a physical board, and all subdevices on a virtual board. 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: dxxxxB1, dxxxxB2, and so on. The standard channel device naming convention for voice devices is: dxxxxB1C1, dxxxxB1C2, and so on. On DM3 boards, if issuing this function for cached prompt management, then this parameter points to a physical board device. The physical board device naming convention is: brdB1, brdB2, and so on. For more information on cached prompt management, see the <em>Voice API Programming Guide</em>.</td>
</tr>
<tr>
<td>oflags</td>
<td>reserved for future use. Set this parameter to 0.</td>
</tr>
</tbody>
</table>
open a voice device and return a unique device handle — dx_open

- **Cautions**
  
  - Do not use the Windows `open()` function to open a voice device. Unpredictable results will occur.
  
  - 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.

- **Errors**

  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.

- **Example 1**

  This example illustrates how to open a channel device.

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

  main()
  {
    int chdev;       /* channel descriptor */
    .
    .
    .

    /* Open Channel */
    if ((chdev = dx_open("dxxxB1C1",0)) == -1) {
      /* process error */
    }
    .
    .
  }
  ```

- **Example 2**

  This example illustrates how to open a physical board device when using cached prompts.

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

  main()
  {
    int brdhdl;     /* board handle */
    .
    .
    .

    /* Open board */
    if ((brdhdl = dx_open("brdB1",0)) == -1) {
      /* process system error */
      exit(1);
    }
    .
    .
  }
dx_open( ) — open a voice device and return a unique device handle

See Also

- dx_close( )
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  
dxxlib.h

**Category:** streaming to board

**Mode:** synchronous

**Platform:** DM3

---

**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.
dx_OpenStreamBuffer( ) — create and initialize a circular stream buffer

### Example

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

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

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

    if (dx_PutStreamData(hBuffer, pData, 1024, STREAM_CONT) < 0)
    {
        printf("Error in dx_PutStreamData \n");
        exit(2);
    }

    if ((vDev = dx_open("dxxxB1C1", 0)) < 0)
    {
        printf("Error opening voice device\n");
        exit(3);
    }

    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_playiottdata(vDev, &iott, NULL, EV_SYNC) < 0)
    {
        printf("Error in dx_play() \n", ATDV_LASTERR(vDev));
    }

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

### See Also

- `dx_CloseStreamBuffer( )`
- `dx_SetWaterMark( )`
dx_pause( )

Name: int dx_pause(chdev)

Inputs: int chdev
        • valid channel device handle

Returns: 0 if success
          -1 if failure

Includes: srllib.h
dxxxlib.h

Category: I/O

Mode: synchronous

Platform: DM3

---

Description

The dx_pause( ) function pauses an on-going play until a subsequent dx_resume( ) function is issued. To stop the paused play, use dx_stopch(). The application will not get an event when dx_pause( ) is issued. This function does not return an error if the channel is already in the requested state. This function returns -1 if no play is in progress on the channel.

You can also pause and resume play using a DTMF digit. For more information, see SV_PAUSE and SV_RESUME in the DX_SVCB data structure and dx_setsvcond( ).

For more information on the pause and resume play feature, 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 dx_open( )</td>
</tr>
</tbody>
</table>

Cautions

None.

Errors

An error of -1 is returned if you issue this function when no play is in progress.

If the function returns -1, use the Standard Runtime Library ATDV_LASTERR( ) standard attribute function to return the error code or ATDV_ERRMSGP( ) to return the descriptive error message. Possible errors for this function include:

EDX_BUSY
    Invalid state. Returned when the function is issued but no play is in progress on the channel.
dx_pause() — pause on-going play

Example

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

main()
{
    int lDevHdl;
    DX_IOTT iott;

    /* Open a voice channel */
    int lDevHdl = dx_open("dxxxB1C1", 0);

    /* Set up DX_IOTT */
    DX_IOTT iott;
    /* Fill in the iott structure for the play */
    .
    .
    .

    /* Start playing a prompt */
    if( dx_playiottdata(lDevHdl, &iott, NULL, NULL, EV_ASYNC) < 0)
    {
        /* process error */
    }

    /* Pause the play */
    if( dx_pause(lDevHdl) < 0 )
    {
        /* process error */
    }
    .
    .
}

See Also

- dx_resume()
**dx_play( )**

**Name:** `int dx_play(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 playing mode bit mask for this play session

**Description**

The `dx_play( )` function plays recorded voice data, which may come from any combination of data files, memory, or custom devices. In the case of DM3 boards, voice data may also come from already downloaded cached prompts.

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_playiotdata( )`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>chdev</code></td>
<td>Specifies the valid channel device handle obtained when the channel was opened using <code>dx_open( )</code>.</td>
</tr>
<tr>
<td><code>iottp</code></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 509, for information about the data structure.</td>
</tr>
</tbody>
</table>
**dx_play( ) — play recorded voice data**

<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 playing. For more information on this structure, see DV_TPT, on page 485.</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.

**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 10).

Choose one only:
- EV_ASYNC – run asynchronously
- EV_SYNC – run synchronously (default)

On DM3 boards, 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.

On Springware boards, 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 (8 bits per sample)
- PM_ALAW – play using A-law
- PM_ADSI – play using the ADSI protocol without an alert tone preceding play. If ADSI protocol mode is selected, it is not necessary to select any other play mode parameters. If ADSI data will be transferred, PM_ADSI should be ORed with the EV_SYNC or EV_ASYNC parameter in the mode parameter.
- PM_ADSIALERT – play using the ADSI protocol with an alert tone preceding play. If ADSI protocol mode is selected, it is not necessary to select any other play mode parameters. PM_ADSIALERT should be ORed with the EV_SYNC or EV_ASYNC parameter in the mode parameter.
- PM_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.

4. dx_play() will run synchronously if you do not specify EV_ASYNC, or if you specify EV_SYNC (default).

5. Intel® telecom boards enable you to select either A-law or mu-law encoding of data. The default on the board is set to mu-law and returns to mu-law after each play. The A-law parameters must be passed each time the play function is called.

Table 10 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 10. 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_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>No Tone</td>
<td>PM_SR6</td>
<td>PM_SR8</td>
<td>PM_SR6</td>
<td>PM_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 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 speed and volume is based on the most recent adjustment made using `dx_adjsv( )` or `dx_setsvcond( )`.
- Intel® telecom boards enable you to select either A-law or mu-law encoding of data. The default on the board is set to mu-law and returns to mu-law after each play. The A-law parameters must be passed each time the play function is called.

### 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; use `dx_fileerrno( )` to obtain error value

### Example 1

This example illustrates how to use `dx_play( )` in synchronous mode.
/* Play a voice file. Terminate on receiving 4 digits or at end of file*/
#include <fcntl.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

main()
{
    int chdev;
    DX_IOTT iott;
    DV_TPT tpt;
    DV_DIGIT dig;

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

    /* set up DX_IOTT */
    iott.io_type = IO_DEV|IO_EOT;
    iott.io_bufp = 0;
    iott.io_offset = 0;
    iott.io_length = -1; /* play till end of file */
    if((iott.io_fhandle = dx_fileopen("prompt.vox", O_RDONLY|O_BINARY)) == -1)  {
        /* process error */
    }

    /* set up DV_TPT */
    dx_clrtpt(&tpt,1);
    tpt.tp_type   = IO_EOT;          /* only entry in the table */
    tpt.tp_termno = DX_MAXDTMF;      /* Maximum digits */
    tpt.tp_length = 4;               /* terminate on four digits */
    tpt.tp_flags  = TF_MAXDTMF;      /* Use the default flags */

    /* clear previously entered digits */
    if (dx_clrdigbuf(chdev) == -1) { /* process error */
        
    }

    /* Now play the file */
    if (dx_play(chdev,&iott,&tpt,EV_SYNC) == -1) { /* process error */
        
    } /* get digit using dx_getdig() and continue processing. */

} /* Example 2
This example illustrates how to use dx_play() in asynchronous mode.
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>
dx_play( ) — play recorded voice data

#define MAXCHAN 24

int play_handler();
DX_IOTT prompt[MAXCHAN];
DV_TPT tpt;
DV_DIGIT dig;

main()
{
    int chdev[MAXCHAN], index, index1;
    char *chname;
    int i, srlmode, voxfd;

    /* Set SRL to run in polled mode. */
    srlmode = SR_POLLMODE;
    if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1) {
        /* process error */
    }

    /* initialize all the DX_IOTT structures for each individual prompt */
    .
    .

    /* Open the vox file to play; the file descriptor will be used
     * by all channels.
     */
    if ((voxfd = dx_fileopen("prompt.vox", O_RDONLY|O_BINARY)) == -1) {
        /* process error */
    }

    /* For each channel, open the device using dx_open(), set up a DX_IOTT
     * structure for each channel, and issue dx_play() in asynchronous mode. */
    for (i=0; i<MAXCHAN; i++) {
        /* Set chname to the channel name, e.g., dxxxB1C1, dxxxB1C2,... */
        /* Open the device using dx_open().  chdev[i] has channel device
         * descriptor.
         */
        if ((chdev[i] = dx_open(chname,NULL)) == -1) {
            /* process error */
        }

        /* Use sr_enbhdlr() to set up handler function to handle play
         * completion events on this channel.
         */
        if (sr_enbhdlr(chdev[i], TDX_PLAY, play_handler) == -1) {
            /* process error */
        }

        /* Set the DV_TPT structures up for MAXDTMF. Play until one digit is
         * pressed or the file is played.
         */
        dx_clrtpt(&tpt,1);
        tpt.tp_type = IO_EOT; /* only entry in the table */
        tpt.tp_termno = DX_MAXDTMF; /* Maximum digits */
        tpt.tp_length = 1; /* terminate on the first digit */
        tpt.tp_flags = TF_MAXDTMF; /* Use the default flags */
        prompt[i].io_type = IO_DEV|IO_EOT; /* play from file */
        prompt[i].io_type = IO_DEV|IO_BUF; /* play from file */
        prompt[i].io_bufp = 0;
        prompt[i].io_bufp += -1; /* play till end of file */
        prompt[i].io_nextp = NULL;
        prompt[i].io_fhandle = voxfd;
    }
}
/* play the data */
if (dx_play(chdev[i],&prompt[i],&tpt,EV_ASYNC) == -1) {
    /* process error */
}

Use sr_waitevt to wait for the completion of dx_play().
* On receiving the completion event, TDX_PLAY, control is transferred
* to the handler function previously established using sr_enbhdlr().
*/

int play_handler()
{
    long term;
    /* Use ATDX_TERMMSK() to get the reason for termination. */
    term = ATDX_TERMMSK(sr_getevtdev());
    if (term & TM_MAXDTMF) {
        printf("play terminated on receiving DTMF digit(s)\n");
    } else if (term & TM_EOD) {
        printf("play terminated on reaching end of data\n");
    } else {
        printf("Unknown termination reason: \x\n", term);
    }

    /* Kick off next function in the state machine model. */
    return 0;
}

Example 3

This example illustrates how to define and play an alert tone, receive acknowledgement of the alert
tone, and use dx_play() to transfer ADSI data.

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

int parm,
DV_TPT tpt[2];
DV_DIGIT digit;
TN_GEN tngen;
DX_IOTT iott;

main(argc,argv)
int argc;
char* argv[];
{
    int chfd;
    char channname[12];
    parm = SR_POLLMODE;
    sr_setparm(SRL_DEVICE, SR_MODEID, &parm);

    /*
    * Open the channel using the command line arguments as input
    */
    sprintf(channname, "\s\s", argv[1],argv[2]);
    if ((chfd = dx_open(channname, NULL)) == -1) {
        printf("Board open failed on device \s\n", channname);
### See Also

- `dx_playf()`
- `dx_playiotdata()`
- `dx_playvox()`
- `dx_setparm()`, `dx_getparm()`
- `dx_adjsv()` (for speed or volume control)
- `dx_setsvcond()` (for speed or volume control)
- `DX_IOTT` data structure (to identify source or destination of the voice data)
- event management functions in *Standard Runtime Library API Library Reference*
- `ATDX_TERMMMSK()`
- `DV_TPT` data structure (to specify a termination condition)
- `dx_setuio()`
**dx_playf( )**

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

**Inputs:**
- 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:** srlib.h
dxxxlib.h

**Category:** I/O Convenience

**Mode:** synchronous

**Platform:** DM3, Springware

---

**Description**

*dx_playf( )* is a convenience function that synchronously plays voice data or transfers ADSI data (using the ADSI protocol) from a single file.

**Note:** Although this function can be used for transmitting ADSI data, the *dx_RxIottData( ), dx_TxIottData( ),* and *dx_TxRxIottData( )* functions are recommended as the preferred method.

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 <em>dx_open( )</em></td>
</tr>
<tr>
<td>fnamep</td>
<td>points to the 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 485.</td>
</tr>
<tr>
<td>mode</td>
<td>specifies the mode. This function supports EV_SYNC (synchronous mode) only.</td>
</tr>
</tbody>
</table>

**Cautions**

None.
dx_playf( ) — synchronously play voice data

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; use dx_fileerrno( ) to obtain error value

Source Code

/***************************************************************************/
/* NAME: int dx_playf(devd,filep,tptp,mode) */
/* DESCRIPTION: This function opens and plays a */
/* named file. */
/* INPUTS: devd - channel descriptor */
/* tptp - pointer to the termination control block */
/* filep - pointer to file name */
/* OUTPUTS: Data is played. */
/* RETURNS: 0 - success -1 - failure */
/* CALLS: open() dx_play() close() */
/* CAUTIONS: none. */
***************************************************************************/

int dx_playf(devd,filep,tptp,mode)
int     devd;
char    *filep;
DV_TPT  *tptp;
USHORT  mode;
{
    DX_IOTT iott;
    int     rval;

    /*
    * If Async then return Error
    * Reason: IOTT's must be in scope for the duration of the play
    */
    if ( mode & EV_ASYNC ) {
        return( -1 );
    }

    /* Open the File */
    if ((iott.io_fhandle = open(filep,0_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;

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

Example

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

main()
{
    int chdev;
    DV_TPT tpt[2];
    /* Open the channel using dx_open( ). Get channel device descriptor in */
    /* chdev. */
    if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
        /* process error */
    }
    /* Set up the DV_TPT structures for MAXDTMF. Play until one digit is */
    /* pressed or the file has completed play */
    dx_clrtpt(tpt,1);
    tpt[0].tp_type   = IO_EOT;      /* only entry in the table */
    tpt[0].tp_termno = DX_MAXDTMF;  /* Maximum digits */
    tpt[0].tp_length = 1;           /* terminate on the first digit */
    tpt[0].tp_flags  = TF_MAXDTMF;  /* Use the default flags */
    if (dx_playf(chdev,"weather.vox",tpt,EV_SYNC) == -1) {
        /* process error */
    }
}

See Also

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

**Name:** short dx_playiottdata(chdev, iottp, tptp, xpbp, mode)

**Inputs:**
- int chdev • valid channel device handle
- DX_IOTT *iottp • pointer to I/O Transfer Table
- DV_TPT *tptp • pointer to Termination Parameter Block
- DX_XPB *xpbp • pointer to I/O Transfer Parameter Block
- unsigned short mode • play mode

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

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

**Category:** I/O

**Mode:** asynchronous or synchronous

**Platform:** DM3, Springware

#### Description

The `dx_playiottdata( )` function plays back recorded voice data, which may come from any combination of data files, memory, or custom devices. In the case of DM3 boards, voice data may also come from already downloaded cached prompts.

The file format for the files to be played is specified in the **wFileFormat** field of the **DX_XPB**. Other fields in the DX_XPB describe the data format. For files that include data format information (for example, WAVE files), these other fields are ignored.

The `dx_playiottdata( )` function is similar to `dx_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.
## Parameters and Description

<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()`.
| iottp     | 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 509, 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 iottp. |
| tptp      | 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 485. |
| xpbp      | Points to the I/O Transfer Parameter Block, DX_XPB. The file format for the files to be played is specified in the `wFileFormat` field of the DX_XPB. Other fields in the DX_XPB describe the data format. For more information about this structure, see the description for DX_XPB, on page 520. For information about supported data formats, see the Voice API Programming Guide. |
| mode      | Specifies the play mode and synchronous/asynchronous mode. For a list of all valid values, see the `dx_play()` function description. • PM_TONE – play 200 msec audible tone • EV_SYNC – synchronous mode • EV_ASYNC – asynchronous mode |

### 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.
- 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 520 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, one of the following error codes will be returned by ATDV_LASTERR():

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; use dx_fileerrno() to obtain error value

EDX_XPBPARM
   Invalid DX_XPB setting

Example 1

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");
        /* 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;

    /* Open VOX file to play */
    if ((fd = dx_fileopen("HELLO.VOX",O_RDONLY|O_BINARY)) == -1) {
        printf("File open error\n");
        exit(2);
    }
```
play back recorded voice data from multiple sources — dx_playioData( )

/* Set up DX_IOTT */
iott.io_fhandle = fd;
iott.ioBufp = 0;
iott.io_offset = 0;
iott.io_length = -1;
iott.io_type = IO_DEV | IO_EOF;

/* 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_wring(chdev,1,DX_OFFHOOK,-1) == -1) {
    printf("Error waiting for ring - %s\n", ATDV_LASTERR(chdev));
    exit(3);
}

/* Start playback */
if (dx_playioData(chdev,&iott,&tpt,&xpb,EV_SYNC)==-1) {
    printf("Error playing file - %s\n", ATDV_ERRMSGP(chdev));
    exit(4);
}

Example 2

This example illustrates how to play back a cached prompt (VOX file) that has already been downloaded to on-board memory.

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

main()
{
    int chdev; /* channel descriptor */
    int brdhdl; /* board handle */
    int promptHandle; /* Handle of the prompt to be downloaded */
    int fd; /* file descriptor for file to be played */
    DX_IOTT iott; /* I/O transfer table for the play operation*/
    DX_IOTT iottp; /* I/O transfer table for the downloaded cache prompt*/
    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);
}

/* Open board */
if ((brdhdl = dx_open("brdB1",0)) == -1) {
    printf("Cannot open board\n");
    /* Perform system error processing */
    exit(1);
}
/ * Open VOX file to cache */
  if ((fd = dx_fileopen("HELLO.VOX",O_RDONLY|O_BINARY)) == -1) {
    printf("File open error\n"); 
    exit(2);
  }

  /* This specifies the data to cache */
  iottp.io_fhandle = fd;
  iottp.io_bufp = 0;
  iottp.io_offset = 0;
  iottp.io_length = -1;
  iottp.io_type = IO_DEV | IO_EOT;

  /* Download a prompt to the on-board memory */
  if (dx_cacheprompt(brdhdl, &iottp, &promptHandle, EV_SYNC) == -1 {
    printf("dx_cacheprompt error\n");
    exit(3);
  }

  /* 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;

  /*This block specifies the downloaded cache prompt */
  iott.io_type = IO_CACHED | IO_EOT;
  iott.io_fhandle = promptHandle;
  iott.io_offset = 0;
  iott.io_length = -1;

  // Specify VOX file format for ADPCM at 8KHz
  xpb.wFileFormat = FILE_FORMAT_VOX;
  xpb.wDataFormat = DATA_FORMAT_DIALOGIC_ADPCM;
  xpb.wBitsPerSample = 4;

  /* Clear digit buffer */
  dx_clrdigbuf(chdev);

  /* Start playback */
  if (dx_playiottdata(chdev, &iott, &tpt, &xpb, EV_SYNC) == -1) {
    printf("Error playing file - %s\n", ATDV_ERRMSGP(chdev));
    exit(4);
  }

dx_fileclose(fd);
  dx_close(brdhdl);
  dx_close(chdev);
}

- **See Also**
  - dx_play()
  - dx_playf()
  - dx_playwav()
  - dx_playvox()
  - dx_setuio()
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

**Platform:** DM3, Springware

---

### 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 <code>TN_GEN</code> structure, which defines the frequency, amplitude, and duration of a single- or dual-frequency tone. For more information, see <code>TN_GEN</code>, on page 530. You can use the <code>dx_bldtngen()</code> function to set up the structure.</td>
</tr>
<tr>
<td>tptp</td>
<td>points to the <code>DV_TPT</code> data structure, which specifies a terminating condition for this function. For more information, see <code>DV_TPT</code>, on page 485.</td>
</tr>
<tr>
<td>mode</td>
<td>specifies whether to run this function asynchronously or synchronously. Set to one of the following:</td>
</tr>
<tr>
<td></td>
<td>• EV_ASYNC – run <code>dx_playtone()</code> asynchronously</td>
</tr>
<tr>
<td></td>
<td>• EV_SYNC – run <code>dx_playtone()</code> synchronously (default)</td>
</tr>
</tbody>
</table>

### 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.
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.
- On DM3 boards, 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; use `dx_fileerrno()` to obtain error value
Example

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

#define TID_1   101

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

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

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

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

    /*
     * Enable Detection of ToneId TID_1
     */
    if ( dx_enbtone( dxxxdev, TID_1 | DM_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;
```
dx_playtone( ) — play tone defined by TN_GEN structure

```c
if ( dx_playtone( dxxxdev, &tngen, tpt, EV_SYNC ) == -1 ) {
    printf( "Unable to Play the Tone\n" );
    printf( "Last error = %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_bldtngen( )`
- 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( )**

**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
dxxlib.h

**Category:** Global Tone Generation

**Mode:** asynchronous or synchronous

**Platform:** DM3, Springware

---

**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( )`. Be aware that on DM3 boards, valid values are for the cycles field of TN_GENCAD is 1 to 40 cycles. On Springware boards, valid values are from 1 to 255 (255 = infinite repetitions).
dx_playtoneEx() — play the cadenced tone defined by TN_GENCAD

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.
play the cadenced tone defined by TN_GENCAD — dx_playtoneEx() 

- 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.
- On DM3 boards, 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_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 tg_dflag field in TN_GEN structure
- **EDX_FREQGEN**
  - Invalid frequency component in TN_GEN structure
- **EDX_SYSTEM**
  - Error from operating system; use dx_fileerrno() to obtain error value

### Example

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

#include <stdio.h>
#include <windows.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 );
    }
```
dx_playtoneEx( ) — play the cadenced tone defined by TN_GENCAD

/*
 * 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_EOT;
tpt[1].tp_termno = DX_MAXTIME; /* On DM3 boards, 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 supported on Springware but not on DM3 boards.
 * 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
 */
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 cadence tone.
 */
if ( dx_playtoneEx( dxxpdev, &tngencad, tpt, EV_SYNC } == -1 ) { 
  printf( "Unable to Play the Cadenced Tone\n" );
  printf( "Last error = %d  Err Msg = %s\n", ATDV_LASTERR( dxxpdev ), ATDV_ERRMSGP( dxxpdev ) );
dx_close( dxxpdev );
exit( 1 );
}

/*
 * Examine termination reason in bitmap.
 * If time-out caused termination, play reorder tone.
 */
if((term = ATDX_TERMMSK(dxxpdev)) == 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( dxxpdev, CP_REORDER, tpt, EV_SYNC } == -1 ) { 
    printf( "Unable to Play the Cadenced Tone\n" );
    printf( "Last error = %d  Err Msg = %s\n",
play the cadenced tone defined by TN_GENCAD — dx_playtoneEx()
dx_playvox( ) — play voice data stored in a single VOX file

**dx_playvox( )**

**Name:** SHORT dx_playvox(chdev, filenamep, ttp, xpbp, mode)

**Inputs:**
- int chdev • valid channel device handle
- char *filenamep • pointer to name of file to play
- DV_TPT *ttt • 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:**
- srllib.h
- dxxxlib.h

**Category:** I/O Convenience

**Mode:** synchronous

**Platform:** DM3, Springware

### 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>ttt</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 485.</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 520.</td>
</tr>
<tr>
<td>mode</td>
<td>specifies the play mode. The following two values must be ORed together: • PM_TONE – play 200 msec audible tone • EV_SYNC – 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 this function returns -1 to indicate failure, one of the following reasons will be contained by ATDV_LASTERR():

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; use dx_fileerrno() to obtain error value. System I/O errors

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
");
        /* 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);
    }
}

play voice data stored in a single VOX file — dx_playvox()
dx_playvox( ) — play voice data stored in a single VOX file

- See Also
  - dx_play( )
  - dx_playf( )
  - dx_playiottdata( )
  - dx_playwav( )
**dx_playwav( )**

**Name:** SHORT dx_playwav(chdev, filenamep, ttp, mode)

**Inputs:**
- int chdev • valid channel device handle
- char *filenamep • pointer to name of file to play
- DV_TPT *ttp • 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

**Platform:** DM3, Springware

---

**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>ttp</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 485.</td>
</tr>
<tr>
<td>filenamep</td>
<td>points to the name of the file to play</td>
</tr>
<tr>
<td>mode</td>
<td>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)</td>
</tr>
</tbody>
</table>

**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 520 and the Voice API Programming Guide.
dx_playwav() — play voice data stored in a single WAVE file

Errors

If this function returns -1 to indicate failure, one of the following reasons will be contained by ATDV_LASTERR():

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; use dx_fileerrno() to obtain error value

EDX_XPBPARAM
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()

- See Also
  - dx_playiottdat()a
  - dx_playvox()
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

**Platform:** DM3

---

**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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hBuffer</td>
<td>specifies the circular stream buffer handle obtained from <code>dx_OpenStreamBuffer( )</code></td>
</tr>
<tr>
<td>pNewData</td>
<td>a pointer to the user buffer containing data to be placed in the circular stream buffer</td>
</tr>
<tr>
<td>BuffSize</td>
<td>specifies the number of bytes in the user buffer</td>
</tr>
<tr>
<td>flag</td>
<td>a flag indicating whether this is the last block of data in the user buffer. Valid values are:</td>
</tr>
<tr>
<td></td>
<td>• STREAM_CONT – for all buffers except the last one</td>
</tr>
<tr>
<td></td>
<td>• STREAM_EOD – for the last buffer</td>
</tr>
</tbody>
</table>

**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;

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

    if (dx_PutStreamData(hBuffer, pData, 1024, STREAM_CONT) < 0)
    {
        printf("Error in dx_PutStreamData \n");
        exit(2);
    }

    if ((vDev = dx_open("dxxxB1C1", 0)) < 0)
    {
        printf("Error opening voice device\n");
        exit(3);
    }

    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_playiottdata(vDev, &iott, NULL, EV_SYNC) < 0)
    {
        printf("Error in dx_play() \n", ATDV_LASTERR(vDev));
    }

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

**See Also**

- `dx_OpenStreamBuffer()`
dx_PutStreamData() — place data into a circular stream buffer
get tone information for a specific call progress tone — dx_querytone( )

**dx_querytone( )**

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

**Inputs:**
- int brdhdl • a valid physical board level device
- 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
- dxxlib.h

**Category:** Call Progress Analysis

**Mode:** asynchronous or synchronous

**Platform:** DM3

---

**Description**

The **dx_querytone()** function returns tone information for a call progress tone currently available on the physical 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.
dx_querytone( ) — get tone information for a specific call progress tone

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 physical 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.

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

## Errors

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

- **EDX_BADPARM**
  - invalid parameter
- **EDX_SYSTEM**
  - 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( ) — record voice data from a single channel

**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

**Platform:** DM3, Springware

---

**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 <strong>dx_open( )</strong></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 <strong>DX_IOTT</strong>, on page 509, for more information on this data structure.</td>
</tr>
</tbody>
</table>
**record voice data from a single channel — dx_rec( )**

<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 485.</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.

**mode** defines the recording mode. One or more of the values listed below may be selected in the bit mask using bitwise OR (see Table 11 for record mode combinations).

Choose one only:
- EV_ASYNC – run asynchronously
- EV_SYNC – run synchronously (default)

Choose one or more:
- MD_ADPCM – record using Adaptive Differential Pulse Code Modulation encoding algorithm (4 bits per sample). Recording with ADPCM is the default setting.
- 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)
- RM_NOTIFY – generate record notification beep tone

*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. 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).

4. 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( ).

5. 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.
Table 11 shows recording mode selections. The first column of the table lists all possible combinations of record features, and the first row lists each type of encoding algorithm (ADPCM or PCM) and the data-storage rate for each algorithm/sampling rate combination in parenthesis (24 kbps, 32 kbps, 48 kbps, or 64 kbps).

Select the desired record feature in the first column of the table and move across that row until the column containing the desired encoding algorithm and data storage rate is reached. The record modes that must be entered in `dx_rec()` are provided where the features row, and encoding algorithm/data storage rate column intersect. Parameters listed in braces, { }, are default settings and do not have to be specified.

**Table 11. 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>AGC</td>
<td>RM_SR6 (MD_ADPCM)</td>
<td>RM_SR6 (MD_ADPCM)</td>
<td>RM_SR6</td>
<td>RM_SR8</td>
</tr>
<tr>
<td></td>
<td>(MD_GAIN)</td>
<td>(MD_GAIN)</td>
<td>(MD_ALAW)</td>
<td>(MD_PCM)</td>
</tr>
<tr>
<td>No AGC</td>
<td>MD_NOGAIN RM_SR6 (MD_ADPCM)</td>
<td>MD_NOGAIN RM_SR6 (MD_ADPCM)</td>
<td>MD_NOGAIN RM_SR6</td>
<td>MD_NOGAIN RM_SR8</td>
</tr>
<tr>
<td>Tone</td>
<td></td>
<td></td>
<td>(MD_PCM)</td>
<td>(MD_PCM)</td>
</tr>
<tr>
<td></td>
<td>RM_TONE RM_SR6 (MD_ADPCM)</td>
<td>RM_TONE RM_SR6 (MD_ADPCM)</td>
<td>RM_TONE RM_SR6</td>
<td>RM_TONE RM_ALAW</td>
</tr>
<tr>
<td></td>
<td>(MD_GAIN)</td>
<td>(MD_GAIN)</td>
<td>(MD_ALAW)</td>
<td>(MD_PCM)</td>
</tr>
<tr>
<td>No AGC</td>
<td>MD_NOGAIN RM_TONE RM_SR6 (MD_ADPCM)</td>
<td>MD_NOGAIN RM_TONE RM_SR6 (MD_ADPCM)</td>
<td>MD_NOGAIN RM_TONE</td>
<td>MD_NOGAIN RM_ALAW</td>
</tr>
<tr>
<td>Tone</td>
<td></td>
<td></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.

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**

- On DM3 boards using a flexible routing configuration, 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:

- `EXX_BADDEV`  
  Invalid Device Descriptor

- `EXX_BADIOOT`  
  Invalid `DX_IOTT` entry

- `EXX_BADPARM`  
  Invalid parameter

- `EXX_BADTPT`  
  Invalid `DX_TPT` entry

- `EXX_BUSY`  
  Busy executing I/O function

- `EXX_SYSTEM`  
  Error from operating system; use `dx_fileerrno( )` to obtain error value

**Example 1**

This example illustrates how to using `dx_rec( )` in synchronous mode.
dx_rec() — record voice data from a single channel

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

#define MAXLEN 10000

main()
{
  DV_TPT tpt;
  DX_IOTT iott[2];
  int chdev;
  char basebufp[MAXLEN];

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

  /*
   * Set up the DV_TPT structures for MAXDTMF
   */
  dx_clrtpt(&tpt, 1);
  tpt.tp_type   = IO_EOT;          /* last entry in the table */
  tpt.tp_termno = DX_MAXDTMF;      /* Maximum digits */
  tpt.tp_length = 1;               /* terminate on the first digit */
  tpt.tp_flags  = TF_MAXDTMF;      /* Use the default flags */

  /*
   * Set up the DX_IOTT. The application records the voice data to memory
   * allocated by the user.
   */
  iott[0].io_type = IO_MEM | IO_CONT;   /* Record to memory */
  iott[0].io_bufp = basebufp;         /* Set up pointer to buffer */
  iott[0].io_offset = 0;              /* Start at beginning of buffer */
  iott[0].io_length = MAXLEN;         /* Record 10,000 bytes of voice data */
  iott[1].io_type = IO_DEV | IO_EOT;    /* Record to file, last DX_IOTT entry */
  iott[1].io_bufp = 0;                /* Set up pointer to buffer */
  iott[1].io_offset = 0;              /* Start at beginning of buffer */
  iott[1].io_length = MAXLEN;         /* Record 10,000 bytes of voice data */

  if (iott[1].io_fhandle = dx_fileopen("file.vox",
    O_RDWR | O_CREAT | O_TRUNC | O_BINARY, 0666)) == -1) {
    /* process error */
  }

  /* clear previously entered digits */
  if (dx_clrdigbuf(chdev) == -1) {
    /* process error */
  }

  if (dx_rec(chdev, &iott[0], &tpt, RM_TONE | EV_SYNC) == -1) {
    /* process error */
  }

  /* Analyze the data recorded */
  .
  .
}
```

Example 2

This example illustrates how to use `dx_rec()` in asynchronous mode.
#include <stdio.h>
#include <fcntl.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <windows.h>

#define MAXLEN 10000
#define MAXCHAN 24

int record_handler();
DV_TPT tpt;
DX_IOTT iott[MAXCHAN];
int chdev[MAXCHAN];
char basebufp[MAXCHAN][MAXLEN];

main()
{
    int i, srlmode;
    char *chname;
    /* Set SRL to run in polled mode. */
    srlmode = SR_POLLMODE;
    if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1) {
        /* process error */
    }

    /* Start asynchronous dx_rec() on all the channels. */
    for (i=0; i<MAXCHAN; i++) {
        /* Set chname to the channel name, e.g., dxxxB1C1, dxxxB1C2,... */
        /* open the channel using dx_open() */
        if ((chdev[i] = dx_open(chname,NULL)) == -1) {
            /* process error */
        }

        /* Using sr_enbhdlr(), set up handler function to handle record
        completion events on this channel. */
        if (sr_enbhdlr(chdev[i], TDX_RECORD, record_handler) == -1) {
            /* process error */
        }

        /* Set up the DV_TPT structures for MAXDTMF */
        dx_clrtpt(&tpt,1);
        tpt.tp_type = IO_EOT;          /* last entry in the table */
        tpt.tp_termno = DX_MAXDTMF;    /* Maximum digits */
        tpt.tp_length = 1;            /* terminate on the first digit */
        tpt.tp_flags = TF_MAXDTMF;     /* Use the default flags */

        /* Set up the DX_IOTT. The application records the voice data to memory
        allocated by the user. */
        iott[i].io_type = IO_MEM|IO_EOT;  /* Record to memory, last DX_IOTT
                                            entry */
        iott[i].io_bufp = basebufp[i];   /* Set up pointer to buffer */
        iott[i].io_offset = 0;           /* Start at beginning of buffer */
        iott[i].io_length = MAXLEN;      /* Record 10,000 bytes voice data */

        /* clear previously entered digits */
        if (dx_clrdigbuf(chdev) == -1) {
            /* process error */
        }
    }
}
dx_rec() — record voice data from a single channel

/* Start asynchronous dx_rec() on the channel */
if (dx_rec(chdev[i], &iott[i], &tpt, RM_TONE | EV_ASYNC) == -1) {
    /* process error */
}

/* Use sr_waitevt to wait for the completion of dx_rec().
* On receiving the completion event, TDX_RECORD, control is transferred
* to a handler function previously established using sr_enbhdlr().
*/
int record_handler()
{
    long term;
    /* Use ATDX_TERMMSK() to get the reason for termination. */
    term = ATDX_TERMMSK(sr_getevtdev());
    if (term & TM_MAXDTMF) {
        printf("record terminated on receiving DTMF digit(s)\n");
    } else if (term & TM_NORMTERM) {
        printf("normal termination of dx_rec()\n");
    } else {
        printf("Unknown termination reason: %x\n", term);
    }
    /* Kick off next function in the state machine model. */
    return 0;
}

See Also

- dx_recf()
- dx_reciottdata()
- dx_recm()
- dx_recmf()
- 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()
dx_recf( )

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

**Inputs:**
- int chdev • valid channel device handle
- char *fnamep • pointer to file to record to
- DV_TPT *tptp • pointer to Termination Parameter Table structure
- unsigned short mode • recording mode bit mask for this record session

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

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

**Category:** I/O Convenience

**Mode:** synchronous

**Platform:** DM3, Springware

---

### 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`.

**Parameter** | **Description**
--- | ---
chdev | specifies the valid channel device handle obtained when the channel was opened using `dx_open()`
fnamep | points to the file from which voice data will be recorded
tptp | 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 485.
mode | defines the recording mode. One or more of the values listed in the `mode` description of `dx_rec()` may be selected in the bitmask using bitwise OR (see Table 11, “Record Mode Selections”, on page 338 for record mode combinations).

### 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_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; use dx_fileerrno() to obtain error value

Source Code

```c
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;
    }
```
/* Use dx_rec() to do the record */
iott.io_type = IO_EOT | IO_DEV;
iott.io_offset = (long)0;
iott.io_length = -1;

rval = dx_rec(devd, &iott, tptp, mode);

if (close(iott.io_fhandle) == -1) {
    return -1;
}

return rval;

Example

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

main()
{
    int chdev;
    long termtype;
    DV_TPT tpt[2];

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

    /* Set the DV_TPT structures up for MAXDTMF and MAXSIL */
    dx_clrtpt(tpt, 2);
    tpt[0].tp_type = IO_CONT;
    tpt[0].tp_termno = DX_MAXDTMF;  /* Maximum digits */
    tpt[0].tp_length = 1;          /* terminate on the first digit */
    tpt[0].tp_flags = TF_MAXDTMF;        /* Use the default flags */

    /*
     * If the initial silence period before the first non-silence period
     * exceeds 4 seconds then terminate. If a silence period after the
     * first non-silence period exceeds 2 seconds then terminate.
     */
    tpt[1].tp_type = IO_EOT;        /* last entry in the table */
    tpt[1].tp_termno = DX_MAXSIL;   /* Maximum silence */
    tpt[1].tp_length = 20;         /* terminate on 2 seconds of
     * continuous silence */
    tpt[1].tp_flags = TF_MAXSIL|TF_SETINIT;  /* Use the default flags and
     * initial silence flag */
    tpt[1].tp_data = 40;           /* Allow 4 seconds of initial
     * silence */

    if (dx_recf(chdev, "weather.vox", tpt, RM_TONE) == -1) {
        /* process error */
    }

    termtype = ATDX_TERMMSK(chdev); /* investigate termination reason */
    if (termtype & TM_MAXDTMF) {
        /* process DTMF termination */
    }
    ...
}
**dx_recf() — record voice data to a single file**

### See Also

- `dx_rec()`
- `dx_recio()`
- `dx_recio_data()`
- `dx_reciof()`
- `dx_reciox()`
- `dx_setparm()`
- `dx_getparm()`
- `ATDX_TERMMSK()`
- `DV_TPT` data structure (to specify a termination condition)
dx_reciottdata()

Name: short 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
Platform: DM3, Springware

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 dx_open()</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 509, 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 485.</td>
</tr>
</tbody>
</table>
dx_recioottdata( ) — record voice data to multiple destinations

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 520.
mode       | Specifies the record mode:
            | • EV_ASYNC – asynchronous mode
            | • EV_SYNC – synchronous mode
            | • PM_TONE – play 200 msec audible tone
            | • RM_NOTIFY – generate record notification beep tone

Cautions

- On High Density Station Interface (HDSI) boards, this function is supported provided that the correct play/record PCD file is downloaded.
- On DM3 boards using a flexible routing configuration, 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.
In synchronous mode, if this function returns -1 to indicate failure, one of the following error codes will be returned by \texttt{ATDV\_LASTERR}():

- \texttt{EDX\_BADIOTT} Invalid \texttt{DX\_IOTT} setting
- \texttt{EDX\_BADWAVFILE} Invalid WAVE file
- \texttt{EDX\_BUSY} Channel is busy
- \texttt{EDX\_SYSTEM} Error from operating system; use \texttt{dx\_fileerrno}() to obtain error value
- \texttt{EDX\_XPBPARM} Invalid \texttt{DX\_XPB} setting
- \texttt{EDX\_SH\_BADCMD} Unsupported command or WAVE file format

\begin{itemize}
\item \textbf{Example}
\end{itemize}

```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");
    /* 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;
  
  /* Open file */
  if ((fd = dx\_fileopen("MESSAGE.VOX",O\_RDWR|O\_BINARY)) == -1) {
    printf("File open error\n");
    exit(2);
  }

  /* Set up DX\_IOTT */
  iott.io_fhandle = fd;
  iott.io_bufp   = 0;
  iott.io_offset = 0;
  iott.io_length = -1;
  iott.io_type   = IO\_DRW | IO\_EOT;
  ```
dx_reciottdata( ) — record voice data to multiple destinations

/*
 * Specify VOX file format for PCM at 8KHz.
 */
xb.wFileFormat = FILE_FORMAT_VOX;
xb.wDataFormat = DATA_FORMAT_PCM;
xb.nSamplesPerSec = DRT_8KHZ;
xb.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_playwav(chdev,"HELLO.WAV",&tpt,EV_SYNC) == -1) {
    printf("Error playing file - %s\n", ATDV_ERRMSGP(chdev));
    exit(4);
}

/* Start recording */
if (dx_reciottdata(chdev,&iott,&tpt,&xbp,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( )**

**Name:** SHORT 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

**Platform:** DM3, Springware

---

### 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 <strong>dx_open( )</strong></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 485.</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 520.</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

- On DM3 boards using a flexible routing configuration, 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_listenn()` function call on the device handle before calling any voice streaming function for that device handle. Furthermore, the `dx_listenn()` 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 this function returns -1 to indicate failure, one of the following reasons will be contained by `ATDV_LASTERR()`:

- **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; use `dx_fileerrno()` to obtain error value.

- **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
");
        /* 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);
}

/* clear digit buffer */
dx_clrdigbuf(chdev);

/* Start 6KHz ADPCM recording */
if (dx_recvox(chdev, "MESSAGE.VOX", &tpt, NULL, EM_TONE | EV_SYNC) == -1) {
    printf("Error recording file - %s\n", ATDV_ERRMSGP(chdev));
    exit(4);
}


- See Also
  - dx_rec()
  - dx_recf()
  - dx_recio_data()
  - dx_recwav()
**dx_recwav( ) — record voice data to a single WAVE file**

### dx_recwav( )

**Name:** \( \text{SHORT dx_recwav(chdev, filenamep, tptp, xpbp, mode)} \)

**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

**Platform:** DM3, Springware

---

### Description

The **dx_recwav()** convenience function records voice data to a single WAVE file. This function in turn calls **dx_reciothdata()**.

<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>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 <strong>DV_TPT</strong>, on page 485.</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 <strong>DX_XPB</strong>, on page 520.</td>
</tr>
</tbody>
</table>

**Note:** If **xpbp** is set to NULL, the function will record in 11 kHz linear 8-bit PCM.

**mode** specifies the play mode. The following two symbolic values may be used individually or ORed together:
- EV_SYNC – synchronous operation (must be specified)
- PM_TONE – play 200 msec audible tone
record voice data to a single WAVE file — dx_recwav()

## Cautions

On DM3 boards using a flexible routing configuration, 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.

## Errors

If this function returns -1 to indicate failure, one of the following reasons will be contained by ATDV_LASTERR():

- **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; use dx_fileerrno() to obtain error value
- **EDX_XPBPARM**
  - Invalid DX_XPB setting

## Example

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

main()
{
    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("dxxxxB1C1",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;

    ... 

    /* Read voice data */
    if ((fd = dx_rec(chdev,0)) == -1) {
        printf("Error reading voice data\n");
        dx_fileerrno();
        exit(1);
    }
}
```
dx_recwav() — record voice data to a single WAVE file

/* 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);
}

/* 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_reciottdata()
- 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

**Platform:** DM3

---

### 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");
    }
```
dx_ResetStreamBuffer( ) — reset internal data for a circular stream buffer

```c
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_resume()**

**Name:** int dx_resume(chdev)  
**Inputs:** int chdev  
**Returns:** 0 if success  
-1 if failure  
**Includes:** srllib.h  
dxxxmlib.h  
**Category:** I/O  
**Mode:** synchronous  
**Platform:** DM3

### Description

The **dx_resume()** function resumes the play that was paused using **dx_pause()**. The play is resumed exactly where the play was paused (that is, no data is lost). The application will not get an event when **dx_resume()** is issued. This function does not return an error if the channel is already in the requested state.

You can also pause and resume play using a DTMF digit. For more information, see SV_PAUSE and SV_RESUME in the DX_SVCB data structure and **dx_setsvcond()**.

For more information on the pause and resume play feature, 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 <strong>dx_open()</strong></td>
</tr>
</tbody>
</table>

### Cautions

None.

### Errors

If the function returns -1, use the Standard Runtime Library **ATDV_LASTERR()** standard attribute function to return the error code or **ATDV_ERRMSGP()** to return the descriptive error message. Possible errors for this function include:

**EDX_BUSY**  
Invalid state. Returned when the function is issued but play has not been paused on the channel.
dx_resume( ) — resume paused play

■ Example

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

main()
{
    int lDevHdl;
    DX_IOTT iott;

    /* Open a voice channel */
    int lDevHdl = dx_open("dxxxB1C1", 0);

    /* Start playing a prompt */
    DX_IOTT iott;
    /* Fill in the iott structure for the play */
    .
    .
    .

    /* Start playing */
    if( dx_playiottdata(lDevHdl, &iott, NULL, NULL, EV_ASYNC) < 0)
    {
        /* process error */
    }

    /* Pause the play */
    if( dx_pause(lDevHdl) < 0 )
    {
        /* process error */
    }

    /* Start the paused play again */
    if_dx_resume(lDevHdl) < 0)
    {
        /* process error */
    }
}
```

■ See Also

- dx_pause( )
**dx_RxIottData( )**

**Name:** int dx_RxIottData(chdev, iottp, lpTerminations, wType, lpParams, mode)

**Inputs:**
- int chdev • valid channel device handle
- DX_IOTT *iottp • pointer to I/O Transfer Table
- DV_TPT *lpTerminations • pointer to Termination Parameter Table
- int wType • data type
- LPVOID lpParams • pointer to data type-specific information
- int mode • function mode

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

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

**Category:** Analog Display Services Interface (ADSI)

**Mode:** asynchronous or synchronous

**Platform:** DM3, Springware

---

**Description**

The `dx_RxIottData( )` function is used to receive data on a specified channel. The data may come from any combination of data files, memory, or custom devices. The `wType` parameter specifies the type of data to be received, for example ADSI data.

After `dx_RxIottData( )` is called, data reception continues until one of the following occurs:

- `dx_stopch( )` is called
- the data requirements specified in the DX_IOTT are fulfilled
- the channel detects end of FSK data
- one of the conditions in the DV_TPT is satisfied

If the channel detects end of FSK data, the function is terminated. Use `ATDX_TERMMSK( )` to return the reason for the last I/O function termination on the channel. Possible return values are:

- TM_EOD
  - End of FSK data detected on receive
- TM_ERROR
  - I/O device error
- TM_MAXDATA
  - Maximum data reached; returned when the last I/O function terminates on DX_MAXDATA
- TM_MAXTIME
  - Maximum function time exceeded
**dx_RxIottData() — receive data on a specified channel**

**TM_USRSTOP**

Function stopped by user

Upon asynchronous completion of **dx_RxIottData()**, the TDX_RXDATA event is posted.

<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>iottp</td>
<td>points to the I/O Transfer Table. The <strong>iottp</strong> parameter specifies the destination for the received data. This is the same DX_IOTT structure used in <strong>dx_playiottdata()</strong> and <strong>dx_recioottdata()</strong>. See <strong>DX_IOTT</strong>, on page 509, for more information on this data structure.</td>
</tr>
<tr>
<td>lpTerminations</td>
<td>points to the Termination Parameter Table Structure, <strong>DV_TPT</strong>, which specifies termination conditions for the device handle. Supported values are:</td>
</tr>
<tr>
<td></td>
<td>• <strong>DX_MAXTIME</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>DX_MAXDATA</strong> (valid values are 1 - 65535 for <strong>tp_length</strong> field)</td>
</tr>
<tr>
<td></td>
<td>For more information on this structure, see <strong>DV_TPT</strong>, on page 485.</td>
</tr>
<tr>
<td>wType</td>
<td>specifies the type of data to be received. To receive ADSI data, set <strong>wType</strong> to <strong>DT_ADSI</strong>.</td>
</tr>
<tr>
<td>lpParams</td>
<td>points to information specific to the data type specified in <strong>wType</strong>. The format of the parameter block depends on <strong>wType</strong>. For ADSI data, set <strong>lpParams</strong> to point to an <strong>ADSI_XFERSTRUC</strong> structure. For more information on this structure, see <strong>ADSI_XFERSTRUC</strong>, on page 478.</td>
</tr>
<tr>
<td>mode</td>
<td>specifies how the function should execute, either EV_ASYNC (asynchronous) or EV_SYNC (synchronous)</td>
</tr>
</tbody>
</table>

**Cautions**

- Library level data is buffered when it is received. Applications can adjust the size of the buffers to address buffering delay. The DXCH_RXDATABUFSIZE channel parameter can be used with the **dx_setparm()** and **dx_getparm()** functions to adjust the buffer size.
- On Springware boards, **dx_RxIottdata()** will sometimes show an extra byte when receiving data. At the application level, this extra byte can be discarded by looking at the total number of bytes of data.

**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** (pointer to I/O transfer table)

**EDX_BADPARM**

Invalid data mode
EDX_BUSY
Channel already executing I/O function

EDX_SYSTEM
Error from operating system; use dx_fileerrno() to obtain error value

Example

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

```
// Synchronous receive ADSI data

#include "srllib.h"
#include "dxxxlib.h"

main()
{
    DX_IOTT iott = {0};
    char *devnamep = "dxxxX1C1";
    char buffer[16];
    ADSI_XFERSTRUC adsimode;
    DV_TPT tpt;
    int chdev;
    .
    .
    .

    sprintf(buffer, "RECEIVE.ADSI");
    if ((iott.io_fhandle = dx_fileopen(buffer, O_BINARY)) == -1) {
        /* Perform system error processing */
        exit(2);
    }
    if ((chdev = dx_open(devnamep, 0)) == -1) {
        fprintf(stderr, "Error opening channel %s\n", devnamep);
        dx_fileclose(iott.io_fhandle);
        exit(1);
    }
    .
    .
    .

    // destination is a file
    iott.io_type = IO_DEV|IO_EOT;
    iott.io_bufp = 0;
    iott.io_offset = 0;
    iott.io_length = -1;
    adsimode.cbSize = sizeof(adsimode);
    adsimode.dwRxDataMode = ADSI_NOALERT;
    printf("Waiting for incoming ring\n");
    dx_wtring(chdev, 2, DX_OFFHOOK, -1);
    // Specify maximum time termination condition in the TPT.
    // Application specific value is used to terminate dx_RxIottData( )
    // if end of data is not detected over a specified duration.
    tpt.tp_type = IO_EOT;
    if (dx_clrtpt(\tpt, 1) == -1) {
        // Process error
    }
```
dx_RxIottData( ) — receive data on a specified channel

```c

tpt.tp_termno = DX_MAXTIME;
tpt.tp_length = 1000;
tpt.tp_flags = TF_MAXTIME;

if (dx_RxIottData(chdev, &iott, NULL, DT_ADSI, &adsimode, EV_SYNC) < 0) {
    fprintf(stderr, "ERROR: dx_TxIottData failed on Channel %s; error: %s\n", ATDV_NAMEP(chdev), ATDV_ERRMSGP(chdev));
}
```

- dx_TxIottData( )
- dx_TxRxIottData( )

**See Also**
**dx_sendevt( )**

**Name:** int dx_sendevt(dev, evttype, evtdatap, evtlen, flags)

**Inputs:**
- int dev: valid channel device handle
- long evttype: type of event to be sent
- void *evtdatap: pointer to data block associated with evttype
- short evtlen: length of the data block in bytes
- unsigned short flags: which processes will receive this event

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

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

**Category:** Call Status Transition Event

**Mode:** synchronous

**Platform:** Springware

---

**Description**

The `dx_sendevt( )` function allows inter-process event communication. The event type parameter, `evttype`, and its associated data are sent to one or all processes that have the `dev` device opened.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dev</td>
<td>specifies the valid channel device handle obtained when the channel was opened using <code>dx_open( )</code></td>
</tr>
<tr>
<td>evttype</td>
<td>specifies the type of event to be sent. See the following page for more information on defining the type of event.</td>
</tr>
<tr>
<td>evtdatap</td>
<td>points to a data block associated with <code>evttype</code>. <strong>Note:</strong> The <code>evtdatap</code> parameter can be NULL and the <code>evtlen</code> parameter 0 if there is no data associated with an event type.</td>
</tr>
<tr>
<td>evtlen</td>
<td>specifies the length of the data block in bytes (between 0 and 256)</td>
</tr>
</tbody>
</table>
| flags     | determines which processes are going to receive this event. Valid values are:  
  - EVFL_SENDSELF – Only the process calling `dx_sendevt( )` will receive the event.  
  - EVFL_SENDOTHERS – All processes that have the device opened except the process calling `dx_sendevt( )` will receive the event.  
  - EVFL_SENDALL – All processes that have the device opened will receive the event. |

The events generated by this function can be retrieved using `sr_waitevt( )`, by registering an event handler via `sr_enbhdlr( )`, or by calling `dx_getevt( )` to catch the event if the `evttype` is set to TDX_CST.
dx_sendevt( ) — allow inter-process event communication

The application can define the **evttype** and **evtdata** to be any values as long as **evttype** is greater than 0x1FFFFFF and less than 0x7FFFFFF0. The only exception to this rule is the use of this function to stop **dx_wtring( )** and **dx_getevt( )** by sending TDX_CST events. To unblock a process waiting in **dx_wtring( )** or **dx_getevt( )**, send an event of type TDX_CST to that process. The **evtlen** will be the size of the DX_CST structure and **evtdatap** will point to a DX_CST structure with cst.cst_event set to DE_STOPRINGS or DE_STOPGETEVT as the case may be.

### Cautions
- This function will fail if an invalid device handle is specified.
- No event will be generated if event type value is greater than 0x7FFFFFF0.

### 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; use **dx_fileerrno( )** to obtain error value

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

main()
{
    int    dev;    /* device handle */
    DX_CST cst;    /* TDX_CST event data block */

    /* Open board 1 channel 1 device */
    if ((dev = dx_open("dxxxB1C1", 0)) == -1) {
        /* Perform system error processing */
        exit(1);
    }

    /* Set up DX_CST structure */
    cst.cst_event = DE_STOPGETEVT;
    cst.cst_data  = 0;

    /* Send the event to all other processes that have dxxxB1Cl open */
    if (dx_sendevt(dev, TDX_CST, &cst, sizeof(DX_CST), EVFL_SENDOTHERS) == -1) {
        printf("Error message = %s", ATDV_ERRMSGP(dev));
        exit(1);
    }
}
```

### See Also
- **dx_getevt( )**
- **sr_enbhdlr( )**
- **sr_waitevt( )**
dx_setchxfercnt( )

**Name:** int dx_setchxfercnt(chdev, bufnum, bufsize_identifier)

**Inputs:**
- int chdev  • valid channel device handle
- int bufnum  • allows for smaller driver data transfer buffer size
- int bufsize_identifier  • equate for a buffer size

**Returns:**
- 0 to indicate successful completion
- -1 if failure

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

**Category:** Configuration

**Mode:** synchronous

**Platform:** DM3, Springware

---

**Description**

The `dx_setchxfercnt( )` function sets the bulk queue buffer size for the channel. This function can change the size of the buffer used to transfer voice data between a user application and the Intel® telecom hardware.

The `dx_setchxfercnt( )` allows a smaller data transfer buffer size. The minimum buffer size is 1 kbytes, and the largest is 32 kbytes. In general, this function is used in conjunction with the User I/O feature; for more information, see the `dx_setuio( )` function. This function sets up the frequency with which the application-registered read or write functions are called by the voice DLL. For applications requiring more frequent access to voice data in smaller chunks, you can use `dx_setchxfercnt( )` on a per channel basis to lower the buffer size.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdev</td>
<td>specifies the valid device handle obtained when the device was opened using <code>xx_open( )</code>, where “xx” is the prefix identifying the device to be opened</td>
</tr>
<tr>
<td>bufsize_identifier</td>
<td>specifies the bulk queue buffer size for the channel. Use one of the following values:  • 0 – sets the buffer size to 4 kbytes  • 1 – sets the buffer size to 8 kbytes  • 2 – sets the buffer size to 16 kbytes  • 3 – sets the buffer size to 32 kbytes (default)  • 4 – sets the buffer size to 2 kbytes  • 5 – sets the buffer size to 1 kbytes  • 6 – sets the buffer size to 1.5 kbytes  Equates for these values are not available as #define in any header file.</td>
</tr>
</tbody>
</table>
dx_setchxfercnt( ) — set the bulk queue buffer size

- **Cautions**
  - This function fails if an invalid device handle is specified.
  - Do not use this function unless it is absolutely necessary to change the bulk queue buffer size between a user application and Intel® telecom hardware. Setting the buffer size to a smaller value can degrade system performance because data is transferred in smaller chunks.
  - A wrong buffer size can result in loss of data.
  - On DM3 boards, it is not recommended to set the bulk queue buffer size to less than 2 kbytes as this can result in loss of data (underrun condition) under high density load. To monitor underrun conditions, set DM_UNDERRUN in dx_setevtmsk( ).

- **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; use dx_fileerrno( ) to obtain error value

- **Example**
  ```c
  #include <windows.h>
  #include "srllib.h"
  #include "dxxxlib.h"

  main()
  {
    int dev;   /* device handle */

    /* Open board 1 channel 1 device */
    if ((dev = dx_open("dxxxB1C1", 0)) == -1) {
      /* Perform system error processing */
      exit(1);
    }

    /* Set the bulk data transfer buffer size to 1.5 kilobytes */
    if (dx_setchxfercnt(dev, 6) == -1) {
      printf("Error message = %s", ATDV_ERRMSGP(dev));
      exit(1);
    }
  }
  ```

- **See Also**
  - dx_setuio( )
  - dx_playiottdata( )
  - dx_reciottdata( )
  - DXCH_XFERBUFSIZE in dx_setparm( )
install and retrieve user-defined I/O functions — dx_setdevuio()

dx_setdevuio()

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

Platform: DM3, Springware

Description

The dx_setdevuio( ) function installs and retrieves user-defined I/O functions on a per channel device basis. The user I/O functions installed using this function 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 dx_setdevuio( ) 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 retuiop. If this is the first time dx_setdevuio( ) is called for a device, then retuiop 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 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.</td>
</tr>
</tbody>
</table>
dx_setdevuio() — install and retrieve user-defined I/O functions

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 519 for more information.
- On DM3 boards, 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_BADPARM
  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\n");
    return(read(fd, ptr, cnt));
  }

int appwrite(fd, ptr, cnt)
  int    fd;
  char   *ptr;
  unsigned cnt;
  {
    printf("appwrite: Write request\n");
    return(write(fd, ptr, cnt));
  }

int appseek(fd, offset, whence)
  int    fd;
  long   offset;
  int    whence;
  {
    printf("appseek: Seek request\n");
    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", ATDV_LASTERR(chdev) );
    }
}

See Also

- dx_setuio( )
dx_setdigbuf( ) — set the digit buffering mode

**dx_setdigbuf( )**

**Name:** int dx_setdigbuf(chdev, mode)

**Inputs:**
- int chdev • valid channel device handle
- int mode • digit buffering mode

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

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

**Category:** Configuration

**Mode:** synchronous

**Platform:** Springware

---

### Description

The **dx_setdigbuf( )** function sets the digit buffering mode that will be used by the voice driver. Once the digit buffer is full (31 digits), the application may select whether subsequent digits will be ignored or will overwrite the oldest digits in the queue.

<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>mode</td>
<td>specifies the type of digit buffering that will be used. Mode can be:</td>
</tr>
<tr>
<td></td>
<td>• <strong>DX_DIGCYCLIC</strong> – Incoming digits will overwrite the oldest digits in the buffer if the buffer is full.</td>
</tr>
<tr>
<td></td>
<td>• <strong>DX_DIGTRUNC</strong> – Incoming digits will be ignored if the digit buffer is full (default).</td>
</tr>
</tbody>
</table>

### Cautions

When you call **dx_setdigbuf( )**, the function clears the previously detected digits in the digit buffer.

### 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; use **dx_fileerrno( )** to obtain error value
EDX_TIMEOUT
  Timeout limit is reached

■ Example

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

main()
{
    int chfd;

    int init_digbuf()
    {
        /* open the device using dx_open, chfd has the device handle */
        /*
         * Set up digit buffering to be Cyclic. When digit
         * queue overflows oldest digit will be overwritten
         */
        if (dx_setdigbuf(chfd, DX_DIGCYCLIC) == -1) {
            printf("Error during setdigbuf %s\n", ATDV_ERRMSG[chfd]);
            return(1);
        }
        return(0);
    }
}
```

■ See Also

None.
dx_setdigtyp( )

**Name:** int dx_setdigtyp(chdev, dmask)

**Inputs:**
- int chdev  
  • valid channel device handle
- unsigned short dmask  
  • type of digit the channel will detect

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

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

**Category:** Configuration

**Mode:** synchronous

**Platform:** DM3, Springware

---

**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, DPD. To set user-defined digits, use the **dx_addtone( )** function.

2. **dx_setdigtyp( )** does not clear the previously detected digits in the digit buffer.

**Parameter** | **Description**
--- | ---
chdev | specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**
dmask | sets the type of digits the channel will detect. More than one type of digit detection can be enabled in a single function call, as shown in the function example.

On DM3 boards, the following are valid values:
- DM_DTMF – enable DTMF digit detection
- DM_MF – enable MF digit detection
- NULL – disable digit detection

On Springware boards, the following are valid values:
- DM_DTMF – enable DTMF digit detection (default setting)
- DM_APD – enable audio pulse digits detection
- DM_MF – enable MF digit detection
- DM_DPD – enable dial pulse digit (DPD) detection
- DM_DPDZ – enable zero train DPD detection

To disable digit detection, set **dmask** to NULL.
Notes: 1. MF detection can only be enabled on systems with MF capability.

2. The digit detection type specified in dmask will remain valid after the channel has been closed and reopened.

3. Global DPD can only be enabled on systems with this capability.

4. The Global DPD feature must be implemented on a call-by-call basis to work correctly. Global DPD must be enabled for each call by calling dx_setdigtyp( ).

5. dx_setdigtyp( ) overrides digit detection enabled in any previous use of dx_setdigtyp( ).

For any digit detected, you can determine the digit type, DTMF, MF, GTD (user-defined) or DPD, 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. This method allows you to determine very quickly whether a pulse or DTMF telephone is being used. For more information on this structure, see DV_DIGIT, on page 483.

■ 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; use dx_fileerrno( ) to obtain error value

■ Example

/*$ 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
    */
dx_setdigtyp( ) — control the types of digits detected by the voice channel

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

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

/* continue processing call */

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

main()
{
    int chdev;

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

/* Set channel to detect DTMF */
if (dx_setdigtyp(chdev, DM_DTMF) == -1) {
    /* error routine */
}

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

**Platform:** DM3, Springware

---

**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 473.

**Notes:**
1. This function can enable detection for all CST events except user-defined tone detection. See `dx_addtone( )` and `dx_enbtone( )` for information.

2. The `dx_wtring( )` function affects CST events that are enabled. It enables detection of the DM_RINGS event and disables detection of other events.
dx_setevtmsk( ) — enable detection of call status transition (CST) events

<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. On DM3 boards, 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. On Springware boards, one or more of the following bits can be set:  
• DM_LCOFF – wait for loop current to be off  
• DM_LCON – wait for loop current to be on  
• DM_RINGS – wait for rings; see also dx_wtring( )  
• DM_RNGOFF – wait for ring to drop (hang-up)  
• DM_SILOF – wait for non-silence  
• DM_SILON – wait for silence  
• DM_WINK – wait for wink to occur on an E&M line. If DM_WINK is not enabled and DM_RINGS is enabled, a wink may be interpreted as an incoming call, depending upon the setting of the DXBD_R_ON parameter.  
• 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_LCREV – wait for flow of current to reverse. When the DM_LCREV bit is enabled, a DE_LCREV event message is queued when the flow of current over the line is reversed. |

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:
enable detection of call status transition (CST) events — dx_setevtmsk( )

/* Set event mask to collect digits */
if (dx_setevtmsk(chdev, DM_DIGITS) == -1) {

To disable DM_DIGITS (turn off the digits flag and stop queuing digits):

dx_setevtmsk(DM_DIGOFF);
dx_clrdigbuf(chdev); /*Clear out queue*/

The following outlines the synchronous or asynchronous handling of CST events:

Synchronous Application
Call dx_setevtmsk( ) to enable CST events.
Call dx_getevt( ) to wait for CST events.
Events are returned to the DX_EBLK structure.

Asynchronous Application
Call dx_setevtmsk( ) to enable CST events.
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.

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; use dx_fileerrno( ) to obtain error value

Example 1

This example illustrates how to use dx_setevtmsk( ) to wait for ring events in a synchronous application.

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

main()
{
    int chdev;
    DX_EBLK eblk;
    .
    .
}
/* open a channel with chdev as descriptor */
if ((chdev = dx_open("dxxxB1C1", NULL)) == -1) {
    /* process error */
}

/* Set event mask to receive ring events */
if (dx_setevtmsk(chdev, DM_RINGS) == -1) {
    /* error setting event */
}

/* check for ring event, timeout set to 20 seconds */
if (dx_getevt(chdev, &eblk, 20) == -1) {
    /* error timeout */
}
if (eblk.ev_event == DE_RINGS) {
    printf("Ring event occurred\n");
}

Example 2

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>
#include <windows.h>
#define MAXCHAN 24

int cst_handler();

main()
{
    int chdev[MAXCHAN];
    char *chname;
    int i, srlmode;
    /* Set SRL to run in polled mode. */
    srlmode = SR_POLLMODE;
    if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1) {
        /* process error */
    }
    for (i = 0; i < MAXCHAN; i++) {
        /* Set chname to the channel name, e.g., dxxxB1C1, dxxxB1C2,... */
        /* Open the device using dx_open(). chdev[i] has channel device
         * descriptor.
         */
        if ((chdev[i] = dx_open(chname, NULL)) == -1) {
            /* process error */
        }
    }
```
/* Use dx_setevtmse() to enable call status transition events on this channel. */
if (dx_setevtmse(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_waitvt to wait for call status transition event. On receiving the transition event, TDX_CST, control is transferred to the handler function previously established using sr_enbhdlr(). */
.
.
int cst_handler()
{
    DX_CST *cstp;
    /* sr_getevtdatap() points to the event that caused the call status transition. */
cstp = (DX_CST *)sr_getevtdatap();
    switch (cstp->cst_event) {
    case DE_RINGS:
        printf("Ring event occurred on channel %s\n", ATDX_NAMEP(sr_getevtdev()));
        break;
    case DE_WINK:
        printf("Wink event occurred on channel %s\n", ATDX_NAMEP(sr_getevtdev()));
        break;
    case DE_LCON:
        printf("Loop current ON event occurred on channel %s\n", ATDX_NAMEP(sr_getevtdev()));
        break;
    case DE_LCOFF:
        .
    }
    /* Kick off next function in the state machine model. */
    return 0;
}

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
dxxlib.h

**Category:** Global Tone Detection

**Mode:** synchronous

**Platform:** DM3, Springware

---

**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_minamp* parameters.</td>
</tr>
<tr>
<td>GT_MAX_DEF</td>
<td>Default value in dB for maximum GTD amplitude that can be entered for gtd_maxamp* 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.
set up the tone detection amplitudes — dx_setgtdamp()

- 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>
#include <windows.h>

#define TID 1;          /* Tone ID */

*dx_setgtdamp(-30, 0, -30, 0);

/*
 * Set amplitude for GTD;
 *   freq1 -30dBm to 0 dBm
 *   freq2 -30dBm to 0 dBm
 */

if (dx_blddt(TID, 1000, 50, 500, 25, TN LEADING) == -1) {
    /* Perform system error processing */
    exit(3);
}
```

**See Also**

None.
**dx_sethook( ) — provide control of the hook switch status**

### dx_sethook( )

**Name:** int dx_sethook(chdev, hookstate, mode)

**Inputs:**
- int chdev • valid channel device handle
- int hookstate • hook state (on-hook or off-hook)
- unsigned short mode • asynchronous/synchronous

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

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

**Category:** Configuration

**Mode:** asynchronous or synchronous

**Platform:** Springware

---

### Description

The **dx_sethook( )** function provides control of the hook switch status of the specified channel. A hook switch state may be either on-hook or off-hook.

<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>
| hookstate | forces the hookstate of the specified channel to on-hook or off-hook. The following values can be specified:  
  - DX_OFFHOOK – set to off-hook state  
  - DX_ONHOOK – set to on-hook state |
| mode      | specifies whether to run **dx_sethook**( ) asynchronously or synchronously. Specify one of the following:  
  - EV_ASYNC – run **dx_sethook**( ) asynchronously  
  - EV_SYNC – run **dx_sethook**( ) synchronously (default) |

**Notes:**
1. Do not call this function for a digital T-1 TDM bus configuration. Transparent signaling for TDM bus digital interface devices is not supported.
2. Calling **dx_sethook**( ) with no parameters clears the loop current and silence history from the channel’s buffers.

### Asynchronous Operation

To run **dx_sethook**( ) asynchronously, set the mode field to EV_ASYNC. The function will return 0 to indicate it has initiated successfully, and will generate a termination event to indicate completion. Use the Standard Runtime Library (SRL) Event Management functions to handle the termination event.
If running asynchronously, termination is indicated by a TDX_SETHOOK event. The cst_event field in the DX_CST data structure will specify one of the following:

- DX_ONHOOK if the hookstate has been set to on-hook
- DX_OFFHOOK if the hookstate has been set to off-hook

Use the Event Management function sr_getevtdatap() to return a pointer to the DX_CST structure.

ATDX_HOOKST() will also return the type of hookstate event.

### Synchronous Operation

By default, this function runs synchronously.

If running synchronously, dx_sethook() will return 0 when complete.

### 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_ERRMSPG( ) 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; use dx_fileerrno( ) to obtain error value

### Example 1

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

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

main()
{
  int chdev,
  /* open a channel with chdev as descriptor */
  if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
    /* process error */
  }
  /* put the channel on-hook */
  if (dx_sethook(chdev,DX_OFFHOOK,EV_SYNC) == -1) {
    /* error setting hook state */
  }
  ...
```
dx_sethook() — provide control of the hook switch status

/* take the channel off-hook */
if (dx_sethook(chdev, DX_OFFHOOK, EV_SYNC) == -1) {
    /* error setting hook state */
    return;
}

Example 2

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

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

#define MAXCHAN 24

int sethook_hdlr();

main()
{
    int i, chdev[MAXCHAN];
    char *chnamep;
    int srilmode;

    /* Set SRL to run in polled mode. */
    srilmode = SR_POLLMODE;
    if (sr_setparm(SRL_SERVICE, SR_MODEID, (void *)&srilmode) == -1) {
        /* process error */
    }
    for (i=0; i<MAXCHAN; i++) {
        /* Set chnamep to the channel name - e.g. dxxxB1C1, dxxxB1C2, ... */
        /* open a channel with chdev[i] as descriptor */
        if ((chdev[i] = dx_open(chnamep, NULL)) == -1) {
            /* process error */
        }
        /* Using sr_enbhdlr(), set up handler function to handle sethook
         * events on this channel. */
        if (sr_enbhdlr(chdev[i], TDX_SETHOOK, sethook_hdlr) == -1) {
            /* process error */
        }
        /* put the channel on-hook */
        if (dx_sethook(chdev[i], DX_ONHOOK, EV_ASYNC) == -1) {
            /* error setting hook state */
        }
    }
    /* Use sr_waitevt() to wait for the completion of dx_sethook().
     * On receiving the completion event, TDX_SETHOOK, control is transferred
     * to the handler function previously established using sr_enbhdlr(). */
}

int sethook_hdlr()
{
    DX_CST *cstp;
```
provide control of the hook switch status — dx_sethook()

/* sr_getevtdatap() points to the call status transition event structure, which contains the hook state of the device. */
cstp = (DX_CST *)sr_getevtdatap();
switch (cstp->cst_event) {
  case DX_ONHOOK:
    printf("Channel %s is ON hook\n", ATDX_NAMEP(sr_getevtdev()));
    break;
  case DX_OFFHOOK:
    printf("Channel %s is OFF hook\n", ATDX_NAMEP(sr_getevtdev()));
    break;
  default:
    /* process error */
    break;
}

/* Kick off next function in the state machine model. */
.
.
return 0;

See Also

- sr_getevtdatap()
- ATDX_HOOKST()}
dx_setparm( ) — set physical parameters of a channel or board device

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

Platform: DM3, Springware

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 dx_open( ).</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.</td>
</tr>
<tr>
<td></td>
<td>For DM3 boards, board parameter defines are described in Table 12. For Springware boards, board parameter defines are described in Table 13. For DM3 boards, channel parameter defines are described in Table 14. For Springware boards, channel parameter defines are described in Table 15.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>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 *dxxlib.h* file contains defined masks for parameters that can be examined and set using `dx_getparm()` and `dx_setparm()`.

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

For DM3 boards, the supported board parameter defines are shown in Table 12.
dx_setparm( ) — set physical parameters of a channel or board device

Table 12. Voice Board Parameters (DM3)

<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_HWTYPE</td>
<td>1</td>
<td>R</td>
<td>-</td>
<td>Hardware Type. On DM3 boards, TYP_D41</td>
</tr>
<tr>
<td>DXBD_SYSCFG</td>
<td>1</td>
<td>R</td>
<td>-</td>
<td>System Configuration. On DM3 boards, 1 is always returned</td>
</tr>
</tbody>
</table>

For Springware boards, the supported board parameter defines are shown in Table 13.

Table 13. Voice Board Parameters (Springware)

<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_FLASHCHR</td>
<td>1</td>
<td>R/W</td>
<td>&amp;</td>
<td>Flash character. Character that causes a hook flash when detected</td>
</tr>
<tr>
<td>DXBD_FLASHTM</td>
<td>2</td>
<td>R/W</td>
<td>50</td>
<td>Flash Time. Length of time onhook during flash (10 msec units)</td>
</tr>
<tr>
<td>DXBD_HWTYPE</td>
<td>1</td>
<td>R</td>
<td>-</td>
<td>Hardware Type. Value can be:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- TYP_D40 – D/40 board</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- TYP_D41 – D/21, D/41, D/xxxSC board</td>
</tr>
<tr>
<td>DXBD_MAXPDOff</td>
<td>2</td>
<td>R/W</td>
<td>50</td>
<td>Maximum Pulse Digit Off. Maximum time loop current may be off before the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>existing loop pulse digit is considered invalid and reception is</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>reinitialized (10 msec units)</td>
</tr>
<tr>
<td>DXBD_MAXSLOFF</td>
<td>2</td>
<td>R/W</td>
<td>25</td>
<td>Maximum Silence Off. Maximum time for silence being off, during</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>audio pulse detection (10 msec units)</td>
</tr>
<tr>
<td>DXBD_MINIPD</td>
<td>2</td>
<td>R/W</td>
<td>25</td>
<td>Minimum Loop Interpulse Detection. Minimum time between loop pulse digits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>during loop pulse detection (10 msec units)</td>
</tr>
<tr>
<td>DXBD_MINISL</td>
<td>2</td>
<td>R/W</td>
<td>25</td>
<td>Minimum Interdigit Silence. Minimum time for silence on between pulse</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>digits for audio pulse detection (10 msec units)</td>
</tr>
<tr>
<td>DXBD_MINLCOFF</td>
<td>2</td>
<td>R/W</td>
<td>0</td>
<td>Minimum Loop Current Off. Minimum time before loop current drop message is</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sent (10 msec units)</td>
</tr>
<tr>
<td>DXBD_MINOFFHKTM</td>
<td>2</td>
<td>R/W</td>
<td>250</td>
<td>Minimum offhook time (10 msec units)</td>
</tr>
<tr>
<td>DXBD_MINPDOff</td>
<td>1</td>
<td>R/W</td>
<td>2</td>
<td>Minimum Pulse Detection Off. Minimum break interval for valid loop pulse</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>detection (10 msec units)</td>
</tr>
<tr>
<td>DXBD_MINPDOn</td>
<td>1</td>
<td>R/W</td>
<td>2</td>
<td>Minimum Pulse Detection On. Minimum make interval for valid loop pulse</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>detection (10 msec units)</td>
</tr>
<tr>
<td>DXBD_MINSLOFF</td>
<td>1</td>
<td>R/W</td>
<td>2</td>
<td>Minimum Silence Off. Minimum time for silence to be off for valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>audio pulse detection (10 msec units)</td>
</tr>
<tr>
<td>DXBD_MINSLON</td>
<td>1</td>
<td>R/W</td>
<td>1</td>
<td>Minimum Silence On. Minimum time for silence to be on for valid audio</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pulse detection (10 msec units)</td>
</tr>
<tr>
<td>DXBD_MINTOFF</td>
<td>1</td>
<td>R/W</td>
<td>5</td>
<td>Minimum DTI Off. Minimum time required between rings-received events (10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>msec units)</td>
</tr>
<tr>
<td>DXBD_MINTION</td>
<td>1</td>
<td>R/W</td>
<td>5</td>
<td>Minimum DTI On. Minimum time required for rings received event (10 msec</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>units)</td>
</tr>
</tbody>
</table>

Table 13. Voice Board Parameters (Springware) ( Continued)

<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_OFFHDLY</td>
<td>2</td>
<td>R/W</td>
<td>50</td>
<td>Offhook Delay. Period after offhook, during which no events are generated; that is, no DTMF digits will be detected during this period (10 msec units).</td>
</tr>
<tr>
<td>DXBD_P_BK</td>
<td>2</td>
<td>R/W</td>
<td>6</td>
<td>Pulse Dial Break. Duration of pulse dial off-hook interval (10 msec units)</td>
</tr>
<tr>
<td>DXBD_P_IDD</td>
<td>2</td>
<td>R/W</td>
<td>100</td>
<td>Pulse Interdigit Delay. Time between digits in pulse dialing (10 msec units)</td>
</tr>
<tr>
<td>DXBD_P_MK</td>
<td>2</td>
<td>R/W</td>
<td>4</td>
<td>Pulse Dial Make. Duration of pulse dial offhook interval (10 msec units)</td>
</tr>
<tr>
<td>DXBD_PAUSETM</td>
<td>2</td>
<td>R/W</td>
<td>200</td>
<td>Pause Time. Delay caused by a comma in the dialing string (10 msec units)</td>
</tr>
<tr>
<td>DXBD_R_EDGE</td>
<td>1</td>
<td>R/W</td>
<td>ET_ROFF</td>
<td>Ring Edge. Detection of ring edge, values can be:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• ET_RON – beginning of ring</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• ET_ROFF – end of ring</td>
</tr>
<tr>
<td>DXBD_R_IRD</td>
<td>2</td>
<td>R/W</td>
<td>80</td>
<td>Inter-ring Delay. Maximum time to wait for the next ring (100 msec units).</td>
</tr>
<tr>
<td>DXBD_R_OFF</td>
<td>2</td>
<td>R/W</td>
<td>5</td>
<td>Ring-off Interval. Minimum time for ring not to be present before qualifying as “not ringing” (100 msec units)</td>
</tr>
<tr>
<td>DXBD_R_ON</td>
<td>2</td>
<td>R/W</td>
<td>3</td>
<td>Ring-on Interval. Minimum time ring must be present to qualify as a ring (100 msec units)</td>
</tr>
<tr>
<td>DXBD_S_BNC</td>
<td>2</td>
<td>R/W</td>
<td>4</td>
<td>Silence and Non-silence Debounce. Length of a changed state before call status transition message is generated (10 msec units)</td>
</tr>
<tr>
<td>DXBD_SYSCFG</td>
<td>1</td>
<td>R</td>
<td>-</td>
<td>System Configuration. JP8 status for D/4x boards.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• 0 – loop start interface (JP8 in)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• 1 – DTI/xxx interface (JP8 out)</td>
</tr>
<tr>
<td>DXBD_T_IDD</td>
<td>2</td>
<td>R/W</td>
<td>5</td>
<td>DTMF Interdigit delay. Time between digits in DTMF dialing (10 msec units)</td>
</tr>
<tr>
<td>DXBD_TTDATA</td>
<td>1</td>
<td>R/W</td>
<td>10</td>
<td>DTMF length (duration) for dialing (10 msec units)</td>
</tr>
</tbody>
</table>

Channel Parameter Defines

For DM3 boards, the supported channel parameter defines are shown in Table 14. All time units are in multiples of 10 msec unless otherwise noted.

Table 14. Voice Channel Parameters (DM3)

<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</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>
</tbody>
</table>
Table 14. Voice Channel Parameters (DM3) (Continued)

<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_AGCG_MEMORY SILENCERESET</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 m sec units.</td>
</tr>
<tr>
<td>DXCH_AGCG_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_AGCG_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_AGCG_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>
<tr>
<td>DXCH_FSKCHSEIZURE</td>
<td></td>
<td></td>
<td></td>
<td>ADSI two-way FSK. For a given FSK protocol standard specified in DXCH_FSKSTANDARD, this parameter allows the application to set the channel seizure. If the FSK protocol standard is set to Bellcore, the default value for the channel seizure when transmitting data is 300 bits. The default value for the channel seizure when receiving data is 60 bits. These values cannot be modified. If the FSK protocol standard is set to ETSI, when transmitting data, the range of possible values is 0 to 300 bits. If you specify a value outside of this range, the library uses 300 bits as the default when transmitting data. If you do not specify a value for channel seizure, the library uses 0 bits as the default. If the FSK protocol standard is set to ETSI, when receiving data, the library uses 0 bits as the default when receiving data. If you do not specify a value for channel seizure, the library uses 0 bits as the default.</td>
</tr>
<tr>
<td>DXCH_FSKINTERBLK TIMEOUT</td>
<td></td>
<td></td>
<td>120</td>
<td>ADSI two-way FSK. Measured in milliseconds. The firmware gets FSK data in bursts. This parameter specifies how long the firmware should wait for the next burst of FSK data before it can conclude that no more data will be coming and can terminate the receive session. In short, this parameter denotes the maximum time between any two FSK data bursts in one receive session. This property can only be supplied for reception of FSK data with dx_RxIottData( ).</td>
</tr>
</tbody>
</table>
Table 14. Voice Channel Parameters (DM3) (Continued)

<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_FSKMARKLENGTH</td>
<td></td>
<td></td>
<td></td>
<td>ADSI two-way FSK. For a given FSK protocol standard specified in DXCH_FSKSTANDARD, the DXCH_FSKMARKLENGTH parameter allows the application to set the mark length. If the FSK protocol standard is set to Bellcore, the default value for the mark length when transmitting data is 180 bits. The default value for the mark length when receiving data is 30 bits. These values cannot be modified. If the FSK protocol standard is set to ETSI, when transmitting data, the range of possible values is 80 to 180 bits. If you specify a value outside of this range, the library uses 180 bits as the default when transmitting data. If you do not specify a value for mark length, the library uses 80 bits as the default. If the FSK protocol standard is set to ETSI, when receiving data, the range of possible values is 0 to 60 bits. If you specify a value outside of this range, it uses 30 bits as the default when receiving data. If you do not specify a value for mark length, the library uses 0 bits as the default.</td>
</tr>
<tr>
<td>DXCH_FSKSTANDARD</td>
<td></td>
<td></td>
<td></td>
<td>ADSI two-way FSK. Specifies the FSK protocol standard, which is used for transmission and reception of FSK data. Using this channel parameter, the protocol standard can be set to either DX_FSKSTDBELLCORE (Bellcore standard) or DX_FSKSTDETSI (ETSI standard). The default value is DX_FSKSTDBELLCORE. If you set DXCH_FSKSTANDARD to DX_FSKSTDETSI, we recommend that you explicitly specify values for the DXCH_FSKCHSEIZURE and DXCH_FSKMARKLENGTH parameters.</td>
</tr>
<tr>
<td>DXCH_PLAYDRATE</td>
<td>2</td>
<td>R/W</td>
<td>6000</td>
<td>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</td>
</tr>
<tr>
<td>DXCH_RECRDRATE</td>
<td>2</td>
<td>R/W</td>
<td>6000</td>
<td>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</td>
</tr>
<tr>
<td>DXCH_SCRFEATURE</td>
<td>2</td>
<td>R/W</td>
<td>-</td>
<td>Silence Compressed Record (SCR). Valid values are: DXCH_SCRDISABLED – SCR feature disabled DXCH_SCRENABLED – SCR feature enabled</td>
</tr>
<tr>
<td>DXCH_XFERBUFSIZE</td>
<td>4</td>
<td>R</td>
<td>32 kbytes</td>
<td>Transfer buffer size. Returns the bulk queue buffer size as set by the dx_setchxfercnt( ) function.</td>
</tr>
</tbody>
</table>

For Springware boards, the supported channel parameter defines are shown in Table 15. All time units are in multiples of 10 msec unless otherwise noted.
### Table 15. Voice Channel Parameters (Springware)

<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_AUDIOLINEIN</td>
<td></td>
<td></td>
<td></td>
<td>Enables or disables the ProLine/2V audio jack line-in on voice channel 2</td>
</tr>
<tr>
<td>DXCH_CALLID</td>
<td></td>
<td></td>
<td>disabled</td>
<td>Enables or disables caller ID for the channel. Valid values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- DX_CALLIDENABLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- DX_CALLIDDISABLE (default)</td>
</tr>
<tr>
<td>DXCH_DFLAGS</td>
<td>2</td>
<td>R/W</td>
<td>0</td>
<td>DTMF detection edge select</td>
</tr>
<tr>
<td>DXCH_DTINITSET</td>
<td>2</td>
<td>R/W</td>
<td>0</td>
<td>Specifies which DTMF digits to initiate play on. Values of different DTMF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>digits may be ORed together to form the bit mask. Possible values and the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>corresponding digits are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- DM_1 – DTMF digit 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- DM_2 – DTMF digit 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- DM_3 – DTMF digit 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- DM_4 – DTMF digit 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- DM_5 – DTMF digit 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- DM_6 – DTMF digit 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- DM_7 – DTMF digit 7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- DM_8 – DTMF digit 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- DM_9 – DTMF digit 9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- DM_0 – DTMF digit 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- DM_S – ^</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- DM_P – #</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- DM_A – a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- DM_B – b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- DM_C – c</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- DM_D – d</td>
</tr>
<tr>
<td>DXCH_DTMFDEB</td>
<td>2</td>
<td>R/W</td>
<td>0</td>
<td>DTMF debounce time (record delay). Sets the minimum time for DTMF to be</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>present to be considered valid. Used to remove false DTMF signals during</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>recording. Increase the value for less sensitivity to DTMF.</td>
</tr>
<tr>
<td>DXCH_DTMFTLK</td>
<td>2</td>
<td>R/W</td>
<td>5</td>
<td>Sets the minimum time for DTMF to be present during playback to be</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>considered valid. Increasing the value provides more immunity to talk-off/</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>playoff. Set to -1 to disable.</td>
</tr>
<tr>
<td>DXCH_MAXRWINK</td>
<td>1</td>
<td>R/W</td>
<td>20</td>
<td>Maximum Loop Current for Wink. Sets the maximum time that loop current</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>needs to be on before recognizing a wink (10 msec units).</td>
</tr>
<tr>
<td>DXCH_MFDELAY</td>
<td>2</td>
<td>R/W</td>
<td>6</td>
<td>MF Interdigit Delay. Sets the length of the silence period between tones</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>during MF dialing. This parameter affects all the channels on the board. (10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>msec units).</td>
</tr>
<tr>
<td>DXCH_MFMINON</td>
<td>2</td>
<td>R/W</td>
<td>0</td>
<td>Minimum MF On. Sets the duration to be added to the standard MF tone</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>duration before the tone is detected. The minimum detection duration is</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>65 msec for KP tones and 40 msec for all other tones. This parameter affects</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>all the channels on the board. (10 msec units).</td>
</tr>
</tbody>
</table>
### Table 15. Voice Channel Parameters (Springware) (Continued)

<table>
<thead>
<tr>
<th>Define</th>
<th>Bytes</th>
<th>Read/ Write</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
</table>
| DXCH_MFMODE             | 2     | R/W         | 2       | Specifies a word-length bit mask that selects the minimum length of KP tones to be detected. The possible values of this field are:  
  - 0 – detect KP tone > 40 msec  
  - 2 – detect KP tone > 65 msec  
  If the value is set to 2, any KP tone greater than 65 msec will be returned to the application during MF detection. This ensures that only standard-length KP tones (100 msec) are detected. If set to 0 (zero), any KP tone longer than 40 msec will be detected. |
| DXCH_MFLKPTONE          | 2     | R/W         | 10      | MF Length of LKP Tone. Specifies the length of the LKP tone during MF dialing. This parameter affects all the channels on the specified board.  
  Maximum value: 15 (10 msec units) |
| DXCH_MFTONE             | 2     | R/W         | 6       | MF Minimum Tone Duration. Specifies the duration of a dialed MF tone. This parameter affects all the channels on the board.  
  Maximum value: 10 (10 msec units) |
| DXCH_MINRWINK           | 1     | R/W         | 10      | Minimum Loop Current for Wink. Specifies the minimum time that loop current needs to be on before recognizing a wink (10 msec units). |
| DXCH_NUMRXBUFFERS       | 4     | R/W         | 2       | Changes the number of record buffers used. Before you can use DXCH_NUMRXBUFFERS, you must set DXCH_VARNUMBUFFERS to 1 and specify the size of the record buffer in DXCH_RXDATABUFSIZE. This value can be 2 or greater. |
| DXCH_NUMTXBUFFERS       | 4     | R/W         | 2       | Sets the number of play buffers. Before you can use DXCH_NUMTXBUFFERS, you must set DXCH_VARNUMBUFFERS to 1 and specify the size of the play buffer in DXCH_TXDATABUFSIZE. This value can be 2 or greater. |
| 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 |
| DXCH_RINGCNT            | 2     | R/W         | 4       | Specifies number of rings to wait before returning a ring event. |
| DXCH_RXDATABUFSIZE      | 4     | R/W         | 32 kbytes | Sets the size of the record buffers only that are used to transfer data (e.g., ADSI data) between the application on the host and the driver to control buffering delay. The buffer is used by the dx_RxIottData( ) and dx.TxRxIottData( ) functions. The minimum buffer size is 128 bytes. The largest available buffer size is 32 kbytes (must be in multiples of 128). If play and record buffers are the same size, use DXCH_XFERBUFSIZE. |
### dx_setparm( ) — set physical parameters of a channel or board device

Table 15. Voice Channel Parameters (Springware)  (Continued)

<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_T_IDD</td>
<td>2</td>
<td>R/W</td>
<td>5</td>
<td>Specifies DTMF Interdigit delay (time between digits in DTMF dialing)</td>
</tr>
<tr>
<td>DXCH_TTDATA</td>
<td>1</td>
<td>R/W</td>
<td>10</td>
<td>Specifies DTMF length (duration) for dialing.</td>
</tr>
<tr>
<td>DXCH_TXDATABUFSIZE</td>
<td>4</td>
<td>R/W</td>
<td>32 kbytes</td>
<td>Sets the size of the play buffers only that are used to transfer data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>between the application on the host and the driver. The minimum buffer size is 128 bytes. The largest available buffer size is 32 kbytes (must be in multiples of 128). If play and record buffers are the same size, use DXCH_XFERBUFSIZE.</td>
</tr>
<tr>
<td>DXCH_VARNUMBUFFERS</td>
<td>4</td>
<td>R/W</td>
<td>0</td>
<td>Allows you to use more than two play or record buffers when set to 1. This parameter is used in conjunction with DXCH_XFERBUFSIZE, DXCH_RXDATABUFSIZE, DXCH_TXDATABUFSIZE, DXCH_NUMRXBUFFERS and DXCH_NUMTXBUFFERS. Valid parameter values are: 1 (True) – more than 2 buffers, 0 (False) – 2 buffers.</td>
</tr>
<tr>
<td>DXCH_WINKDLY</td>
<td>1</td>
<td>R/W</td>
<td>15</td>
<td>Wink Delay. Specifies the delay after a ring is received before issuing a wink (10 msec units)</td>
</tr>
<tr>
<td>DXCH_WINKLEN</td>
<td>1</td>
<td>R/W</td>
<td>15</td>
<td>Wink Length. Specifies the duration of a wink in the off-hook state (10 msec units)</td>
</tr>
<tr>
<td>DXCH_XFERBUFSIZE</td>
<td>4</td>
<td>R/W</td>
<td>32 kbytes</td>
<td>Sets the size of both the play and record buffers used to transfer data between the application on the host and the driver. These buffers are also called driver buffers. The minimum buffer size is 128 bytes. The largest available buffer size is 32 kbytes (must be in multiples of 128). This parm can be used with the dx_getparm( ) function. The dx_setchxfercnt( ) function sets the bulk queue buffer size for the channel. This function can change the size of the buffer used to transfer voice data between a user application and the hardware. The dx_setchxfercnt( ) function allows a smaller driver data transfer buffer size. The minimum buffer size is now 1KB. The largest available buffer size is 32 kbytes. In general, this function is used in conjunction with the User I/O feature; for more information, see the dx_setuio( ) function. This function sets up the frequency with which the application-registered read or write functions are called by the voice dll. For applications requiring more frequent access to voice data in smaller chunks, you can use this function on a per channel basis to lower the buffer size.</td>
</tr>
</tbody>
</table>

**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.
set physical parameters of a channel or board device — dx_setparm()  

- **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; use dx_fileerrno() to obtain error value

- **Example**

```c
#include <srllib.h>
#include <dx xxxlib.h>
#include <windows.h>

main()
{
    int bddev, parmval;
    /* Open the board using dx_open( ). Get board device descriptor in
     * bddev.
     */
    if ((bddev = dx_open(“dxxxB1”, NULL)) == -1) {
        /* process error */
    }

    /* Set the inter-ring delay to 6 seconds (default = 8) */
    parmval = 6;
    if (dx_setparm(bddev, DXBD_R_IRD, (void *)&parmval) == -1) {
        /* process error */
    }

    /* now wait for an incoming ring */
    ...
}
```

- **See Also**

- dx_getparm()
dx_setparm() — set physical parameters of a channel or board device
dx_SetRecordNotifyBeepTone( )

Description

The dx_SetRecordNotifyBeepTone( ) function specifies the template of the cadenced tone to be used as the record notification beep tone during subsequent calls to the Voice record functions. This function overwrites the default template used on DM3 boards. If no template is specified, the default beep tone has these specifications: 1400 Hz, -18 dB, 420 msecs on, 15 secs off.

The RM_NOTIFY flag in the mode parameter of various Voice record functions is used to instruct these functions to generate a record notification beep tone.

Note:
The amplitude for the beep tone specified in the TN_GEN structure is reduced by 9 dB due to the high impedance telephone interface. Therefore, if you require an amplitude of -18 dB, you must specify the value of -9 dB in the TN_GEN structure. It is not recommended that you specify a value higher than -8 dB (such as -7 dB or -6 dB) as this can produce a distorted beep tone on the line.

<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>tngencadp</td>
<td>points to a TN_GENCAD structure which contains parameters for the cadenced tone generation template. This structure in turn uses the TN_GEN structure which specifies single-frequency or dual-frequency tone definitions.</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.

**Example**

For an example of how to use this function, see the example code for `dx_playtoneEx()`.

**See Also**

- `dx_playtoneEx()`
set conditions that adjust speed or volume of play — dx_setsvcond()

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
dxxxlib.h

Category: Speed and Volume
Mode: synchronous
Platform: DM3, Springware

Description

The `dx_setsvcond()` function sets adjustments and adjustment conditions for all subsequent plays on the specified channel (until changed or cancelled).

An adjustment is a modification to play speed, play volume, or play (pause/resume) due to an adjustment condition such as start of play, or the occurrence of an incoming digit during play. This function uses the specified channel’s Speed or Volume Modification Table. For more information about these tables, see the Voice API Programming Guide.

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):

- which Speed or Volume Modification Table to use (speed or volume)
- adjustment type (increase/decrease, absolute value, toggle, pause/resume)
- adjustment conditions (incoming digit, beginning of play)
- level/edge sensitivity for incoming digits

See DX_SVCB, on page 514, 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 speed and 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 speed and volume is based on the most recent modification.
dx_setsvcond( ) — set conditions that adjust speed or volume of play

<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>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

- On DM3 boards, 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.
- On DM3 boards, digits that are used for play adjustment may also be used as a terminating condition. If a digit is defined as both, then both actions are applied upon detection of that digit.
- On Springware boards, digits that are used for play adjustment will not be used as a terminating condition. If a digit is defined as both, then the play adjustment will take priority.
- On DM3 boards, when adjustment is associated with a DTMF digit, speed can be increased or decreased in increments of 1 (10%) only.
- On DM3 boards, 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 speed 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( ) or ATDX_BUFDIGS( ).

### 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; use dx_fileerrno( ) to obtain error value
**Example**

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

/*
 * 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_VOLMETBL | SV_ABSPOS, 1, '3', 0 }, /* 3 */
    { SV_SPEEDTBL | SV_ABSPOS, -1, '4', 0 }, /* 4 */
    { SV_SPEEDTBL | SV_ABSPOS, 1, '5', 0 }, /* 5 */
    { SV_VOLMETBL | SV_RELCURPOS, 1, '6', 0 }, /* 6 */
    { SV_SPEEDTBL | SV_RELCURPOS, -1, '7', 0 }, /* 7 */
    { SV_SPEEDTBL | SV_ABSPOS, 6, '8', 0 }, /* 8 */
    { SV_VOLMETBL | SV_RELCURPOS, -1, '9', 0 }, /* 9 */
    { SV_SPEEDTBL | SV_ABSPOS, 10, '0', 0 }, /* 10 */
};

main()
{
    int dxxxdev;

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

    /*
     * Set Speed and Volume Adjustment Conditions
     */
    if ( dx_setsvcond( dxxxdev, 10, svcb ) == -1 ) {
        printf( "Unable to Set Speed and Volume\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" );
        exit( 0 );
    }

    /* Terminate the Program */
    exit( 0 );
}
```
dx_setsvcond( ) — set conditions that adjust speed or volume of play

- dx_clrsvcond( )
- DX_SVCB structure
- dx_pause( )
- dx_resume( )
- dx_setsvmt( )
- dx_getcursv( )
- dx_getsvmt( )
- dx_adjsv( )
- speed and volume modification tables in Voice API Programming Guide
dx_setsvmt( )

Name: int dx_setsvmt(chdev, tabletype, svmtp, flag)

Inputs: int chdev  • valid channel device handle
unsigned short tabletype • type of table to update (speed or volume)
DX_SVMT * svmtp • pointer to speed or volume modification table to modify
unsigned short flag • optional modification flag

Returns: 0 if success
-1 if failure

Includes: srllib.h
dxxlib.h

Category: Speed and Volume
Mode: synchronous
Platform: DM3, Springware

Description

The dx_setsvmt( ) function updates the speed or volume modification table for a channel using the values contained in a specified DX_SVMT structure.

This function can modify the speed or volume modification table so that the following occurs:

• When speed or volume adjustments reach their highest or lowest value, wrap the next adjustment to the extreme opposite value. For example, if volume reaches a maximum level during a play, the next adjustment would modify the volume to its minimum level.

• Reset the speed or volume modification table to its default values. Defaults are listed in the Voice API Programming Guide.

For more information on speed and volume modification tables, refer to DX_SVMT, on page 517, and see also the Voice API Programming Guide.

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>
</tbody>
</table>
| tabletype | specifies whether to update the speed modification table or the volume modification table:  
SV_SPEEDTBL – update the speed modification table values  
SV_VOLUMETBL – update the volume modification table values |
dx_setsvmt( ) — change default values of the speed or volume modification table

Parameter | Description
--- | ---
svtmp | points to the DX_SVMT structure whose contents are used to update either the speed or volume modification table

This structure is not used when SV_SETDEFAULT has been set in the flag parameter.

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 svtmp is ignored.
- SV_WRAPMOD – wrap around the speed or volume adjustments that occur at the top or bottom of the speed or volume modification table.

Note: Set flag to 0 if you do not want to use either SV_WRAPMOD or SV_SETDEFAULT.

Cautions

On DM3 boards, if you close a device via dx_close( ) after modifying speed and volume table values using dx_setsvmt( ), the dx_getcursv( ) function may return incorrect speed and volume settings for the device. This is because the next dx_open( ) resets the speed and volume tables to their default values. Therefore, it is recommended that you do not issue a dx_close( ) during a call where you have modified speed and 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; use dx_fileerrno( ) to obtain error value
Example

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

/*
 * Global Variables
 */

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

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

    /*
     * Set up the Speed/Volume Modification
     */
    memset( &svmt, 0, sizeof( DX_SVMT ) );
    svmt.decrease[ 0 ] = -128;
    svmt.decrease[ 1 ] = -128;
    svmt.decrease[ 2 ] = -128;
    svmt.decrease[ 3 ] = -128;
    svmt.decrease[ 4 ] = -128;
    svmt.decrease[ 5 ] = -20;
    svmt.decrease[ 6 ] = -16;
    svmt.decrease[ 7 ] = -12;
    svmt.decrease[ 8 ] = -8;
    svmt.decrease[ 9 ] = -4;
    svmt.origin = 0;
    svmt.increase[ 0 ] = 4;
    svmt.increase[ 1 ] = 8;
    svmt.increase[ 2 ] = 10;
    svmt.increase[ 3 ] = -128;
    svmt.increase[ 4 ] = -128;
    svmt.increase[ 5 ] = -128;
    svmt.increase[ 6 ] = -128;
    svmt.increase[ 7 ] = -128;
    svmt.increase[ 8 ] = -128;
    svmt.increase[ 9 ] = -128;

    /*
     * Update the Volume Modification Table without Wrap Mode.
     */
    if ( (dx_setsvmt( dxxxdev, SV_VOLUMETBL, &svmt, 0 ) == -1) {
        printf( "Unable to Set the Volume Modification Table\n" );
        ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev )
        dx_close( dxxxdev );
        exit( 1 );
    }

    /*
     * Continue Processing
     */
}
dx_setsvmt() — change default values of the speed or volume modification table

/*
 * 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()
- speed and volume modification tables in Voice API Programming Guide
- DX_SVMT data structure
The `dx_settonelen()` function changes the duration of the built-in beep tone (sometimes referred to as a pre-record beep), which some application programs make use of to indicate the start of a recording or playback.

When a record or playback function specifies RM_TONE or PM_TONE (respectively) in the `mode` parameter, a beep tone will be transmitted immediately before the record or play is initiated. The duration of the beep tone can be altered by this function.

A device handle is not used when calling `dx_settonelen()`. The beep tone will be modified for all voice resources used in the current process. The beep tone will not be affected in other processes.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tonelength</code></td>
<td>specifies the duration of the tone in 200 ms units.</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**

None.
example

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

int chdev; /* channel descriptor */
DV_TPT tpt; /* termination parameter table */
DX_XPB xpb; /* I/O transfer parameter block */
.
.
.
/* Increase beep tone len to 800ms */
dx_settonelen (4);

/* Open channel */
if ((chdev = dx_open("dxxxB1C1",0)) == -1) {
    printf("Cannot open channel\n"),
    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(2);
}

/* Start playback */
if (dx_playwav(chdev,"HELLO.WAV", &tpt, PM_TONE|EV_SYNC) == -1) {
    printf("Error playing file - \%s\n", ATDV_ERRMSGP(chdev));
    exit(3);
}

/* clear digit buffer */
dx_clrdigbuf(chdev);

/* Start 6KHz ADPCM recording */
if (dx_recvox(chdev,"MESSAGE.VOX", &tpt, NULL, KM_TONE|EV_SYNC) == -1) {
    printf("Error recording file - \%s\n", ATDV_ERRMSGP(chdev));
    exit(4);
}

/* hang up the phone*/
if (dx_sethook (chdev,DX_ONHOOK,EV_SYNC)) {
    printf("Error putting phone on hook - \%s\n", ATDV_ERRMSGP(chdev));
    exit(5);
}

/* close the channel */
if (dx_close (chdev,DX_ONHOOK,EV_SYNC)) {
    printf("Error closing channel - \%s\n", ATDV_ERRMSGP(chdev));
    exit(6);
}

see also

- dx_play()
- dx_playiottdata()
change the duration of the built-in beep tone — dx_settonelen()
dx_setuio() — install user-defined I/O functions

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

Platform: DM3, Springware

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 519 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 509 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.
- On DM3 boards, 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.
install user-defined I/O functions — dx_setuio()

Example

```c
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>  /* voice library header file */
#include <windows.h>

int cd;                 /* channel descriptor */
DX_UIO myio;            /* user definable I/O structure */

/*
 * User defined I/O functions
 */
int my_read(int fd, char * ptr, unsigned cnt)
int fd;
char * ptr;
unsigned cnt;
{
  printf("My read\n");
  return(read(fd,ptr,cnt));
}

/*
 * my write function
 */
int my_write(int fd, char * ptr, unsigned cnt)
int fd;
char * ptr;
unsigned cnt;
{
  printf("My write \n");
  return(write(fd,ptr,cnt));
}

/*
 * my seek function
 */
long my_seek(int fd, long offset, int whence)
int fd;
long offset;
int whence;
{
  printf("My seek\n");
  return(lseek(fd,offset,whence));
}

void main(argc, argv)
int argc;
char * argv[];
{
  /* Other initialization */
  
  DX_UIO uioblk;
  
  /* Initialize the UIO structure */
  uioblk.u_read = my_read;
  uioblk.u_write = my_write;
  uioblk.u_seek = my_seek;

  /* Install my I/O routines */
  dx_setuio(uioblk);
  vodat_fd = dx_fileopen("JUNK.VOX",O_RDWR|O_BINARY);
```

dx_setuio( ) — install user-defined I/O functions

/*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*/
iottt++;
iottt->io_type = IO_DEV|IO_UIO|IO_CONT
iottt->io_fhandle = vodat_fd;
iott->io_offset = 20001;
iott->io_length = 20000;

/*This block uses standard I/O functions*/
iottt++
iottt->io_type = IO_DEV|IO_CONT
iottt->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("dxxCB1C1", 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_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

**Platform:** DM3

---

### 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 **Voice API Programming Guide**.

The application receives TDX_LOWWATER and TDX_HIGHWATER events regardless of whether or not **dx_SetWaterMark( )** is used in your application. These events are generated when there is a play operation with this buffer and are reported on the device that is performing the play. If there is no active play, the application will not receive any of these events.

<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.
dx_SetWaterMark( ) — set water mark for the circular stream buffer

- **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 <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_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( )

**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
- dxxxlib.h

**Category:** I/O

**Mode:** asynchronous or synchronous

**Platform:** DM3, Springware

---

**Description**

The `dx_stopch()` function forces termination of currently active I/O functions on a channel. It forces a channel in the busy state to become idle. If the channel specified in `chdev` already is idle, `dx_stopch()` has no effect and will return a success.

Running this function asynchronously will initiate the `dx_stopch()` without affecting processes on other channels.

Running this function synchronously within a process does not block other processing. Other processes continue to be serviced.

When 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.
dx_stopch() — force termination of currently active I/O functions

Parameter | Description
-----------|--------------------------------------------------
chdev      | specifies the valid channel device handle obtained when the channel was opened using dx_open()
mode       | a bit mask that specifies the mode:
            | • EV_SYNC – synchronous mode
            | • EV_ASYNC – asynchronous mode. The stop will be issued, but the driver does not “sleep” and wait for the channel to become idle before dx_stopch() returns.
            | • EV_NOSTOP – if this bit is set and the channel is idle, TDX_NOSTOP event is generated.
            | • EV_STOPGETEV — if this bit is set and dx_stopch() is issued during dx_getevt(), TDX_CST event is generated with reason of DE_STOPGETEV.
            | • EV_STOPWTRING — if this bit is set and dx_stopch() is issued during dx_wtring(), EDX_WTRINGSTOP error is be generated.
            | • IGNORESTATE — 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.

**Cautions**

- dx_stopch() has no effect on a channel that has any of the following functions issued:
  - dx_dial() without call progress analysis enabled
  - dx_wink()

  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 EVASYNC.

- An application can use dx_stopch() from within a signal handler to stop the dx_getevt() and dx_wtring() functions. To do so, “OR” the mode flag with the EV_STOPGETEV and EV_STOPWTRING flags, respectively, to stop these functions. In these cases, dx_getevt() will successfully return with the event DE_STOPGETEV while dx_wtring() will fail with a return value of -1 and the lasterr will be set to EDX_WTRINGSTOP.

**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
force termination of currently active I/O functions — dx_stopch()

EDX_SYSTEM
Error from operating system; use dx_fileerrno() to obtain error value

Example

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

main()
{
    int chdev, srlmode;

    /* Set SRL to run in polled mode. */
    srlmode = SR_POLLMODE;
    if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1) {
        /* process error */
    }

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

    /* continue processing */

    /* Force the channel idle. The I/O function that the channel is
     * executing will be terminated, and control passed to the handler
     * function previously enabled, using sr_enbhdlr(), for the
     * termination event corresponding to that I/O function.
     * In the asynchronous mode, dx_stopch() returns immediately,
     * without waiting for the channel to go idle. */
    if (dx_stopch(chdev, EV_ASYNC) == -1) {
        /* process error */
    }
}
```

See Also

- dx_dial()
- dx_getdig()
- dx_play()
- dx_playf()
- dx_playiottdata()
- dx_playtone()
- dx_playvoxx()
- dx_rec()
- dx_recf()
- dx_reciottdata()
- dx_recvoxx()
- dx_wink()
- ATDX_TERMMSK()
dx_stopch() — force termination of currently active I/O functions

- ATDX_CPTERM() - dx_dial() with call progress analysis
dx_TSFStatus( )

Name: int dx_TSFStatus ( void )
Inputs: None
Returns: 0 if TSF loading was successful
         non-zero value if TSF loading failed; see EDX error codes for reason
Includes: srllib.h
dxslib.h
Category: Configuration
Mode: synchronous
Platform: Springware

Description
The dx_TSFStatus( ) function returns the status of tone set file loading. Tone set file (TSF) loading is an optional procedure used to customize the default call progression analysis tone definitions with TSF Tone definitions created by the PBX Expert utility. TSF loading occurs when you execute your application and a valid, existing TSF was configured and enabled in the configuration manager (DCM).

Cautions
None.

Errors
If this function returns a negative value (corresponding to the EDX_ define below), it indicates that the TSF failed to load for one of the following error reasons:

EDX_SYSTEM
   Error from operating system; use dx_fileerrno( ) to obtain error value. Failed to load PBXPERT.DLL.

EDX_BADREGVALUE
   Unable to locate value in registry. The configuration manager (DCM) does not specify a TSF name and therefore the registry either doesn’t contain a value for “TSF Download File” or the PBX Expert key is missing.

EDX_BADTSFFILE
   The TSF specified in the configuration manager (DCM) does not exist or is not a valid TSF file.

EDX_BADTSFDATA
   TSF data not consolidated. The TSF specified in the configuration manager (DCM) does not contain valid downloadable data.
**dx_TSFStatus( ) — return the status of tone set file loading**

EDX_FEATUREDISABLED
The TSF feature is disabled in the configuration manager (DCM).

### Example

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

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

main ( )
{
    int rc;

    rc = dx_TSFStatus ( );
    switch ( rc )
    {
    case 0:
        break;

    case EDX_SYSTEM:
        printf ( "General system error loading PBXpert.DLL \n");
        break;

    case EDX_BADREGVALUE:
        printf ( "Cannot find PBX Expert registry entry\n"");
        break;

    case EDX_BADTSFFILE:
        printf ( "Downloadable filename in registry invalid or does not exist \n"");
        break;

    case EDX_BADTSFDATA:
        printf ( "Downloadable TSF file does not contain valid consolidated data\n");
        break;

    case EDX_FEATUREDISABLED:
        printf ( "TSF feature is disabled in Intel Dialogic Configuration Manager\n");
        break;

    default:
        break;
    }
}
```

### See Also

- `dx_initcallp( )`
**dx_TxIottData( )**

**Name:** int dx_TxIottData(chdev, iottp, lpTerminations, wType, lpParams, mode)

**Inputs:**
- int chdev: valid channel device handle
- DX_IOTT *iottp: pointer to I/O Transfer Table
- DV_TPT *lpTerminations: pointer to Termination Parameter Table
- int wType: data type
- LPVOID lpParams: pointer to data type-specific information
- int mode: function mode

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

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

**Category:** Analog Display Services Interface (ADSI)

**Mode:** asynchronous or synchronous

**Platform:** DM3, Springware

---

**Description**

The **dx_TxIottData( )** function is used to transmit data on a specified channel. The data may come from any combination of data files, memory, or custom devices. The **wType** parameter specifies the type of data to be transmitted, for example ADSI data. The **iottp** parameter specifies the messages to be transmitted.

Upon asynchronous completion of **dx_TxIottData( )**, the TDX_TXDATA event is posted. Use **ATDX_TERMMSK( )** to return the reason for the last I/O function termination on the channel. Possible return values are:

- **TM_EOD**
  - End of FSK data detected on transmit
- **TM_ERROR**
  - I/O device error
- **TM_MAXDATA**
  - Maximum data reached; returned when the last I/O function terminates on DX_MAXDATA
- **TM_MAXTIME**
  - Maximum function time exceeded
- **TM_USRSTOP**
  - Function stopped by user
Library level data is buffered when it is received. The buffer size is 255, which is the default buffer size used by the library.

**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 (pointer to I/O transfer table)
- `EDX_BADPARM`: Invalid data mode
- `EDX_BUSY`: Channel already executing I/O function
- `EDX_SYSTEM`: Error from operating system; use `dx_fileerrno()` to obtain error value

**Example**

```c
// Synchronous transmit ADSI data
```
transmit data on a specified channel — dx_TxIottData( )

#include "srllib.h"
#include "dxxxlib.h"

main()
{
    DX_IOTT iott = {0};
    char *devnamep = "dxxxB1C1";
    char buffer[16];
    ADSI_XFERSTRUC adsimode;
    int chdev;
    
    sprintf(buffer, "MENU.ADSI");
    if ((iott.io_fhandle = dx_fileopen(buffer, _O_RDONLY|O_BINARY)) == -1) {
        /* Perform system error processing */
        exit(1);
    }
    if ((chdev = dx_open(devnamep, 0)) == -1) {
        fprintf(stderr, "Error opening channel %s
", devnamep);
        dx_fileclose(iott.io_fhandle);
        exit(2);
    }
    // source is a file
    iott.io_type = IO_DEV|IO_EOT;
    iott.io_bufp = 0;
    iott.io_offset = 0;
    iott.io_length = -1;
    adsimode.cbSize = sizeof(adsimode);
    adsimode.dwTxDataMode = ADSI_ALERT;  // send out ADSI data with CAS
    printf("Waiting for incoming ring\n");
    dx_wtring(chdev, 2, DX_OFFHOOK, chdev);
    if (dx_TxIottData(chdev, &iott, NULL, DT_ADSI, &adsimode, EV_SYNC) < 0) {
        fprintf(stderr, "ERROR: dx_TxIottData failed on Channel %s; error:
        %s\n" , ATDV_NAMEP(chdev), ATDV_ERRMSGP(chdev));
    }
}

See Also

- dx_RxIottData( )
- dx_TxRxIottData( )
dx_TxRxIottData( ) — start a transmit-initiated reception of data

**dx_TxRxIottData( )**

**Name:** int dx_TxRxIottData(chdev, lpTxIott, lpTxTerminations, lpRxIott, lpRxTerminations, wType, lpParams, mode)

**Inputs:**
- int chdev: valid channel device handle
- DX_IOTT *lpTxIott: pointer to I/O Transfer Table
- DV_TPT *lpTxTerminations: pointer to Termination Parameter Table
- DX_IOTT *lpRxIott: pointer to I/O Transfer Table
- DV_TPT *lpRxTerminations: pointer to Termination Parameter Table
- int wType: data type
- LPVOID lpParams: pointer to data type-specific information
- int mode: function mode

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

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

**Category:** Analog Display Services Interface (ADSI)

**Mode:** asynchronous or synchronous

**Platform:** DM3, Springware

---

**Description**

The **dx_TxRxIottData()** function is used to start a transmit-initiated reception of ADSI two-way FSK (Frequency Shift Keying) data, where faster remote terminal device (CPE) turnaround occurs, typically within 100 msec. Faster turnaround is required for two-way FSK so that the receive data is not missed while the application turns the channel around after the last sample of FSK transmission is sent.

The **wType** parameter specifies the type of data that will be transmitted and received; that is, two-way ADSI. The transmitted data may come from and the received data may be directed to any combination of data files, memory, or custom devices. The data is transmitted and received on a specified channel.

<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>lpTxIott</td>
<td>Points to the I/O Transfer Table structure. lpTxIott specifies the location of the messages to be transmitted. This is the same DX_IOTT structure used in dx_playiottdata() and dx_reciottdata(). See DX_IOTT, on page 509, for more information on this data structure.</td>
</tr>
</tbody>
</table>
start a transmit-initiated reception of data —\texttt{dx\_TxRxIottData( )}

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| \texttt{lpTxTerminations}     | Points to the Termination Parameter Table structure, \texttt{DV\_TPT}, which specifies termination conditions for the device handle. Supported values are:  
\begin{itemize}   
\item DX\_MAXTIME   
\item DX\_MAXDATA (valid values are 1 - 65535 for tp\_length field) \end{itemize}  
For more information on this structure, see \texttt{DV\_TPT}, on page 485. |
| \texttt{lpRxIott}             | Points to the I/O Transfer Table structure. \texttt{lpRxIott} specifies the destination of the messages to be received. This is the same DX\_IOTT structure used in \texttt{dx\_playiottdata( )} and \texttt{dx\_reciottdata( )}. See \texttt{DX\_IOTT}, on page 509, for more information on this data structure. |
| \texttt{lpRxTerminations}     | Points to the Termination Parameter Table structure, \texttt{DV\_TPT}, which specifies termination conditions for the device handle. Supported values are:  
\begin{itemize}   
\item DX\_MAXTIME   
\item DX\_MAXDATA (valid values are 1 - 65535 for tp\_length field) \end{itemize}  
For more information on this structure, see \texttt{DV\_TPT}, on page 485. |
| \texttt{wType}                | Specifies the type of data to be transmitted and received. To transmit and receive ADSI data, set \texttt{wType} to \texttt{DT\_ADSI}.                                                                                           |
| \texttt{lpParams}             | Points to a structure that specifies additional information about the data that is to be sent and received. The structure type is determined by the data type (ADSI) specified by \texttt{wType}. For ADSI data, set \texttt{lpParams} to point to an ADSI\_XFERSTRUC parameter block structure. For more information on this structure, see \texttt{ADSI\_XFERSTRUC}, on page 478. |
| \texttt{mode}                 | Specifies how the function should execute, either \texttt{EV\_ASYNC} (asynchronous) or \texttt{EV\_SYNC} (synchronous).                                                                                                   |

The transmit portion of the \texttt{dx\_TxRxIottData( )} function will continue until one of the following occurs:  
\begin{itemize}   
\item all data specified in DX\_IOTT has been transmitted   
\item \texttt{dx\_stopch( )} is issued on the channel   
\item one of the conditions specified in \texttt{DV\_TPT} is satisfied \end{itemize}  

The receive portion of the \texttt{dx\_TxRxIottData( )} function will continue until one of the following occurs:  
\begin{itemize}   
\item \texttt{dx\_stopch( )} is called   
\item the data requirements specified in the DX\_IOTT are fulfilled   
\item the channel detects end of FSK data   
\item one of the conditions in the \texttt{DV\_TPT} is satisfied \end{itemize}
dx TxRxIottData( ) — start a transmit-initiated reception of data

If the channel detects end of FSK data during the receive portion, the function is terminated. Use ATDX_TERMMSK( ) to return the reason for the last I/O function termination on the channel. Possible return values are:

- **TM_EOD**
  - End of FSK data detected on transmit or receive
- **TM_ERROR**
  - I/O device error
- **TM_MAXDATA**
  - Maximum data reached; returned when the last I/O function terminates on DX_MAXDATA
- **TM_MAXTIME**
  - Maximum function time exceeded
- **TM_USRSTOP**
  - Function stopped by user

Upon asynchronous completion of the transmit portion of the function, a TDX_TXDATA event is generated. Upon asynchronous completion of the receive portion of the function, a TDX_RXDATA event is generated.

### Cautions

- Library level data is buffered when it is received. The buffer size is 255, which is the default buffer size used by the library.
- When using dx TxRxIottData( ) in asynchronous mode, note the following:
  - If the FSK transmission is completed with a termination mask value of TM_MAXTIME, TM_MAXDATA or TM_EOD, then the channel automatically initiates a receive session. On completion of the receive session, a TDX_RXDATA event will be generated.
  - If the FSK transmission is completed with a termination mask value of TM_USRSTOP or TM_ERROR, then the channel does not initiate a receive session and the TDX_RXDATA event will not be generated.

### 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 (pointer to I/O transfer table)
- **EDX_BADPARM**
  - Invalid data mode
- **EDX_BUSY**
  - Channel already executing I/O function
- **EDX_SYSTEM**
  - Error from operating system; use dx fileerrno( ) to obtain error value
Example

// Synchronous transmit initiated receive ADSI data

#include "srllib.h"
#include "dxxxlib.h"

main()
{

DX_IOTT TxIott = {0};
DX_IOTT RxIott = {0};
DV_TPT tpt;
char *devnamep = "dxxxB1C1";
char buffer[16];
ADSI_XFERSTRUC adsimode;
int chdev;
.
.
.
sprintf(buffer, "MENU.ADSI");
if ((TxIott.io_fhandle = dx_fileopen(buffer, _O_RDONLY|O_BINARY)) == -1) {
   /* Perform system error processing */
   exit(1);
}

sprintf(buffer, "RECEIVE.ADSI");
if ((RxIott.io_fhandle = dx_fileopen(buffer, O_BINARY)) == -1) {
   /* Perform system error processing */
   dx_fileclose(TxIott.io_fhandle);
   exit(2);
}

if ((chdev = dx_open(devnamep, 0)) == -1) {
   fprintf(stderr, "Error opening channel %s\n", devnamep);
   dx_fileclose(TxIott.io_fhandle);
   dx_fileclose(RxIott.io_fhandle);
   exit(1);
}
.
.
.
// source is a file
TxIott.io_type = IO_DEV|IO_EOT;
TxIott.io_bufp = 0;
TxIott.io_offset = 0;
TxIott.io_length = -1;

// destination is a file
RxIott.io_type = IO_DEV|IO_EOT;
RxIott.io_bufp = 0;
RxIott.io_offset = 0;
RxIott.io_length = -1;

adsimode.cbSize = sizeof(adsimode);
adsimode.dwTxDataMode = ADSI_ALERT;
adsimode.dwRxDataMode = ADSI_NOALERT;

// Specify maximum time termination condition in the TPT for the
// receive portion of the function. Application specific value is
// used to terminate dx_TxRxIottData() if end of data is not
// detected over a specified duration.
tpt.tp_type = IO_EOT;
dx_TxRxlottData() — start a transmit-initiated reception of data

if (dx_clrtpt(&tpt, 1) == -1) {
    // Process error
}

tpt.tp_termno = DX_MAXTIME;
tpt.tp_length = 1000;
tpt.tp_flags = TF_MAXTIME;

printf("Waiting for incoming ring\n");
dx_wtring(chdev, 2, DX_OFFHOOK, -1);

if (dx_TxRxIottData(chdev, &TxIott, NULL, &RxIott, &tpt, DT_ADSI,
        &adsmode, EV_SYNC) < 0) {
    fprintf(stderr, "ERROR: dx_TxRxIottData failed on Channel \s; error: \s\n", ATDV_NAMEP(chdev), ATDV_ERRMSGP(chdev));
}

See Also

- dx_TxIottData()
- dx_RxIottData()
**dx_unlisten( )**

<table>
<thead>
<tr>
<th>Name:</th>
<th>int dx_unlisten(chdev)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs:</td>
<td>int chdev</td>
</tr>
<tr>
<td></td>
<td>• voice channel device handle</td>
</tr>
<tr>
<td>Returns:</td>
<td>0 on success</td>
</tr>
<tr>
<td></td>
<td>-1 on error</td>
</tr>
<tr>
<td>Includes:</td>
<td>srllib.h</td>
</tr>
<tr>
<td></td>
<td>dxxxlib.h</td>
</tr>
<tr>
<td>Category:</td>
<td>TDM Routing</td>
</tr>
<tr>
<td>Mode:</td>
<td>synchronous</td>
</tr>
<tr>
<td>Platform:</td>
<td>DM3, Springware</td>
</tr>
</tbody>
</table>

### 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.
- On DM3 boards, this function is supported in a flexible routing configuration but not a fixed routing configuration. This document assumes that a flexible routing configuration is the configuration of choice. For more information on API restrictions in a fixed routing configuration, see the *Voice API Programming Guide*.

### 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
dx_unlisten( ) — disconnect voice receive channel from TDM bus

EDX_SH_BADINDX
Invalid Switch Handler index number

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 failback failed

EDX_SYSTEM
Error from operating system; use dx_fileerrno() to obtain error value

■ Example

```c
#include <windows.h>
#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( )
**dx_unlistenecr( )**

**Name:** int dx_unlistenecr(chdev)

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

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

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

**Category:** Echo Cancellation Resource

**Mode:** synchronous

**Platform:** Springware

---

**Description**

The **dx_unlistenecr( )** function disables echo cancellation resource (ECR) mode on the voice channel and sets the channel back into standard voice processing (SVP) mode echo cancellation.

**Notes:**

1. Calling the **dx_listenecr( )** or **dx_listeneceex( )** 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_unlistenecr( )** function.

2. The ECR functions have been replaced by the continuous speech processing (CSP) API functions. CSP provides enhanced echo cancellation. For more information, see the *Continuous Speech Processing API Programming Guide* and *Continuous Speech Processing API Library Reference*.

**Parameter**

<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**

This function fails when:

- An invalid channel device handle is specified.
- The ECR feature is not enabled on the board specified.
- The ECR feature is not supported on the board 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**
  - Invalid parameter
**dx_unlistenecr() — disable echo cancellation resource (ECR) mode**

EDX_SH_BADCMD  
Function is not supported in current bus configuration

EDX_SH_BADEXTTS  
TDM bus time slot is not supported at current clock rate

EDX_SH_BADINDEX  
Invalid Switch Handler index number

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 function 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; use dx_fileerrno() to obtain error value

### Example

```c
#include <stdio.h>
#include <windows.h>
#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) {
        /* Perform system error processing */
        exit(1);
    }
```

/* Disconnect echo-reference receive of board 1, channel 1 from the TDM bus, and stop the ECR feature */
if (dx_unlistenecr(chdev) == -1) {
    printf("Error message = %s", ATDV_ERRMSGP(chdev));
    exit(1);
}
return(0);

See Also

- dx_listenecr()
- dx_listenecrex()
dx_wink( ) — generate an outbound wink

**dx_wink( )**

**Name:** int dx_wink(chdev, mode)

**Inputs:**
- int chdev • valid channel device handle
- unsigned short mode • synchronous/asynchronous setting

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

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

**Category:** I/O

**Mode:** asynchronous or synchronous

**Platform:** Springware

---

### Description

The **dx_wink( )** function generates an outbound wink on the specified channel. A wink from a voice board is a momentary rise of the A signaling bit, which corresponds to a wink on an E&M line. A wink’s typical duration of 150 to 250 milliseconds is used for communication between the called and calling stations on a T-1 span.

**Note:** Do not call this function on a non-E&M line or for a TDM bus T-1 digital interface device such as on an Intel® Dialogic® D/240SC-2T1 board. Transparent signaling for TDM bus digital interface devices is not supported. See the *Digital Network Interface Software Reference* for information about E&M lines.

**Parameter** | **Description**
---|---
chdev | specifies the valid channel device handle obtained when the channel was opened using **dx_open( )**
mode | specifies whether to run **dx_wink( )** asynchronously or synchronously:
  - EV_ASYNC – run asynchronously
  - EV_SYNC – run synchronously (default)

**Notes:**

1. The **dx_wink( )** function is supported on a T-1 E&M line connected to a DTI/101 board. In addition, the **dx_wink( )** function is supported on the DTI/211 board in transparent mode.

2. All values referenced for this function are subject to a 10 msec clocking resolution. Actual values will be in a range: (parameter value - 9 msec) ≤ actual value ≤ (parameter value)

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

To run this function asynchronously set the **mode** parameter to EV_ASYNC. When running asynchronously, this function will return 0 to indicate it has initiated successfully, and will generate...
generate an outbound wink — dx_wink( )

a TDX_WINK termination event to indicate completion. Use the Standard Runtime Library (SRL) Event Management functions to handle the termination event.

For more information on wink signaling, such as how to set delay prior to wink, see the Voice API Programming Guide.

■ Cautions

Make sure the channel is on-hook when dx_wink() is called.

■ 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; use dx_fileerrno() to obtain error value

■ Example 1

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

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

main()
{
    int chdev;
    DV_TPT tpt;
    DV_DIGIT digitp;
    char buffer[8];

    /* open a channel with chdev as descriptor */
    if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
        /* process error */
    }

    /* set hookstate to on-hook and wink */
    if (dx_sethook(chdev,DX_ONHOOK,EV_SYNC) == -1) {
        /* process error */
    }

    if (dx_wink(chdev,EV_SYNC) == -1) {
        /* error winking channel */
    }
    dx_clrtpt(&tpt,1);

    /* set up DV_TPT */
    tpt.tp_type   = IO_EOT;         /* only entry in the table */
    tpt.tp_termno = DX_MAXDTMF;     /* Maximum digits */
    tpt.tp_length = 1;              /* terminate on the first digit */
    tpt.tp_flags  = TF_MAXDTMF;     /* Use the default flags */
```
Example 2

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

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

#define MAXCHAN 24

int wink_handler();

main()
{
    int i, chdev[MAXCHAN];
    char *chnamep;
    int srlmode;

    /* Set SRL to run in polled mode. */
    srlmode = SR_POLLMODE;
    if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1) {
        /* process error */
    }

    for (i=0; i<MAXCHAN; i++) {
        /* Set chnamep to the channel name - e.g., dxxxB1C1 */
        /* open the channel with dx_open(). Obtain channel device
         * descriptor in chdev[i]
         */
        if ((chdev[i] = dx_open(chnamep,NULL)) == -1) {
            /* process error */
        }

        /* Using sr_enbhdlr(), set up handler function to handle wink
         * completion events on this channel. */
        if (sr_enbhdlr(chdev[i], TDX_WINK, wink_handler) == -1) {
            /* process error */
        }

        /* Before issuing dx_wink(), ensure that the channel is onhook,
         * else the wink will fail. */
        if(dx_sethook(chdev[i], DX_ONHOOK, EV_ASYNC)==-1){
            /* error setting channel on-hook */
        }

        /* Use sr_waitevt() to wait for the completion of dx_sethook(). */
        if ((dx_wink(chdev[i], EV_ASYNC) == -1) {
            /* error winking channel */
        }
    }
}
```
/* Use sr_waitev() to wait for the completion of wink.
 * On receiving the completion event, TDX_WINK, control is transferred
 * to the handler function previously established using sr_enbhdlr().
 */

int wink_handler()
{
    printf("wink completed on channel \%s\n", ATDX_NAMEP(sr_getevtdev()));
    return 0;
}

See Also

- dx_setparm()
- dx_getparm()
- event management functions in Standard Runtime Library API Library Reference
- DV_TPT data structure (to specify a termination condition)
- ATDX_TERMMSK()
- dx_wtring() (when handling outbound winks)
- dx_setevtmask() (when handling inbound winks)
- dx_sethook() (when handling inbound winks)
- DX_CST data structure (call status transition)
- dx_getevt() (for synchronous applications)
- DX_EBLK data structure (for synchronous applications)
**dx_wtcallid()** — wait for rings and report caller ID

### Description

The **dx_wtcallid()** function is a convenience function that waits for rings and reports caller ID, if available. Using this function is equivalent to using the voice functions **dx_setevtmsk( )** and **dx_getevt( )**, and the caller ID function **dx_gtcallid( )** to return the caller’s Directory Number (DN).

On successful completion, a NULL-terminated string containing the caller’s phone number is placed in the buffer pointed to by **bufferp**.

**Note:** Non-numeric characters (punctuation, space, dash) may be included in the number string. The string may not be suitable for dialing without 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 <strong>dx_open( )</strong></td>
</tr>
<tr>
<td>nrings</td>
<td>specifies the number of rings to wait before answering</td>
</tr>
</tbody>
</table>

**Valid values:**

- \( \geq 1 \) (Note: Minimum 2 for CLASS and ACLIP)
The `dx_wtcallid()` function is a caller ID convenience function provided to allow applications to wait for a specified number of rings (as set for the ring event) and returns the calling station’s Directory Number (DN).

Caller ID information is available for the call from the moment the ring event is generated (if the ring event is set to occur on or after the second ring (CLASS, ACLIP), or set to occur on or after the first ring (CLIP, JCLIP) until either of the following occurs:

- If the call is answered (the application channel goes off-hook), the caller ID information is available to the application until the call is disconnected (the application channel goes on-hook).
- If the call is not answered (the application channel remains on-hook), the caller ID information is available to the application until rings are no longer received from the Central Office (signaled by ring off event, if enabled).

### Cautions

- `dx_wtcallid()` changes the event enabled on the channel to DM_RINGS.
- If a checksum error occurs on the line, the API functions will fail and return EDX_CLIDINFO.
- Make sure the buffer is large enough to hold the DN returned by the function.
- If caller ID is enabled, on-hook digit detection (DTMF, MF, and global tone detection) will not function.

### 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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| `timeout` | specifies the maximum length of time to wait for a ring
  Valid values (0.1-second units):
  - ≥ 0
  - -1 waits forever; never times out
  If timeout is set to 0 and a ring event does not already exist, the function returns immediately. |
| `bufferp` | pointer to buffer where the calling line Directory Number (DN) is to be stored
  Note: The application must allocate a buffer large enough to accommodate the DN. |
dx_wtcallid( ) — wait for rings and report caller ID

EDX_CLIDBLK
Caller ID is blocked or private or withheld
(other information may be available using dx_gtextcallid( ))

EDX_CLIDINFO
Caller ID information not sent, sub-message(s) requested not available or caller ID
information invalid

EDX_CLIDOAA
Caller ID is out of area
(other information may be available using dx_gtextcallid( ))

EDX_SYSTEM
Error from operating system; use dx_fileerrno( ) to obtain error value

EDX_TIMEOUT
Time out limit is reached

### Example

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

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

unsigned char buffer[21]; /* char buffer */
int rc; /* value returned by function */
int chdev; /* channel descriptor */
unsigned short parmval; /* Parameter value */

/* open channel */
if ((chdev = dx_open("dxxxB1C1", NULL) == -1) {
  /* process error */
}
/* Enable Caller ID */
parmval = DX_CALLIDENABLE;
if (dx_setparm(chdev, DXCH_CALLID, (void *)&parmval) == -1) {
  /* process error */
}
/* sit and wait for two rings on this channel - no timeout */
if (dx_wtcallid(chdev,2,-1,buffer) == -1) {
  printf("Error waiting for ring (with Caller ID): 0x%x\n", ATDV_LASTERR(chdev));
  /* process error */
}
printf("Caller ID = %s\n", buffer);
```

### See Also
- dx_gtextcallid( )
- dx_setevtmsk( )
- dx_getevt( )
**dx_wtring()**

**Name:** int dx_wtring(chdev, nrings, hstate, timeout)

**Inputs:**
- int chdev: valid channel device handle
- int nrings: number of rings to wait for
- int hstate: hook state to set after rings are detected
- int timeout: timeout, in seconds

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

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

**Category:** Configuration

**Mode:** synchronous

**Platform:** Springware

---

**Description**

The `dx_wtring()` function waits for a specified number of rings and sets the channel to on-hook or off-hook after the rings are detected. Using `dx_wtring()` is equivalent to using `dx_setevtmsk()`, `dx_getevt()`, and `dx_sethook()` to wait for a ring. When `dx_wtring()` is called, the specified channel’s event is set to DM_RINGS in `dx_setevtmsk()`.

**Note:** Do not call this function for a digital T-1 TDM bus configuration that includes a D/240SC, D/240SC-T1, or DTI/241SC board. Transparent signaling for TDM bus digital interface devices is not supported.

An application can stop the `dx_wtring()` function from within a process or from another process, as follows:

- From within a process, a signal handler may issue a `dx_stopch()` with the handle for the device waiting in `dx_wtring()`. The mode parameter to `dx_stopch()` should be ORed with EV_STOPWTRING flag to stop `dx_wtring()`. The EV_STOPWTRING flag influences `dx_wtring()` only. It does not affect the existing functionality of `dx_stopch()`. Specifically, if a different function besides `dx_wtring()` is in progress when `dx_stopch()` is called with EV_STOPWTRING mode, that function will be stopped as usual. EV_STOPWTRING will simply be ignored if `dx_wtring()` is not in progress.

- From another process, `dx_wtring()` may be stopped using the inter-process event communication mechanism. The event-sending process should open the device that has issued `dx_wtring()` and call `dx_sendevt()` with its device handle to send the DE_STOPWTRING event.

Using either of the two mechanisms above, `dx_wtring()` will fail and return a -1. `lasterr` will be set to EDX_WTRINGSTOP.
**dx_wtring( ) — wait for a specified number of rings**

<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> function</td>
</tr>
<tr>
<td>nrings</td>
<td>specifies the number of rings to wait for before setting the hook state</td>
</tr>
</tbody>
</table>
| hstate    | sets the hookstate of the channel after the number of rings specified in `nrings` are detected. Valid values:  
  - DX_OFFHOOK – channel goes off-hook when `nrings` number of rings are detected  
  - DX_ONHOOK – channel remains on-hook when `nrings` number of rings are detected |
| timeout   | specifies the maximum length of time in tenths of seconds to wait for a ring. Valid values:  
  - number of seconds – maximum length of time to wait for a ring  
  - -1 – `dx_wtring( )` waits forever and never times out  
  - 0 – `dx_wtring( )` returns -1 immediately if a ring event does not already exist |

**Cautions**

- `dx_wtring( )` changes the event enabled on the channel to DM_RINGS. For example, process A issues `dx_setevtmsk( )` to enable detection of another type of event (such as DM_SILON) on channel one. If process B issues `dx_wtring( )` on channel one, then process A will now be waiting for a DM_RINGS event since process B has reset the channel event to DM_RINGS with `dx_wtring( )`.  
- A channel can detect rings immediately after going on hook. Rings may be detected during the time interval between `dx_sethook( )` and `dx_wtring( )`. Rings are counted as soon as they are detected.

If the number of rings detected before `dx_wtring( )` returns is equal to or greater than `nrings`, `dx_wtring( )` will not terminate. This may cause the application to miss calls that are already coming in when the application is first started.

- Do not use the `sigset( )` system call with SIGALRM while waiting for rings.

**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; use `dx_fileerrno( )` to obtain error value

- **EDX_TIMEOUT**  
  Timeout limit is reached
### Example

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

main()
{
    int chdev; /* channel descriptor */
    ..

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

    /* Wait for two rings on this channel - no timeout */
    if (dx_wtring(chdev,2,DX_OFFHOOK,-1) == -1) {
        /* process error */
    }
    ..
}
```

### See Also
- `dx_setevtmsk( )`
- `dx_getevt( )`
- `dx_sethook( )`
- `DX_EBLK` data structure
li_attendant( ) — perform the actions of an automated attendant

li_attendant( )

Name: int li_attendant(pAtt)

Inputs: DX_ATTENDANT *pAtt  • pointer to DX_ATTENDANT data structure

Returns: 0 if success
          EDX_BADPARM, EDX_BADPROD, EDX_SYSTEM, or -1 if failure

Includes: syntellect.h

Category: Syntellect License Automated Attendant

Mode: synchronous, multitasking

Platform: Springware

---

Description

The li_attendant( ) function performs the actions of an automated attendant. It is an implementation of an automated attendant application and works as a created thread. Before the application can create the thread, it must initialize the DX_ATTENDANT data structure.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pAtt</td>
<td>pointer to the Automated Attendant data structure, DX_ATTENDANT, that specifies termination conditions for this function and more.</td>
</tr>
</tbody>
</table>

This function loops forever or until the named event specified in the szEventName field of the DX_ATTENDANT data structure becomes signaled. While waiting for the named event to be signaled, this function checks for an incoming call. By default, it assumes that an analog front end is present and uses dx_setevtmsk( ) and dx_getevt( ) to determine if an incoming call is present.

The application can override the default analog front end behavior by supplying a function in the pfnWaitForRings field of the data structure.

Once an incoming call is detected, the call is answered. A voice file intro.att is played back, and li_attendant( ) waits for digit input. By default, dx_sethook( ) is called unless pfnAnswerCall is not NULL. The application can override the default analog front end behavior by supplying a function in the pfnAnswerCall field.

The maximum number of DTMF digits is specified in the nExtensionLength field. If timeout occurs or the maximum number is reached, the translation function in the pfExtensionMap field is called. The translated string, whose maximum length is nDialStringLength, is then dialed. The translation function should insert pauses and flash hook sequences where appropriate. The call is terminated using dx_sethook( ) unless pfnDisconnectCall is registered, and li_attendant( ) awaits the next incoming call. The application can override the default analog front end behavior by supplying a function in the pfnDisconnectCall field.
perform the actions of an automated attendant — li_attendant( )

### Cautions

- This function must supply values for all required fields in the `DX_ATTENDANT` structure.
- This function must supply an extension mapping function even if no extension translation is required. You should prefix the extension to be dialed with the “flash hook” character and possibly the “pause” character as well.
- The function does not return when a non-fatal error occurs during operation. The current call may be dropped but `li_attendant()` continues its operation. The application can choose to open the device on its own and use `ATDV_LASTERR()` to find out if the `li_attendant()` thread is experiencing trouble.

### Errors

This function fails and returns the specified error under the following conditions:

-1  
  Indicates one of the following:
  - Unable to open the device specified in the `szDevName` field
  - `pfnDisconnectCall` fails the first time around

**EDX_BADPARM**
  Indicates one of the following:
  - `pAtt` is NULL
  - `pfnExtensionMap` is NULL
  - `nDialStringLength` is 0
  - `nExtensionLength` is 0
  - named event does not exist

**EDX_BADPROD**
  The opened device is on a board that is not enabled with the Syntellect patent license (non-STC board).

**EDX_SYSTEM**
  Indicates one of the following:
  - Error from operating system; use `dx_fileerrno()` to obtain error value.
  - Unable to allocate `nDialStringLength +1` characters.

### Source Code

To view the source code for `li_attendant()`, refer to the `syntellect.c` file in the `samples\syntellect` directory under the Intel® Dialogic® home directory.

### Example

To view the source file for the example, refer to the `attendant.c` file in the `samples\syntellect` directory under the Intel® Dialogic® home directory.
li_attendant( ) — perform the actions of an automated attendant

#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <process.h>
#include <srllib.h>
#include <dxxxlib.h>
#include "syntellect.h"

#define EXTENSION_LENGTH   2

#define EVENT_NAME  "ExitEvent"

// define functions used for the hook
static int att_onhook(int dev);  // optional
static int att_offhook(int dev); // optional
static BOOL att_mapextension(char *, char *); // obligatory !!
static int att_waitforrings(int dev, BOOL *bWaiting); // optional

int main (int argc, char *argv[])
{
    HANDLE hEvent;
    HANDLE hThread[2];
    DX_ATTENDANT Att[2];
    BOOL ret;

    ZeroMemory(&Att, sizeof(Att));

    // initialize structure for two thread
    // for telephony control
    Att[0].nSize = sizeof(DX_ATTENDANT);
    strcpy(Att[0].szDevName,  "dxxxB1C1");
    Att[0].pfDisconnectCall = (PFUNC) att_onhook;
    Att[0].pfAnswerCall = (PFUNC) att_offhook;
    Att[0].pfExtensionMap = (PMAPFUNC) att_mapextension;
    strcpy(Att[0].szEventName, EVENT_NAME);
    Att[0].nExtensionLength = EXTENSION_LENGTH;
    Att[0].nDialStringLength = EXTENSION_LENGTH+10;
    Att[0].nTimeOut = 5;

    // create the named event
    if ((hEvent = CreateEvent(          
           NULL,       // no security attributes
           FALSE,      // not a manual-reset event
           FALSE,      // initial state is not signaled
           EVENT_NAME ) // object name
    ) == (HANDLE) NULL)
        return (-1);

    // thread 2 uses built-in functions
    // for telephony control
    Att[1].nSize = sizeof(DX_ATTENDANT);
    strcpy(Att[1].szDevName,  "dxxxB1C2");
    Att[1].pfDisconnectCall = (PFUNC) NULL;
    Att[1].pfAnswerCall = (PFUNC) NULL;
    Att[1].pfExtensionMap = (PMAPFUNC) att_mapextension;
    Att[1].pfWaitForRings = (PWAITFUNC) NULL;
    strcpy(Att[1].szEventName, EVENT_NAME);
    Att[1].nExtensionLength = EXTENSION_LENGTH;
    Att[1].nDialStringLength = EXTENSION_LENGTH+10;
    Att[1].nTimeOut = 5;
}
perform the actions of an automated attendant — li_attendant()

// start the first attendant thread
if ((hThread[0] = (HANDLE) _beginthread(li_attendant, 0, (void *) &Att[0])) == (HANDLE) -1)
{
    printf("Cannot create thread 1.\n");
    exit(0);
}

// start the second attendant thread
if ((hThread[1] = (HANDLE) _beginthread(li_attendant, 0, (void *) &Att[1])) == (HANDLE) -1)
{
    printf("Cannot create thread 2.\n");
    exit(0);
}

Sleep(30000); // Wait as long as you want to run the application

SetEvent(hEvent); // notify threads to exit

WaitForMultipleObjects(2, hThread, TRUE, INFINITE); // wait until the threads are done

CloseHandle(hEvent);

return(0);
}

int att_onhook(int dev)
{
    printf("ONHOOK\n");
    return(dx_sethook(dev, DX_ONHOOK, EV_SYNC));
}

int att_offhook(int dev)
{
    printf("OFFHOOK\n");
    return(dx_sethook(dev, DX_OFFHOOK, EV_SYNC));
}

int att_waitforrings(int dev, BOOL *bWaiting)
{
    int ret;
    DX_EBLK eblk;

    ret = dx_getevt(dev, &eblk, 0);
    if (ret == 0)
    {
        if (eblk.ev_event == DE_RINGS)
            *bWaiting = FALSE;
    }
    return (0);
}

BOOL att_mapextension(char *szExtension, char *szMappedExtension)
{
    int nExtId;

    // for demo purposes use a dumb translation, increment extension by one...
    nExtId = atoi(szExtension) + 1;

    // prefix with flash hook and pause characters
    sprintf(szMappedExtension, ",,", EXTENSION_LENGTH, nExtId);
    return(TRUE);
}

See Also

- li_islicensed_syntellect()
li_islicenced_syntellect( )

**Name:** BOOL li_islicenced_syntellect(chdev)

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

**Returns:** TRUE if board is enabled with Syntellect license  
FALSE if board is not enabled with Syntellect license

**Includes:** syntellect.h

**Category:** Syntellect License Automated Attendant

**Mode:** synchronous

**Platform:** Springware

---

**Description**

The `li_islicenced_syntellect( )` function verifies Syntellect patent license on board. This function is a convenience function used to determine whether the board is enabled with the Syntellect patent license.

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

**Cautions**

When an internal error occurs, `li_islicenced_syntellect( )` returns FALSE. When FALSE is returned on a board that you are certain is enabled with the Syntellect patent license, use `ATDV_LASTERR( )` to find the reason for the error.

**Errors**

None.

**Source Code**

To view the source code for `li_islicenced_syntellect( )`, refer to the end of the `syntellect.c` file in the `samples\syntellect` directory under the Intel® Dialogic® home directory.

**Example**

To view the source file for the example, refer to the `attendant.c` file in the `samples\syntellect` directory under the Intel® Dialogic® home directory.

**See Also**

- `li_attendant( )`
nr_scroute( )

**Name:** int nr_scroute(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
- dtilib.h (optional)
- msilib.h (optional)
- faxlib.h (optional)
- sctools.h

**Category:** TDM Routing

**Mode:** synchronous

**Platform:** DM3, Springware

---

### 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 `\ Dialogic \ sctools` directory.

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:** Digital Network Interface (DTI), Modular Station Interface (MSI), and fax functionality may be conditionally compiled in or out of the function using the DTISC, MSISC, and FAXSC defines in the makefile provided with the function. For example, to compile in DTI functionality, link with the DTI library. 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.
nr_scroute( ) — make a full or half-duplex connection

<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_LSI – analog network (loop start interface) channel device</td>
</tr>
<tr>
<td></td>
<td>• SC_DTI – digital network interface device</td>
</tr>
<tr>
<td></td>
<td>• SC_MSI – MSI station device</td>
</tr>
<tr>
<td></td>
<td>• SC_FAX – fax channel device</td>
</tr>
<tr>
<td></td>
<td>On DM3 boards, the SC_LSI value is not supported.</td>
</tr>
<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>
<tr>
<td>mode</td>
<td>specifies full or half-duplex connection. This parameter contains one of the following defines from sctools.h to specify full or half duplex:</td>
</tr>
<tr>
<td></td>
<td>• SC_FULLDUP – full-duplex connection (default)</td>
</tr>
<tr>
<td></td>
<td>• SC_HALFDUP – half-duplex connection</td>
</tr>
<tr>
<td></td>
<td>When SC_HALFDUP is specified, the function returns with the second device listening to the TDM bus time slot connected to the first device.</td>
</tr>
</tbody>
</table>

**Cautions**

- The devtype1 and devtype2 parameters must match the types of the device handles in devh1 and devh2.
- If you have not defined DTISC, MSISCI, and FAXSC when compiling the sctools.c file, you cannot use this function to route digital channels or 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 \dialogic\sctools directory.
#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"

#ifdef ( __STDC__ ) || defined( __cplusplus )
#else
#endif

int nr_scroute( int devh1, unsigned short devtype1,
                int devh2, unsigned short devtype2 unsigned char mode )
{

  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_tsarraryp = &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;

#ifdef 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
  }
nr_scroute() — make a full or half-duplex connection

#define 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

default:
    nr_scerror("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_scerror("nr_scroute: %s: dx_listen ERROR: %s\n",
                ATDV_NAMEP(devh2), ATDV_ERRMSGP(devh2));
            return -1;
        }
        if (mode == SC_HALFDUP) {
            return 0;
        }
        if (dx_getxmitslot(devh2, &sc_tsinfo) == -1) {
            nr_scerror("nr_scroute: %s: dx_getxmitslot ERROR: %s\n",
                ATDV_NAMEP(devh2), ATDV_ERRMSGP(devh2));
            return -1;
        }
        break;
    case SC_LSI:
        if (ag_listen(devh2, &sc_tsinfo) == -1) {
            nr_scerror("nr_scroute: %s: ag_listen ERROR: %s\n",
                ATDV_NAMEP(devh2), ATDV_ERRMSGP(devh2));
            return -1;
        }
        if (mode == SC_HALFDUP) {
            return 0;
        }
        if (ag_getxmitslot(devh2, &sc_tsinfo) == -1) {
            nr_scerror("nr_scroute: %s: ag_getxmitslot ERROR: %s\n",
                ATDV_NAMEP(devh2), ATDV_ERRMSGP(devh2));
            return -1;
        }
        break;
#endif

#define DTISC

case SC_DTI:
    if (dt_listen(devh2, &sc_tsinfo) == -1) {
        nr_scerror("nr_scroute: %s: dt_listen ERROR: %s\n",
            ATDV_NAMEP(devh2), ATDV_ERRMSGP(devh2));
        return -1;
    }
    if (mode == SC_HALFDUP) {
        return 0;
    }
if (dt_getxmitslot(devh2, &sc_tsinfo) == -1) {  
    nr_scerror("nr_scroute: %s: dt_getxmitslot ERROR: %s\n",  
                 ATDV_NAMEP(devh2), ATDV_ERRMSGP(devh2));  
    return -1;  
}
break;
#endif
#ifdef FAXSC
#else
    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_HALF DUP) {
            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;
#endif
    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
    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
}
nr_scroute( ) — make a full or half-duplex connection

#ifdef FAXSC
  case SC_FAX:
    if (fx_listened(devh1, &sc_tsinfo) == -1) {
      nr_scerror("nr_scroute: %s: fx_listen ERROR: %s",
                 ATDV_NAMEP(devh1), ATDV_ERRMSGP(devh1));
      return -1;
    }
    break;
#endif
return 0;

static void nr_scerror(va_list)
  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
}

— 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
- srlib.h
dxxlib.h
dtlib.h (optional)
msilib.h (optional)
faxlib.h (optional)
sctools.h

**Category:** TDM Routing

**Mode:** synchronous

**Platform:** DM3, Springware

---

**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 \dialogic\sctools directory.

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:** Digital Network Interface (DTI), Modular Station Interface (MSI), and fax functionality may be conditionally compiled in or out of the function using the DTISC, MSISC, and FAXSC defines in the makefile provided with the function. For example, to compile in DTI functionality, link with the DTI library. 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.
nr_scunroute( ) — break a full or half-duplex connection

<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_LSI – analog (loop start interface) channel device</td>
</tr>
<tr>
<td></td>
<td>• SC_DTI – digital network interface device</td>
</tr>
<tr>
<td></td>
<td>• SC_MSI – MSI station device</td>
</tr>
<tr>
<td></td>
<td>• SC_FAX – fax channel device</td>
</tr>
<tr>
<td></td>
<td>On DM3 boards, the SC_LSI value is not supported.</td>
</tr>
<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>
<tr>
<td>mode</td>
<td>specifies full or half-duplex connection. This parameter contains one of the following defines from sctools.h to specify full or half duplex:</td>
</tr>
<tr>
<td></td>
<td>• SC_FULLDUP – full-duplex connection (default)</td>
</tr>
<tr>
<td></td>
<td>• SC_HALFDUP – half-duplex connection</td>
</tr>
<tr>
<td></td>
<td>When SC_HALFDUP is specified, the function returns with the second device listening to the TDM bus time slot connected to the first device.</td>
</tr>
</tbody>
</table>

### Cautions

- The devtype1 and devtype2 parameters must match the types of the device handles in devh1 and devh2.
- If you have not defined DTISC, MSISCI, and FAXSC when compiling the sctools.c file, you cannot use this function to route digital channels or 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 \dialogic\sctools directory.
#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_scunroute( int devh1, unsigned short devtype1,
int devh2, unsigned short devtype2 unsigned char mode )
#else
int nr_scunroute( devh1, devtype1, devh2, devtype2, mode )
#endif

{ /*
* 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;

#endif 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 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;
}
nr_scunroute() — break a full or half-duplex connection

default:
   nr_serror("nr_scunroute: %s: ERROR: Invalid 2nd device type
            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_serror("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_serror("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_serror("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_serror("nr_scunroute: %s: fx_unlisten ERROR: %s
                  ATDV_NAMEP(devh1),ATDV_ERRMSGP(devh1));
         return -1;
      }
      break;
#endif
   default:
      nr_serror("nr_scunroute: %s: ERROR: Invalid 1st device type
            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
}

See Also

- nr_scroute( )
**r2_creatfsig( )**

**Name:** int r2_creatfsig(chdev, forwardsig)

**Inputs:**
- int chdev • channel device handle
- int forwardsig • group I/II forward signal

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

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

**Category:** R2/MF Convenience

**Mode:** synchronous

**Platform:** Springware

### Description

The `r2_creatfsig( )` function is a convenience function that defines and enables leading edge detection of an R2/MF forward signal on a channel. This function calls the `dx_blddt( )` function to create the template.

User-defined tone IDs 101 through 115 are used by this function.

**Note:** R2/MF signaling is typically accomplished through the Global Call API. For more information, see the Global Call documentation set. The R2/MF function described here is provided for backward compatibility only and should not be used for R2/MF signaling.

**Parameter** | **Description**
--- | ---
chdev | specifies the valid channel device handle obtained when the channel was opened using `dx_open( )`
forwardsig | specifies the name of a Group I or Group II forward signal which provides the tone ID for detection of the associated R2/MF tone (or tones). Set to R2_ALLSIG to enable detection of all 15 tones or set to one of the following defines:

<table>
<thead>
<tr>
<th>Group I Defines</th>
<th>Group II Defines</th>
<th>Associated Tone ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGL_1</td>
<td>SIGII_1</td>
<td>101</td>
</tr>
<tr>
<td>SIGL_2</td>
<td>SIGII_2</td>
<td>102</td>
</tr>
<tr>
<td>SIGL_3</td>
<td>SIGII_3</td>
<td>103</td>
</tr>
<tr>
<td>SIGL_4</td>
<td>SIGII_4</td>
<td>104</td>
</tr>
<tr>
<td>SIGL_5</td>
<td>SIGII_5</td>
<td>105</td>
</tr>
<tr>
<td>SIGL_6</td>
<td>SIGII_6</td>
<td>106</td>
</tr>
<tr>
<td>SIGL_7</td>
<td>SIGII_7</td>
<td>107</td>
</tr>
</tbody>
</table>
create R2/MF forward signal tone — r2_creatfsig()

SIGI_8  SIGII_8  108
SIGI_9  SIGII_9  109
SIGI_10 SIGII_10 110
SIGI_11 SIGII_11 111
SIGI_12 SIGII_12 112
SIGI_13 SIGII_13 113
SIGI_14 SIGII_14 114
SIGI_15 SIGII_15 115

Note: Either the Group I or the Group II define can be used to specify the forward signal, because the Group I and Group II defines correspond to the same set of 15 forward signals, and the same user-defined tones are used for Group I and Group II.

Cautions

- The channel must be idle when calling this function.
- Prior to creating the R2/MF tones on a channel, you should delete any previously created user-defined tones (including non-R2/MF tones) to avoid getting an error for having too many tones enabled on a channel.
- This function creates R2/MF tones with user-defined tone IDs from 101 to 115, and you should reserve these tone IDs for R2/MF. If you attempt to create a forward signal tone with this function and you previously created a tone with the same tone ID, an invalid tone ID error will occur.
- The maximum number of user-defined tones is on a per-board basis.

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_INVSUBCMD
Invalid sub-command
**r2_creatfsig() — create R2/MF forward signal tone**

EDX_MAXTMLPT  
Maximum number of user-defined tones for the board

EDX_SYSTEM  
Error from operating system; use `dx_fileerrno()` to obtain error value

EDX_TONEID  
Invalid tone template ID

### Example

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

main()
{
    int dxxxdev;
    /* Open the Voice Channel Device and Enable a Handler */
    /*
    if ( ( dxxxdev = dx_open( "dxxxB1C1", NULL) ) == -1 ) {
        perror( "dxxxB1C1" );
        exit( 1 );
    }
    */
    /* * Create all forward signals */
    /*
    if ( r2_creatfsig( dxxxdev, R2_ALLFSIG ) == -1 ) {
        printf( "Unable to Create the Forward Signals\n" );
        printf( "Lasterror = %d  Err Msg = %s\n",
            ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
        dx_close( dxxxdev );
        exit( 1 );
    }
    */
    /* * Continue Processing */
    /* * * * * */
    /* */
    /* * Close the opened Voice Channel Device */
    /*
    if ( dx_close( dxxxdev ) != 0 ) {
        perror( "close" );
    }
    */
    /* Terminate the Program */
    exit( 0 );
}
```

### See Also
- `r2_playbsig()`
- `dx_addtone()`
- `dx_blddt()`
create R2/MF forward signal tone — r2_creatfsig()
r2_playbsig( )

**Name:** int r2_playbsig(chdev, backwardsig, forwardsig, mode)

**Inputs:**
- int chdev • channel device handle
- int backwardsig • group A/B backward signal
- int forwardsig • group I/II forward signal
- int mode • asynchronous/synchronous

**Returns:**
- 0 if success
- error return code

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

**Category:** R2/MF Convenience

**Mode:** asynchronous or synchronous

**Platform:** Springware

**Description**

The `r2_playbsig()` function is a convenience function that plays a tone and controls the timing sequence required by the R2/MF compelled signaling procedure.

**Note:** R2/MF signaling is typically accomplished through the Global Call API. For more information, see the Global Call documentation set. The R2/MF function described here is provided for backward compatibility only and should not be used for R2/MF signaling.

This function plays a specified backward R2/MF signal on the specified channel until a tone-off event is detected for the specified forward signal.

Compelled signaling sends each signal until it is responded to by a return signal, which in turn is sent until responded to by the other party. See the Voice API Programming Guide for more information about R2/MF compelled signaling.

This function calls the `dx_playtone()` function to play the 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 <code>dx_open()</code></td>
</tr>
<tr>
<td>backwardsig</td>
<td>specifies the name of a Group A or Group B backward signal to play</td>
</tr>
</tbody>
</table>

Specify one of the defines in Group A or one of the defines in Group B:

<table>
<thead>
<tr>
<th>Group A Defines</th>
<th>Group B Defines</th>
<th>Associated Tone ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGA_1</td>
<td>SIGB_1</td>
<td>101</td>
</tr>
<tr>
<td>SIGA_2</td>
<td>SIGB_2</td>
<td>102</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>---</td>
</tr>
<tr>
<td>SIGA_3</td>
<td>SIGB_3</td>
<td>103</td>
</tr>
<tr>
<td>SIGA_4</td>
<td>SIGB_4</td>
<td>104</td>
</tr>
<tr>
<td>SIGA_5</td>
<td>SIGB_5</td>
<td>105</td>
</tr>
<tr>
<td>SIGA_6</td>
<td>SIGB_6</td>
<td>106</td>
</tr>
<tr>
<td>SIGA_7</td>
<td>SIGB_7</td>
<td>107</td>
</tr>
<tr>
<td>SIGA_8</td>
<td>SIGB_8</td>
<td>108</td>
</tr>
<tr>
<td>SIGA_9</td>
<td>SIGB_9</td>
<td>109</td>
</tr>
<tr>
<td>SIGA_10</td>
<td>SIGB_10</td>
<td>110</td>
</tr>
<tr>
<td>SIGA_11</td>
<td>SIGB_11</td>
<td>111</td>
</tr>
<tr>
<td>SIGA_12</td>
<td>SIGB_12</td>
<td>112</td>
</tr>
<tr>
<td>SIGA_13</td>
<td>SIGB_13</td>
<td>113</td>
</tr>
<tr>
<td>SIGA_14</td>
<td>SIGB_14</td>
<td>114</td>
</tr>
<tr>
<td>SIGA_15</td>
<td>SIGB_15</td>
<td>115</td>
</tr>
</tbody>
</table>

**Cautions**

The channel must be idle when calling this function.
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; use `dx_fileerrno( )` to obtain error value

Example

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

main()
{
    int dxxxdev;

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

    /*
     * Create all forward signals
     */
    if ( r2_createfsig( dxxxdev, R2_ALLFSIG ) == -1 ) {
        printf( "Unable to Create the Forward Signals\n" );
        printf( "Lasterror = %d  Err Msg = %s\n", ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
        dx_close( dxxxdev );
        exit( 1 );
    }
}
```
play R2/MF backward signal tone — r2_playbsig()

```c
/*
 * Continue Processing
 * .
 * .
 * .
 * Detect an incoming call using dx_wtring()
 * .
 * Enable the detection of all forward signals using
dx_enbtone(). In this example, only the first
* forward signal will be enabled.
*/
if (dx_enbtone( dxxxdev, SIGI_1, DM_TONEON | DM_TONEOFF ) == -1 ) {
    printf( "Unable to Enable Detection of Tone %d
n", SIGI_1 );
    printf( "Lasterror = %d  Err Msg = %s
",
            ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
    dx_close( dxxxdev );
exi( 1 );
}

/*
 * Now wait for the TDX_CST event and event type,
* DE_TONEON. The data part contains the Tond of
* the forward signal detected. Based on the forward
* signal, determine the backward signal to generate.
* .
* In this example, we will be generating the Group A
* backward signal A-1 (send next digit) assuming
* forward signal received is SIGI_1.
*/
if ( r2_playbsig( dxxxdev, SIGA_1, SIGI_1, EV_SYNC ) == -1 ) {
    printf( "Unable to generate the backward signals\n" );
    printf( "Lasterror = %d  Err Msg = %s
",
            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**
  - r2_creatfsig()
  - dx_blddt()
  - dx_playtone()
  - R2/MF Signaling in *Voice API Programming Guide*
r2_playbsig() — play R2/MF backward signal tone
This chapter provides information on events that may be returned by the voice software. The following topics are discussed:

- **Overview of Events**: 471
- **Termination Events**: 471
- **Unsolicited Events**: 473
- **Call Status Transition (CST) Events**: 473

### 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_CACHEPROMPT
Termination event. Indicates that downloading a cached prompt using dx_cacheprompt() completed.

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 473 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().

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.
Events

TDX_RECORD
Termination event. Returned by record functions such as `dx_rec()` to indicate completion of record.

TDX_RXDATA
Termination event. Returned by `dx_RxIottData()` and `dx_TxRxIottData()` to indicate completion of ADSI two-way FSK data reception.

TDX_SETHOOK
Termination event. Returned by `dx_sethook()` to indicate completion of this function in asynchronous mode. The cst_event field in the DX_CST data structure or the ev_event field in the DX_EBLK data structure indicates whether the hook switch state has been set to on or off.

TDX_TXDATA
Termination event. Returned by `dx_TxIottData()` and `dx_TxRxIottData()` to indicate completion of ADSI two-way FSK data transmission.

TDX_WINK
Termination event. Returned by `dx_wink()` to indicate completion of this function in asynchronous mode.

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 473.

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 506. 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.

### Call Status Transition Events on DM3 Boards

On DM3 boards, 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.

- **DE_SILOFF**
  - Call status transition event. Indicates silence detected on the channel.

- **DE_STOPGETEVT**
  - Call status transition event. Indicates that the `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.

### Call Status Transition Events on Springware Boards

On Springware boards, 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_LCOFF**
  - Call status transition event. Indicates loop current off.
Events

DE_LCON
  Call status transition event. Indicates loop current on.

DE_LCREV
  Call status transition event. Indicates loop current reversal.

DE_RINGS
  Call status transition event. Indicates rings received.

DE_RNGOFF
  Call status transition event. Specifies ring off event.

DE_SILOFF
  Call status transition event. Indicates non-silence detected on the channel.

DE_SIOLON
  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_STOPWTRING
  Call status transition event. Indicates that the \texttt{dx\_wtring()} function which was in progress has been stopped.

DE_STOPRINGS
  Call status transition event.

DE_TONEOFF
  Call status transition event. Indicates tone off event received.

DE_TONEON
  Call status transition event. Indicates tone on event received.

DE_WINK
  Call status transition event. Indicates wink received.

DX_OFFHOOK
  Call status transition event. Indicates off-hook status.

DX_ONHOOK
  Call status transition event. Indicates on-hook status.
This chapter provides an alphabetical reference to the data structures used by voice library functions. The following data structures are discussed:

- **ADSI_XFERSTRUC** ................................................................. 478
- **CT_DEVINFO** ................................................................. 479
- **DV_DIGIT** ................................................................. 483
- **DV_TPT** ................................................................. 485
- **DX_ATTENDANT** ................................................................. 493
- **DX_CAP** ................................................................. 496
- **DX_CST** ................................................................. 504
- **DX_EBLK** ................................................................. 506
- **DX_ECRCT** ................................................................. 508
- **DX_IOTT** ................................................................. 509
- **DX_SVCB** ................................................................. 514
- **DX_SVMT** ................................................................. 517
- **DX_STREAMSTAT** ................................................................. 512
- **DX_UIO** ................................................................. 519
- **DX_XPB** ................................................................. 520
- **FEATURE_TABLE** ................................................................. 525
- **SC_TSINFO** ................................................................. 529
- **TN_GEN** ................................................................. 530
- **TN_GENCAD** ................................................................. 531
- **TONE_DATA** ................................................................. 533
**ADSI_XFERSTRUC**

```c
typedef struct_ADSI_XFERSTRUC {
    UINT    cbSize;
    DWORD   dwTxDataMode;
    DWORD   dwRxDataMode;
} ADSI_XFERSTRUC;
```

### Description

The ADSI_XFERSTRUC data structure stores information for the reception and transmission of Analog Display Services Interface (ADSI) 2-way frequency shift keying (FSK) data. This structure is used by the `dx_RxIottData()`, `dx_TxIottData()`, and `dx_TxRxIottData()` functions.

This structure is defined in `dxxxlib.h`.

### Field Descriptions

The fields of the ADSI_XFERSTRUC data structure are described as follows:

- **cbSize**
  - Specifies the size of the structure, in bytes.

- **dwTxDataMode**
  - Specifies one of the following data transmission modes:
    - `ADSI_ALERT` – for FSK with Alert (CAS)
    - `ADSI_NOALERT` – for FSK without Alert (CAS)
    - `ADSI_ONHOOK_SEIZURE` – for on-hook with seizure
    - `ADSI_ONHOOK_NOSEIZURE` – for on-hook without seizure

- **dwRxDataMode**
  - Specifies one of the following data reception modes:
    - `ADSI_ALERT` – for FSK with Alert (CAS)
    - `ADSI_NOALERT` – for FSK without Alert (CAS)
    - `ADSI_ONHOOK_SEIZURE` – for on-hook with seizure
    - `ADSI_ONHOOK_NOSEIZURE` – for on-hook without seizure

### Example

For an example of how to use this data structure, see the Example section for `dx_RxIottData()`, `dx_TxIottData()`, or `dx_TxRxIottData()` in Chapter 2, “Function Information”.

CT_DEVINFO

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. This structure is used by
time division multiplexing (TDM) bus routing functions identified by the suffix _getxmitslot(). On
return from the 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 dxxxlib.h.

■ Field Descriptions

DM3 Boards

On DM3 boards, the fields of the CT_DEVINFO data structure are described as follows:

c_t_prodid
    Contains a valid product identification number for the device [length: 4 (unsigned long)].

c_t_devfamily
    Specifies the device family [length: 1 (unsigned char)]. Possible values are:
    • CT_DFM3 – DM3 device
    • CT_DFHMPDM3 – HMP device (Host Media Processing)

c_t_devmode
    Specifies the device mode [length: 1 (unsigned char)] that is valid only for a D/xx or VFX/xx
    board. Possible values are:
    • CT_DMRESOURCE – DM3 voice device in flexible routing configuration
    • CT_DMNETWORK – DM3 network device or DM3 voice device in fixed routing
      configuration
    For information on flexible routing and fixed routing, see the Voice API Programming Guide.

c_t_nettype
    Specifies the type of network interface for the device [length: 1 (unsigned char)]. Possible
    values are:
CT_DEVINFO — channel/time slot device information

- CT_IPT – IP connectivity
- CT_NTANALOG – analog interface. Analog and voice devices on board are handling call processing
- CT_NTT1 – T-1 digital network interface
- CT_NTE1 – E-1 digital network interface
- CT_NTMSI – MSI/SC station interface
- CT_NTHIZ – high impedance (HiZ) interface. This value is bitwise-ORed with the type of network interface. A digital HiZ T-1 board would return CT_NTHIZ | CT_NTT1. A digital HiZ E-1 board would return CT_NTHIZ | CT_NTE1. An analog HiZ board would return CT_NTHIZ | CT_NTTXZSWITCHABLE | CT_NTANALOG.
- CT_NTTXZSWITCHABLE – The network interface can be switched to the transmit impedance state. This value is bitwise-ORed with the type of network interface. An analog HiZ board would return CT_NTHIZ | CT_NTTXZSWITCHABLE | CT_NTANALOG. This is used to transmit the record notification beep tone.

ct_busmode

Specifies the bus architecture used to communicate with other devices in the system [length: 1 (unsigned char)]. Possible values are:
- CT_BMSCBUS – TDM bus architecture
- CT_H100 – H.100 bus
- CT_H110 – H.110 bus

cctusbusencoding

Describes the PCM encoding used on the bus [length: 1 (unsigned char)]. Possible values are:
- CT_BEULAW – mu-law encoding
- CT_BEALAW – A-law encoding
- CT_BELLAW – linear encoding
- CT_BEBYPASS – encoding is being bypassed

ct_rfu

Returned by ms_getctinfo() for DM3 MSI devices. This field returns a character string containing the board and channel of the voice channel resource associated with the station interface. This data is returned in BxxCy format, where xx is the voice board and y is the voice channel. For example, dxxxB1C1 would be returned as B1C1. To subsequently use this information in a dx_open() function, you must add the dxxx prefix to the returned character string.

c_ext_devinfo.ct_net_devinfo.ct_prottype

Contains information about the protocol used on the specified digital network interface device. Possible values are:
- CT_CAS – channel associated signaling
- CT_CLEAR – clear channel signaling
- CT_ISDN – ISDN
- CT_R2MF – R2MF

On Intel® NetStructure® IPT Series boards, the ct_devfamily field is described as follows:

ct_devfamily

Specifies the device family [length: 1 (unsigned char)]. Possible values are:
- CT_NETSTRUCTIP – IPT series board
**Springware Boards**

On Springware boards, the fields of the CT_DEVINFO data structure are described as follows:

- **ct_prodid**
  Contains a valid product identification number for the device [length: 4 (unsigned long)].

- **ct_devfamily**
  Specifies the device family [length: 1 (unsigned char)]. Possible values are:
  - CT_DFD41D – D/41D board family
  - CT_DFD41E – analog or voice channel of a D/xx or VFX/xx board such as D/41ESC or VFX/40ESC
  - CT_DFSPAN – analog channel such as of a D/160SC-LS board; a voice channel such as of a D/240SC, D/320SC, D/240SC-T1, D/300SC-E1 or D/160SC-LS board; or a digital channel such as of a D/240SC-T1 or D/300SC-E1 board
  - CT_DFMSI – a station on an MSI board
  - CT_DFSCX – SCX160 SCXbus adapter family

- **ct_devmode**
  Specifies the device mode field [length: 1 (unsigned char)] that is valid only for a D/xx or VFX/xx board. Possible values are:
  - CT_DMRESOURCE – analog channel not in use
  - CT_DMNETWORK – analog channel available to process calls from the telephone network

- **ct_nettype**
  Specifies the type of network interface for the device [length: 1 (unsigned char)]. Possible values are:
  - CT_NTNONE – D/xx or VFX/xx board configured as a resource device; voice channels are available for call processing; analog channels are disabled.
  - CT_NTANALOG – analog and voice devices on board are handling call processing
  - CT_NTT1 – T-1 digital network interface
  - CT_NTE1 – E-1 digital network interface
  - CT_NTMSI – MSI/SC station interface

*Note:* The `dx_getctinfo()` function does not return a value of ct_nettype = CT_NTNONE when a D/41ESC or D/41E-PCI board is configured as a resource device. Use `ct_devmode` returned from `dx_getctinfo()` to determine the resource mode of the product. If D41ESCRESOURCE is set to ON in the Intel Dialogic Configuration Manager, `ct_devmode` = CT_DMRESOURCE. If D41ESCRESOURCE is OFF, `ct_devmode` = CT_DMNETWORK.

- **ct_busmode**
  Specifies the bus architecture used to communicate with other devices in the system [length: 1 (unsigned char)]. Possible values are:
  - CT_BMSCBUS – TDM bus architecture

- **ct_busencoding**
  Describes the PCM encoding used on the bus [length: 1 (unsigned char)]. Possible values are:
  - CT_BEULAW – Mu-law encoding
  - CT_BEALAW – A-law encoding

- **ct_rfu**
  Reserved for future use.
ct_ext_devinfo.ct_net_devinfo.ct_prottype
Contains information about the protocol used on the specified digital network interface device.
Possible values are:
- CT_CAS – channel associated signaling
- CT_CLEAR – clear channel signaling
- CT_ISDN – ISDN
- CT_R2MF – R2/MF signaling

Example
For an example of how to use the CT_DEVINFO structure, see the Example section for dx_getctinfo().
**user digit buffer — DV_DIGIT**

**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 a **dx_getdig()** is performed, the digits are collected from the firmware and transferred to the user’s digit buffer. The digits are stored as an array inside the DV_DIGIT structure.

The DG_MAXDIGS define in *dxxxlib.h* indicates the maximum number of digits (31) that can be returned by a single call to **dx_getdig()**.

**Note:** 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()**.

■ **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.

On DM3 boards, use the following defines to identify the digit type:

- **DG_DTMF_ASCII** – DTMF
- **DG_DPD_ASCII** – DPD (dial pulse)
- **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

On Springware boards, use the following defines to identify the digit type:

- **DG_DTMF_ASCII** – DTMF
- **DG_DPD_ASCII** – DPD (dial pulse)
- **DG_MF_ASCII** – MF
- **DG_USER1_ASCII** – GTD user-defined
- **DG_USER2_ASCII** – GTD user-defined
- **DG_USER3_ASCII** – GTD user-defined
- **DG_USER4_ASCII** – GTD user-defined
- **DG_USER5_ASCII** – 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()`.
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`).

*Note:* 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.

On DM3 boards, 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. On DM3 boards, this termination condition is only supported by the `dx_getdig()` function.
- DX_MAXDATA – maximum data for ADSI 2-way FSK. A Transmit/Receive FSK session is terminated when the specified value of FSK DX_MAXDATA (in bytes) is transmitted/received.
- DX_MAXDTMF – maximum number of digits received
DV_TPT — termination parameter table

- DX_MAXNOSIL – maximum length of non-silence. The range is 10 msec to 250 sec (25000 in 10 msec units).
- DX_MAXSIL – maximum length of silence. The range is 10 msec to 250 sec (25000 in 10 msec units).
- DX_MAXTIME – maximum function time. On DM3 boards, 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)

On Springware boards, the supported termination conditions are:
- DX_DIGMASK – digit termination for bit mask of digits received
- DX_DIGTYPE – digit termination for user-defined tone
- DX_IDDTIME – maximum delay between digits
- DX_LCOFF – loop current drop
- DX_MAXDATA – maximum data for ADSI 2-way FSK. A Transmit/Receive FSK session is terminated when the specified value of FSK DX_MAXDATA (in bytes) is transmitted/received.
- DX_MAXDTMF – maximum number of digits received
- DX_MAXNOSIL – maximum length of non-silence
- DX_MAXSIL – maximum length of silence
- DX_MAXTIME – maximum function time
- DX_PMOFF – pattern match of non-silence
- DX_PMON – pattern match of silence
- DX_TONE – tone on or tone off termination for global tone detection (GTD) termination conditions

Note: DX_PMOFF and DX_PMON must be used in tandem. See the Example section for more information.

Note: When using the DX_PMON and DX_PMOFF termination conditions, some of the DV_TPT fields are set differently from other termination conditions.

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.
- size – When using DX_MAXDATA, which specifies maximum data for ADSI 2-way FSK, valid values in tp_length are 1 to 65535.
- number of digits – Applies when using DX_MAXDTMF, which specifies termination after a certain number of digits is received.
termination parameter table — DV_TPT

- 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>
<tr>
<td>c</td>
<td>DM_C</td>
</tr>
<tr>
<td>d</td>
<td>DM_D</td>
</tr>
</tbody>
</table>

- number of pattern repetitions – Applies to DX_PMOFF, which specifies the number of times a pattern should repeat before termination.

**Note:** When DX_PMOFF is the termination condition, tp_length contains the tp_flags information. See the tp_flags description and also the Example section for more information.

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 DM3 boards 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 DM3 boards 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. This bit is not supported on Springware boards. If set, the silence timer starts immediately at the onset of `ec_stream()` or `ec_reciottdata()` 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 DM3 boards 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
  - DX_PMOFF
  - DX_PMON

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_LCOFF</td>
<td>(TF_LEVEL</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_PMON</td>
<td>(TF_EDGE)</td>
</tr>
<tr>
<td>TF_TONE</td>
<td>(TF_LEVEL</td>
</tr>
</tbody>
</table>
**Notes:**

1. The TF_SETINIT termination flag cannot be used with RM_TONE record mode on Springware boards with analog front-ends.

2. DX_PMOFF does not have a default tp_flags value. The tp_flags value for DX_PMOFF is set in tp_length. See the tp_length field description and also the Example section for more information.

3. 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>

   For **DM3 boards**, 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>

   For **Springware boards**, the following table shows whether a termination condition can be level-sensitive or edge-sensitive.

<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_LCOFF</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_PMON/DX_PMOFF</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_PMOFF, tp_data can specify the maximum time of silence off.
DV_TPT — termination parameter table

- If tp_termno contains DX_PMON, tp_data can specify the maximum time of silence on.
- If tp_termno contains DX_TONEOFF, tp_data can specify termination after a tone-off event.
- If tp_termno contains DX_TONEON, tp_data can specify 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 16 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 16. 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_CLREND</td>
<td></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>DX_MAXSIL</td>
<td>IO_LINK</td>
<td>max length silence</td>
<td>bit 0:</td>
<td>TF_EDGE*</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_LEVEL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IO_CONT</td>
<td></td>
<td>bit 2: no clr*</td>
<td>TF_CLREND</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 3: clr hist</td>
<td>TF_CLRBEG</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 4: no-setinit</td>
<td>TF_USE*</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 5: 100 msec*</td>
<td>TF_SETINIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TF_10MS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DX_MAXNOSIL</td>
<td>IO_LINK</td>
<td>max length non-silence</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_CLREND</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IO_CONT</td>
<td></td>
<td>bit 1: no clr*</td>
<td>TF_CLRBEG</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 2: no clr*</td>
<td>TF_USE*</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 3: clr hist</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 4: N/A</td>
<td>TF_10MS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 5: 100 msec*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DX_LCOFF</td>
<td>IO_LINK</td>
<td>max length loop current drop</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_CLREND*</td>
<td></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>bit 4: N/A</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 5: 100 msec*</td>
<td>TF_10MS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DX_IDDTIME</td>
<td>IO_LINK</td>
<td>max length interdigit delay</td>
<td>bit 0: TF_EDGE*</td>
<td>N/A</td>
<td></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>start@1st</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IO_CONT</td>
<td></td>
<td>bit 2: N/A</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 3: N/A</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 4: N/A</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 5: 100 msec*</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 16. DV_TPT Field Settings Summary (Continued)

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>tp_termno</td>
<td>tp_type</td>
<td>tp_length</td>
<td>tp_flags: not set</td>
<td>tp_flags: set</td>
<td>tp_data</td>
<td>tp_nextp</td>
<td></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>
<td></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></td>
<td></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></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 3: N/A</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 4: N/A</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bit 5: 100 msec*</td>
<td>TF_10MS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DX_DIGMASK</td>
<td>IO_LINK</td>
<td>bit 0: d (set)</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>
<td></td>
</tr>
<tr>
<td></td>
<td>IO_EOT</td>
<td>bit 1: 1</td>
<td>_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IO_CONT</td>
<td>bit 2: 2</td>
<td>_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 3: 3</td>
<td>_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 4: 4</td>
<td>_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 5: 5</td>
<td>_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 6: 6</td>
<td>_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 7: 7</td>
<td>_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 8: 8</td>
<td>_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 9: 9</td>
<td>_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 10: 0</td>
<td>_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 11: .</td>
<td>_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 12: #</td>
<td>_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 13: a</td>
<td>_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 14: b</td>
<td>_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 15: c</td>
<td>_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DX_PMOFF</td>
<td>IO_LINK</td>
<td>number of pattern repetitions</td>
<td>minimum time silence off</td>
<td>max time silence off</td>
<td>pointer to next DV_TPT if linked list</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IO_EOT</td>
<td></td>
<td>_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IO_CONT</td>
<td></td>
<td>_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DX_PMON</td>
<td>IO_LINK</td>
<td>bit 0: TF_EDGE/TF_LEVEL</td>
<td>maximum time silence on</td>
<td>max time silence on</td>
<td>pointer to next DV_TPT if linked list</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IO_EOT</td>
<td>bit 1: N/A</td>
<td>_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IO_CONT</td>
<td>bit 2: N/A</td>
<td>_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 3: N/A</td>
<td>_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 4: N/A</td>
<td>_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 5: 100 msec/TF_10MS</td>
<td>_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<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 DX_TONEOFF</td>
<td>pointer to next DV_TPT if linked list</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IO_EOT</td>
<td>bit 1: no clr</td>
<td>_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IO_CONT</td>
<td>bit 2: no clr*</td>
<td>_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 3: clr hist</td>
<td>_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>_</td>
<td>TF_CRLREND*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>_</td>
<td>TF_CLRBEG</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>_</td>
<td>TF_USE*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DX_DIGTYPE</td>
<td>IO_LINK</td>
<td>low byte: ASCII val.</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>
<td></td>
</tr>
<tr>
<td></td>
<td>IO_EOT</td>
<td>_</td>
<td>bit 1: no clr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IO_CONT</td>
<td>_</td>
<td>bit 2: no clr*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>_</td>
<td>bit 3: clr hist</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

This section provides an example of how to use DX_PMOFF and DX_PMON.

The DX_PMOFF and DX_PMON termination conditions must be used in tandem. The
DX_PMON termination condition must directly follow the DX_PMOFF termination condition.
Each condition is specified in a DV_TPT structure. A combination of both DV_TPT structures is
used to form a single termination condition.

In the first block of the example code below, tp_termno is set to DX_PMOFF. The tp_length holds
the number of patterns before termination. tp_flags holds the minimum time for silence off while
tp_data holds the maximum time for silence off. In the next DV_TPT structure, tp_termno is
DX_PMON, and the tp_length field holds the flag bit mask. Only the “units” bit is valid; all other
bits must be 0. The tp_flags field holds the minimum time for silence on, while tp_data holds the
maximum time for silence on.

```
#include <srllib.h>
#include <dxxxlib.h>
DV_TPT   tpt[2];
/*
 * detect a pattern which repeats 4 times of approximately 2 seconds
 * off 2 seconds on.
 */
tpt[0].tp_type = IO_CONT; /* next entry is contiguous */
tpt[0].tp_termno = DX_PMOFF; /* specify pattern match off */
tpt[0].tp_length = 4;       /* terminate if pattern repeats 4 times */
tpt[0].tp_flags = 175;      /* minimum silence off is 1.75 seconds
     * (10 msec units) */
tpt[0].tp_data = 225;       /* maximum silence off is 2.25 seconds
     * (10 msec units) */

tpt[1].tp_type = IO_EOT;    /* This is the last in the chain */
tpt[1].tp_termno = DX_PMON; /* specify pattern match on */
tpt[1].tp_length = TF_10MS; /* use 10 msec timer units */
tpt[1].tp_flags = 175;      /* minimum silence on is 1.75 seconds
     * (10 msec units) */
tpt[1].tp_data = 225;       /* maximum silence on is 2.25 seconds
     * (10 msec units) */
/* issue the function */
```
DX_ATTENDANT

typedef int (*PWAITFUNC)(int dev, BOOL *bWaiting);
typedef int (*PFUNC)(int dev);
typedef BOOL (*PMAPFUNC)(char *, char *);

typedef struct {
    int nSize;
    char szDevName[15];
    PFUNC pfnDisconnectCall;
    PWAITFUNC pfnWaitForRings;
    PFUNC pfnAnswerCall;
    PMAPFUNC pfnExtensionMap;
    char szEventName[MAX_PATH+1];
    int nExtensionLength;
    int nTimeOut; // in seconds
    int nDialStringLength;
} DX_ATTENDANT, *PDX_ATTENDANT;

### Description

The DX_ATTENDANT data structure is not supported on DM3 boards.

The DX_ATTENDANT data structure contains parameters for Syntellect License Automated Attendant.

This structure provides the information necessary for the proper operation and initialization of li_attendant(). This structure is used in a synchronous environment and is defined in syntellect.h located in the \inc directory.

### Field Descriptions

The fields of the DX_ATTENDANT data structure are described as follows:

**nSize**

*Required.* Represents the size of this data structure in bytes. Used for version control.

**SzDevName**

*Required.* Identifies the device name to open on which li_attendant() will run; for example, “dxxxB1C1”.

**pfnDisconnectCall**

*Optional.* Specifies the address of a disconnect function. When NULL, dx_sethook() is called. This field can be used to override default analog front end interface behavior. For example, on a T-1 interface a function that manipulates the A and B bits can be used instead to disconnect a call.

**pfnWaitForRings**

*Optional.* Specifies the address of a “Wait for Rings” function. When NULL, dx_getevt() is called. This field can be used to override default analog front end interface behavior. For example, on a T-1 interface, a function that monitors the A and B bits can be used instead to wait for an incoming call.

**pfnAnswerCall**

*Optional.* Specifies the address of a connect function. When NULL, dx_sethook() is called. This field can be used to override default analog front end interface behavior. For example, on
a T-1 interface a function that manipulates the A and B bits can be used instead to answer a call.

PFNExtensionMap

*Required.* Specifies the address of a function that translates the extension digits as received for the caller to a digit string, representing the physical extension, to actually dial. For example, when a caller enters “0” (usually for operator) the extension for the operator may actually be “1500”.

szEventName

*Required.* Specifies the string name for the event used by the application to notify the *li_attendant()* thread to terminate. An example is “MyEventName”.

nExtensionLength

*Required.* Specifies the maximum number of DTMF digits a caller can enter in response to the prompt asking for an extension.

nTimeOut

*Required.* Specifies the amount of time, in seconds, before *dx_getdig()* returns and times out when waiting for caller input.

nDialStringLength

*Required.* Specifies the length in bytes of the maximum translated extension dial string. For example, for “1500” this field would be 4.

**Example**

For an example of DX_ATTENDANT, see the Example section for *li_attendant()*.
DX_CAP — call progress analysis parameters

typedef struct DX_CAP {
  unsigned short ca_nbrdna; /* # of rings before no answer. */
  unsigned short ca_stdelay; /* 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; /* % acceptable pos. dev of short low sig. */
  unsigned short ca_l0t01b; /* % acceptable neg. dev of short low sig. */
  unsigned short ca_l0t0tola; /* % acceptable pos. dev of long low sig. */
  unsigned short ca_l0t0tolb; /* % acceptable neg. dev of long low sig. */
  unsigned short ca_hiltola; /* % acceptable pos. dev of high signal. */
  unsigned short ca_hiltolb; /* % acceptable neg. dev of high signal. */
  unsigned short ca_l01bmax; /* Maximum interval for short low for busy. */
  unsigned short ca_l02bmax; /* Maximum interval for long low for busy. */
  unsigned short ca_hilbmax; /* Maximum interval for 1st high for busy */
  unsigned short ca_nbnsbusy; /* Num. of highs after nbdna busy check. */
  unsigned short ca_logltch; /* Silence deglitch duration. */
  unsigned short ca_higltch; /* Non-silence deglitch duration. */
  unsigned short ca_lonrmax; /* Max. short low dur. of double ring. */
  unsigned short ca_l02rmin; /* Min. long low dur. of double ring. */
  unsigned short ca_intflg; /* Operator intercept mode. */
  unsigned short ca_intfltr; /* Minimum signal to qualify freq. detect. */
  unsigned short rfu1; /* reserved for future use */
  unsigned short rfu2; /* reserved for future use */
  unsigned short rfu3; /* reserved for future use */
  unsigned short rfu4; /* reserved for future use */
  unsigned short ca_hisiz; /* Used to determine which lowmax to use. */
  unsigned short ca_alowmax; /* Max. low before con. if high hisize. */
  unsigned short ca_blowmax; /* Max. low before con. if high hisize. */
  unsigned short ca_nbvorbg; /* Number of rings before analysis begins. */
  unsigned short ca_lolcei; /* Maximum 2nd high dur. for a retrain. */
  unsigned short ca_lolcei1; /* 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_tmefrq; /* Total duration of good signal required. */
  unsigned short ca_rejctfrq; /* Allowable % of bad signal. */
  unsigned short ca_mmaxnrs; /* Maximum duration of answer. */
  unsigned short ca_smnrdgl; /* Silence deglitching value for answer. */
  unsigned short ca_mxtimemrq; /* Max time for 1st freq to remain in bounds */
  unsigned short ca_lower2frq; /* lower bound for second frequency */
  unsigned short ca_upper2frq; /* upper bound for second frequency */
  unsigned short ca_tme2frq; /* min time for 2nd freq to remain in bounds */
  unsigned short ca_mxtimemrfq; /* max time for 2nd freq to remain in bounds */
  unsigned short ca_lower3frq; /* lower bound for third frequency */
  unsigned short ca_upper3frq; /* upper bound for third frequency */
  unsigned short ca_tme3frq; /* min time for 3rd freq to remain in bounds */
  unsigned short ca_mxtimem3rfq; /* max time for 3rd freq to remain in bounds */
  unsigned short ca_dtn_pres; /* Length of a valid dial tone (def=1sec) */
  unsigned short ca_dtn_npres; /* Max time to wait for dial tone (def=3sec) */
  unsigned short ca_dtn_deboff; /* The dialtone off debouncer (def=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;
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

DM3 Boards

On DM3 boards, the following fields of the DX_CAP data structure are supported (DM3 boards use PerfectCall call progress analysis):

- `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_PVDDIS – Enable PVD.
    This setting provides PVD call analysis only (no call progress).
  
  - DX_PVDOPTDIS – Enable PVD and DX_OPTNOCON.
    This setting provides call progress with SIT frequency detection and PVD call analysis.
  
  - DX_PAMDDISABLE – Enable PAMD and PVD.
    This setting provides PAMD and PVD call analysis only (no call progress).
  
  - DX_PAMDOPTE – Enable PAMD, PVD, and DX_OPTNOCON.
    This setting provides full call progress and call analysis.

  Length: 1   Default: DX_OPTNOCON
**DX_CAP — call progress analysis parameters**

**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

**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

---

**Springware Boards**

On Springware boards, the fields of the DX_CAP data structure are described as follows:

*Note:* A distinction is made in the following descriptions between support for PerfectCall call progress analysis (PerfectCall CPA only), basic call progress analysis (Basic CPA only), and call progress analysis (CPA).

**ca_nbrdna**
Number of Rings before Detecting No Answer. The number of single or double rings to wait before returning a “no answer” (Basic CPA only)
Length: 1   Default: 4   Units: rings

**ca_stdely**
Start Delay. The delay after dialing has been completed and before starting analysis for cadence detection, frequency detection, and positive voice detection (CPA)
Length: 2   Default: 25   Units: 10 msec

**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. (CPA)
Length: 2   Default: 4000   Units: 10 msec

**ca_lcdly**
Loop Current Delay. The delay after dialing has been completed and before beginning loop current detection. (CPA) The value -1 means disable loop current detection.
Length: 2   Default: 400   Units: 10 msec

**ca_lcdly1**
Loop Current Delay 1. The delay after loop current detection detects a transient drop in loop current and before call analysis returns a “connect” to the application (CPA)
Length: 2   Default: 10   Units: 10 msec
call progress analysis parameters — DX_CAP

ca_hedge
Hello Edge. The point at which a “connect” will be returned to the application (CPA)
- 1 – Rising Edge (immediately when a connect is detected)
- 2 – Falling Edge (after the end of the salutation)
Length: 1 Default: 2

csa cnosil
Continuous Non-silence. The maximum length of the first or second period of non-silence allowed. If exceeded, a “no ringback” is returned. (CPA)
Length: 2. Default: 650 Units: 10 msec

csa lo1tola
Low 1 Tolerance Above. Percent acceptable positive deviation of short low signal (Basic CPA only)
Length: 1 Default: 13 Units:%

csa lo1tolb
Low 1 Tolerance Below. Percent acceptable negative deviation of short low signal (Basic CPA only)
Length: 1 Default: 13 Units:%

csa lo2tola
Low 2 Tolerance Above. Percent acceptable positive deviation of long low signal (Basic CPA only)
Length: 1 Default: 13 Units:%

csa lo2tolb
Low 2 Tolerance Below. Percent acceptable negative deviation of long low signal (Basic CPA only)
Length: 1 Default: 13 Units:%

csa hi1tola
High 1 Tolerance Above. Percent acceptable positive deviation of high signal (Basic CPA only)
Length: 1 Default: 13 Units:%

csa hi1tolb
High 1 Tolerance Below. Percent acceptable negative deviation of high signal (Basic CPA only)
Length: 1 Default: 13 Units:%

csa lo1bmax
Low 1 Busy Maximum. Maximum interval for short low for busy (Basic CPA only)
Length: 2 Default: 90 Units: 10 msec

csa lo2bmax
Low 2 Busy Maximum. Maximum interval for long low for busy (Basic CPA only)
Length: 2 Default: 90 Units: 10 msec

csa hi1bmax
High 1 Busy Maximum. Maximum interval for first high for busy (Basic CPA only)
Length: 2 Default: 90 Units: 10 msec
**DX_CAP — call progress analysis parameters**

ca_nsbusy
Non-silence Busy. The number of non-silence periods in addition to nbrdna to wait before returning a “busy” (Basic CPA only)
Length: 1 Default: 0 Negative values are valid

ca_logltch
Low Glitch. The maximum silence period to ignore. Used to help eliminate spurious silence intervals. (CPA)
Length: 2 Default: 15 Units: 10 msec

ca_higltch
High Glitch. The maximum nonsilence period to ignore. Used to help eliminate spurious nonsilence intervals. (CPA)
Length: 2 Default: 19 Units: 10 msec

ca_lo1rmax
Low 1 Ring Maximum. Maximum short low duration of double ring (Basic CPA only)
Length: 2 Default: 90 Units: 10 msec

ca_lo2rmin
Low 2 Ring Minimum. Minimum long low duration of double ring (Basic CPA only)
Length: 2 Default: 225 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 (CPA)
- DX_OPTDIS – Disable SIT frequency detection, PAMD, and PVD.
- DX_OPTNOCON – Enable SIT frequency detection and return an “intercept” immediately after detecting a valid frequency.
- DX_PVDENABLE – Enable PVD.
- DX_PVDOPTNOCON – Enable PVD and DX_OPTNOCON.
- DX_PAMDENABLE – Enable PAMD and PVD.
- DX_PAMDOPTEN – Enable PAMD, PVD, and DX_OPTNOCON.

*Note:* DX_OPTEN and DX_PVDOPTEN are obsolete. Use DX_OPTNOCON and DX_PVDOPTNOCON instead.
Length: 1 Default: DX_OPTNOCON

ca_intfltr
Not used

ca_hisiz
High Size. Used to determine whether to use alowmax or blowmax (Basic CPA only)
Length: 2 Default: 90 Units: 10 msec

ca_alowmax
A Low Maximum. Maximum low before connect if high > hisiz (Basic CPA only)
Length: 2 Default: 700 Units: 10 msec

ca_blowmax
B Low Maximum. Maximum low before connect if high < hisiz (Basic CPA only)
Length: 2 Default: 530 Units: 10 msec
call progress analysis parameters — DX_CAP

cal_nbrbeg
  Number Before Beginning. Number of non-silence periods before analysis begins (Basic CPA only)
  Length: 1  Default: 1  Units: rings

cal_hi1ceil
  High 1 Ceiling. Maximum 2nd high duration for a retrain (Basic CPA only)
  Length: 2  Default: 78  Units: 10 msec

cal_lo1ceil
  Low 1 Ceiling. Maximum 1st low duration for a retrain (Basic CPA only)
  Length: 2  Default: 58  Units: 10 msec

cal_lowerfrq
  Lower Frequency. Lower bound for 1st tone in an SIT (CPA)
  Length: 2  Default: 900  Units: Hz

cal_upperfrq
  Upper Frequency. Upper bound for 1st tone in an SIT (CPA)
  Length: 2  Default: 1000  Units: Hz

cal_timefrq
  Time Frequency. Minimum time for 1st tone in an SIT to remain in bounds. The minimum amount of time required for the audio signal to remain within the frequency detection range specified by upperfrq and lowerfrq for it to be considered valid. (CPA)
  Length: 1  Default: 5  Units: 10 msec

cal_rejctfrq
  Not used

cal_maxansr
  Maximum Answer. The maximum allowable length of ansrsize. When ansrsize exceeds maxansr, a “connect” is returned to the application. (CPA)
  Length: 2  Default: 1000  Units: 10 msec

cal_ansrdgl
  Answer Deglitcher. The maximum silence period allowed between words in a salutation. This parameter should be enabled only when you are interested in measuring the length of the salutation. (CPA)
  • -1 – Disable this condition
  Length: 2  Default: -1  Units: 10 msec

cal_mxtimefrq
  Maximum Time Frequency. Maximum allowable time for 1st tone in an SIT to be present
  Default: 0  Units: 10 msec

cal_lower2frq
  Lower Bound for 2nd Frequency. Lower bound for 2nd tone in an SIT
  Default: 0  Units: Hz

cal_upper2frq
  Upper Bound for 2nd Frequency. Upper bound for 2nd tone in an SIT
  Default: 0  Units: Hz
DX_CAP — call progress analysis parameters

ca_time2frq
Time for 2nd Frequency. Minimum time for 2nd tone in an SIT to remain in bounds
Default: 0  Units: 10 msec

canctime2frq
Maximum Time for 2nd Frequency. Maximum allowable time for 2nd tone in an SIT to be present
Default: 0  Units: 10 msec

calower3frq
Lower Bound for 3rd Frequency. Lower bound for 3rd tone in an SIT
Default: 0  Units: Hz

caupper3frq
Upper Bound for 3rd Frequency. Upper bound for 3rd tone in an SIT
Default: 0  Units: Hz

catime3frq
Time for 3rd Frequency. Minimum time for 3rd tone in an SIT to remain in bounds
Default: 0  Units: 10 msec

camxtime3frq
Maximum Time for 3rd Frequency. Maximum allowable time for 3rd tone in an SIT to be present
Default: 0  Units: 10 msec

cadtn_pres
Dial Tone Present. Length of time that a dial tone must be continuously present (PerfectCall CPA only)
Default: 100  Units: 10 msec

cadttn_npres
Dial Tone Not Present. Maximum length of time to wait before declaring dial tone failure (PerfectCall CPA only)
Default: 300  Units: 10 msec

cadttn_deboff
Dial Tone Debounce. Maximum gap allowed in an otherwise continuous dial tone before it is considered invalid (PerfectCall CPA only)
Default: 10  Units: 10 msec

capamdfailtime
PAMD Fail Time. Maximum time to wait for positive answering machine detection or positive voice detection after a cadence break (PerfectCall CPA only)
Default: 400  Units: 10 msec

capamd_minring
Minimum PAMD Ring. Minimum allowable ring duration for positive answering machine detection (PerfectCall CPA only)
Default: 190  Units: 10 msec

capamdspdval
PAMD Speed Value. Quick or full evaluation for PAMD detection
call progress analysis parameters — DX_CAP

- PAMD_FULL – Full evaluation of response
- PAMD_QUICK – Quick look at connect circumstances (PerfectCall CPA only)
- 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_FULL

c_ca_pamd_qtemp
  
PAMD Qualification Template. Which PAMD template to use. Options are
  PAMD_QUAL1TMP or PAMD_QUAL2TMP; at present, only PAMD_QUAL1TMP is available. (PerfectCall CPA only)
  
  Default: PAMD_QUAL1TMP

c_ca_noanswer
  
  No Answer. Length of time to wait after first ringback before deciding that the call is not answered. (PerfectCall CPA only)
  
  Default: 3000 Units: 10 msec

c_ca_maxintering
  
  Maximum Inter-ring Delay. Maximum time to wait between consecutive ringback signals before deciding that the call has been connected. (PerfectCall CPA only)
  
  Default: 800 Units: 10 msec

Example

For an example of DX_CAP, see the Example section for dx_dial().
DX_CST — call status transition (CST) information

DX_CST

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 or TDX_SETHOOK 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_LCOFF – loop current off
   - DE_LCON – loop current on
   - DE_LCREV – loop current reversal
   - DE_RINGS – rings received
   - DE_RNGOFF – caller hang up event (incoming call is dropped before being accepted)
   - DE_SILOFF – non-silence detected
   - DE_SILON – silence detected
   - DE_TONEOFF – tone off event
   - DE_TONEON – tone on event
   - DE_WINK – received a wink
   - DX_OFFHOOK – offhook event
   - DX_ONHOOK – onhook event

   Note: DX_ONHOOK and DX_OFFHOOK are returned if a TDX_SETHOOK termination event is received.

cst_data
   Contains data associated with 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_LCOFF – time previous last loop current on transition in 10 msec units
   - DE_LCON – time since previous loop current off transition in 10 msec units
   - DE_LCREV – time since previous loop current reversal transition in 10 msec units
   - DE_RINGS – 0
   - DE_SILOFF – time since previous silence started in 10 msec units
   - DE_SILON – time since previous silence stopped in 10 msec units
   - DE_TONEOFF – user-specified tone ID
   - DE_TONEON – user-specified tone ID
   - DE_WINK – N/A
   - DX_OFFHOOK – N/A
• DX_ONHOOK – N/A

Example

For an example of how to use the DX_CST structure, see the Example section for `dx_sendevt()` and `dx_setevtmsk()`. 
**DX_EBLK**

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_setevtmsk()`.

**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_LCOFF` – loop current off
    - `DE_LCON` – loop current on
    - `DE_LCREV` – loop current reversal
    - `DE_RINGS` – rings received
    - `DE_SILOFF` – non-silence detected
    - `DE_SILON` – silence detected
    - `DE_TONEOFF` – tone off event
    - `DE_TONEON` – tone on event
    - `DE_WINK` – received a wink
    - `DX_OFFHOOK` – offhook event
    - `DX_ONHOOK` – onhook event

  DX_ONHOOK and DX_OFFHOOK are returned if a TDX_SETHOOK termination event is received.

- **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_LCOFF` – length of time that loop current was on before the loop-current-off event was detected
    - `DE_LCON` – length of time that loop current was off before the loop-current-on event was detected
    - `DE_LCREV` – length of time that loop current was reversed before the loop-current-reversal event was detected
    - `DE_RINGS` – 0 (no data)
    - `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
- DE_WINK – (no data)
- DX_OFFHOOK – (no data)
- DX_ONHOOK – (no data)

**Example**

For an example of how to use the DX_EBLK structure, see the Example section for `dx_getevt()` and `dx_setevtsk()`.
DX_ECRCT — echo cancellation resource (ECR) characteristics

typedef struct dx_ecrct {
    int ct_length;            /* size of this structure */
    unsigned char ct_NLPflag  /* ECR with NLP requested or not */
} DX_ECRCT;

#define SIZE_OF_ECR_CT sizeof (DX_ECRCT)   /* size of DX_ECRCT */
#define ECR_CT_ENABLE 0
#define ECR_CT_DISABLE 1

Description

The DX_ECRCT data structure describes echo cancellation resource (ECR) characteristics. This structure is used by the dx_listenecrex( ) function.

Note: The ECR feature has been replaced by the continuous speech processing (CSP) feature. CSP provides enhanced echo cancellation. For more information, see the Continuous Speech Processing API Programming Guide and Continuous Speech Processing API Library Reference.

Field Descriptions

The fields of the DX_ECRCT data structure are described as follows:

ct_length
Specifications the size of this structure. Use the following value to accommodate future growth in the DX_ECRCT and the possibility of DX_ECRCT structures with different sizes:

- SIZE_OF_ECR_CT – size of the DX_ECRCT structure

ct_NLPflag
Specifies whether non-linear processing (NLP) is enabled or not. When NLP is enabled, the output of the echo canceller is replaced with an estimate of the background noise. NLP provides full echo suppression as long as the echo reference signal contains speech signals and the echo-carrying signal does not. In this case, the echo canceller cancels the echo and maintains the full duplex connection. Note: Do not enable NLP when using the echo canceller output for voice recognition algorithms because the NLP may clip the beginning of speech.

The ct_NLPflag default is disabled. Values are:

- ECR_CT_ENABLE – enables NLP
- ECR_CT_DISABLE – disables NLP (default)

Note: The application must include the following line in order to handle DX_ECRCT structures of different sizes without the need for recompiling the application:

ecrct.ct_length=size_of_ecr_ct;

Example

See dx_listenecrex( ) for an example of how to use the DX_ECRCT structure.
DX_IOTT

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()` and `lseek()`.

To use multiple combinations, each source or destination of I/O is specified as one element in an array of DX_IOTT structures. The last DX_IOTT entry must have IO_EOT specified in the io_type field.

**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 W A V E data offset I/O. Set the io_type field to an OR combination of the following defines.

On **DM3 boards**, specify the data transfer type as follows:

- IO_CACHED – cached prompt
- 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

On **Springware boards**, specify the data transfer type as follows:

- IO_DEV  – file data
- IO_MEM  – memory data
• 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).

Wave files cannot be recorded to memory buffers or played from memory buffers.

io_fhandle
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.

Note: When playing a GSM WAVE file and using an offset, you must set the io_length field to the actual length of the file. Setting this field to -1 is not supported.

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.
#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. */
DX_STREAMSTAT

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.
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 — speed and volume adjustment condition block**

**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.

This structure is used by `dx_setsvcond()` function to specify a play adjustment condition that is added to the internal speed and volume condition table (SVCT). The play adjustment conditions in the SVCT are used to adjust speed or volume automatically at the beginning of playback or in response to digits entered by the user during playback.

The `dx_setsvcond()`, `dx_addspddig()`, and `dx_addvoldig()` functions can be used to add play adjustment conditions to the SVCT. These functions tie a speed or volume adjustment to an external event, such as a DTMF digit.

You cannot change an existing speed or volume 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 (speed modification table, volume modification table)
- adjustment type (step, index, toggle, pause/resume play)
- adjustment size or action
- adjustment condition (incoming digit, beginning of play)
- level/edge sensitivity for incoming digits

For more information on speed and volume modification tables as well as the pause and resume play feature, 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 one adjustment type, either speed or volume
      - `SV_SPEEDTBL` – selects speed table to be modified
      - `SV_VOLUMETBL` – selects volume table to be modified
    - **Adjustment Method** (required except for pause/resume play): 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 speed or 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 speed modification table, the default entries for index values -10 to -6 and +6 to +10 are -128 which represent a null-entry. 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 speed or volume relative to the current position in the table. Specify a positive number of steps to increase the current speed or volume, or a negative number of steps to decrease it. For example, specify -2 to lower the speed (or volume) by two steps in the speed (or 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 speed and volume settings and allow you to toggle the speed or 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. 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 speed or volume level without using a digit condition. The digit and digtype fields are ignored.

• **SV_PAUSE** – Use with SV_SPEEDTBL to pause the play on detection of the specified DTMF digit.

• **SV_RESUME** – Use with SV_SPEEDTBL to resume the play on detection of the specified DTMF digit.

**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)

On **DM3 boards**, the following are valid values:

• **SV_TOGORIGIN** – sets the digit to toggle between the origin and the last modified speed or volume level (for example, between the -5 and 0 levels)

• **SV_CURORIGIN** – resets the current speed or volume level to the origin (same effect as SV_ABSPOS with adjsize 0)

On **Springware boards**, the following are valid values:
• SV_TOGORIGIN – sets the digit to toggle between the origin and the last modified speed or volume level (for example, between the -5 and 0 levels)
• SV_CURORIGIN – resets the current speed or volume level to the origin (same effect as SV_ABSPOS with adjsize 0)
• SV_CURLASTMOD – sets the current speed or volume to the last modified speed volume level (swaps the current and last-modified settings)
• SV_RESETORIG – resets the current speed or volume to the origin and the last modified speed or volume to the origin

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;
```

This example illustrates how to set a DTMF digit to adjust playback speed. The following DX_SVCB structure will set the playback speed to the value in the speed modification table position 5 whenever the DTMF digit 2 is detected:

```c
svcb[0].type     = SV_SPEEDTBL | SV_ABSPOS;
svcb[0].adjsize  = 5;
svcb[0].digit    = '2';
svcb[0].digtype  = DG_DTMF;
```

This example illustrates how to set a DTMF digit to pause and resume play.

```c
svcb[0].type     = SV_SPEEDTBL | SV_PAUSE;
svcb[0].adjsize  = 0;
svcb[0].digit    = '2';
svcb[0].digtype  = DG_DTMF;
svcb[0].type     = SV_SPEEDTBL | SV_RESUME;
svcb[0].adjsize  = 0;
svcb[0].digit    = '5';
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

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.

You can specify the rate of change for speed or volume adjustments by customizing the speed or volume modification table (SVMT) per channel. The DX_SVMT structure has 21 entries that represent different levels of speed or volume. This structure is used to set or retrieve the SVMT values, using `dx_setsvmt()` or `dx_getsvmt()` respectively.

For detailed information on speed and 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 speed or 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 speed or 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) speed or 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:

- Speed – Percentage decrease from the origin (which is set to 0). Values must be between -1 and -50.
- Volume – Decibel decrease from the origin (which is set to 0). Values must be between -1 and -30.

origin

Specifies the standard play speed or volume. This is the original setting or starting point for speed and volume control. Set the origin to 0 to assume normal playback speed/volume for the standard (normal volume is -8 dB).

increase[10]

Array that provides a maximum of 10 upward steps from the standard (normal) speed or 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:

- Speed – Percentage increase from the origin (which is set to 0). Values must be between 1 and 50.
- 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

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**

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.

The G.726 and GSM voice coders are supported by the I/O functions that use a DX_XPB data structure:

- The G.726 voice coder is supported by the `dx_playiottdata()`, `dx_reciottdata()`, `dx_playvox()`, and `dx_recvox()` functions.
- The GSM voice coders are supported by the `dx_playiottdata()`, `dx_reciottdata()`, and `dx_recwav()` functions.

For a list of voice coders supported on a board, see the Release Guide for your system release.

### 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.

On **DM3 boards**, use one of the following data formats:
- `DATA_FORMAT_DIALOOGIC_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 or 16-bit linear PCM
- `DATA_FORMAT_TRUESPEECH` – TrueSpeech coder
- `DATA_FORMAT_G721` – G.721 coder
- `DATA_FORMAT_G726` – G.726 bit-exact coder
• DATA_FORMAT_GSM610_MICROSOFT – GSM 6.10 full-rate coder (Microsoft Windows compatible format) (Microsoft Windows Media Recorder Audio Compression Codec: GSM 6.10 Audio CODEC)
• DATA_FORMAT_GSM610_TIPHON – GSM 6.10 VOX full-rate coder (TIPHON format)
• DATA_FORMAT_IMA_ADPCM – IMA ADPCM coder (IMA is an acronym for Interactive Multimedia Association)

On Springware boards, use one of the following data formats:
• DATA_FORMAT_DIALOGIC_ADPCM – 4-bit OKI ADPCM (Dialogic registered format)
• DATA_FORMAT_MULAW – 8-bit mu-law PCM
• DATA_FORMAT_ALAW – 8-bit A-law PCM
• DATA_FORMAT_PCM – 8-bit linear PCM
• DATA_FORMAT_G726 – G.726 bit-exact coder
• DATA_FORMAT_GSM610_MICROSOFT – GSM 6.10 full-rate coder (Microsoft Windows compatible format) (Microsoft Windows Media Recorder Audio Compression Codec: GSM 6.10 Audio CODEC)
• DATA_FORMAT_GSM610_TIPHON – GSM 6.10 VOX full-rate coder (TIPHON format)

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.
On DM3 boards, this number varies with the data format. For more information, refer to the Examples section next.
On Springware boards, set to 8 for mu-law, A-law, and linear PCM. Set to 4 for ADPCM. For G.726 and GSM, refer to the Examples section next.

Examples (DM3)

Table 17 through Table 24 provide examples of how to fill the DX_XPB structure for various voice coders on DM3 boards.

Table 17. G.711 Voice Coder Support Fields (DM3)

<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 DATA_FORMAT_G711_MULAW</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DATA_FORMAT_ALAW or DATA_FORMAT_G711_MULAW</td>
<td></td>
</tr>
<tr>
<td>nSamplesPerSec</td>
<td>DRT_6KHZ or DRT_8KHZ</td>
<td></td>
</tr>
<tr>
<td>wBitsPerSample</td>
<td>8</td>
<td>48 or 64 kbps</td>
</tr>
</tbody>
</table>

On Springware boards, use one of the following data formats:
### DX_XPB — input/output transfer parameter block

Table 18. G.721 Voice Coder Support Fields (DM3)

<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_G721</td>
<td></td>
</tr>
<tr>
<td>nSamplesPerSec</td>
<td>DRT_8KHZ</td>
<td></td>
</tr>
<tr>
<td>wBitsPerSample</td>
<td>4</td>
<td>32 kbps</td>
</tr>
</tbody>
</table>

Table 19. Linear PCM Voice Coder Support Fields (DM3)

<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_8KHZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DRT_11KHZ</td>
<td></td>
</tr>
<tr>
<td>wBitsPerSample</td>
<td>8 or 16</td>
<td>64, 128, 88 or 176 kbps</td>
</tr>
</tbody>
</table>

Table 20. OKI ADPCM Voice Coder Support Fields (DM3)

<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</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 21. G.726 Voice Coder Support Fields (DM3)

<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, 3, 4, or 5</td>
<td>16, 24, 32, or 40 kbps</td>
</tr>
</tbody>
</table>
Examples (Springware)

Table 25 and Table 26 provide examples of how to fill the DX_XPB structure for various voice coders on Springware boards.

Table 25. G.726 Voice Coder Support Fields (Springware)

<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_VOX</td>
<td></td>
</tr>
<tr>
<td>wDataFormat</td>
<td>DATA_FORMAT_G726</td>
<td></td>
</tr>
<tr>
<td>nSamplesPerSec</td>
<td>DRT_G726</td>
<td></td>
</tr>
<tr>
<td>wBitsPerSample</td>
<td>4</td>
<td>32 kbps</td>
</tr>
</tbody>
</table>
Table 26. GSM Voice Coder Support Fields (Springware)

<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</td>
<td></td>
</tr>
<tr>
<td>wDataFormat</td>
<td>DATA_FORMAT_GSM610_MICROSOFT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DATA_FORMAT_GSM610_TIPHON</td>
<td></td>
</tr>
<tr>
<td>nSamplesPerSec</td>
<td>DRT_8KHZ</td>
<td></td>
</tr>
<tr>
<td>wBitsPerSample</td>
<td>0</td>
<td>This field can be any numeric value; it is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ignored. However, the recommended setting is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13 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_FFT**
    - **FT_FSK_OH**
    - **FT_G729A** – supports G.729a encoding
    - **FT_ITU_G_726** – supports ITU-T G.726 encoding
    - **FT_LINEAR** – supports linear PCM encoding
    - **FT_MSGSM** – supports Microsoft GSM encoding
    - **FT_PCM** – supports PCM encoding
    - **FT_RAW64BIT** – supports raw 64 bit
    - **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.
**FEATURE_TABLE — feature information**

- FT_ADPCM – supports ADPCM encoding
- FT_ADSI – supports Analog Display Services Interface (ADSI)
- 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_FFT –
- FT_FSK_OH –
- FT_G729A – supports G.729a encoding
- FT_LINEAR – supports linear PCM encoding
- FT_MSGSM – supports Microsoft GSM encoding
- FT_PCM – supports PCM encoding
- FT_RESRVD1 – reserved
- FT_RESRVD2 – reserved
- FT_ULAW – supports mu-law encoding

**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_DPD – supports dial pulse detection
- FT_ECR – supports echo cancellation resource
- FT_SYNTTELLECT – supports Syntellect patent protection

**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_RS_SHARE – supports fax resource sharing
- FT_VFX40 – specifies that the device is a VFX/40 fax board
- FT_VFX40E – specifies that the device is a VFX/40E fax board
- FT_VFX40E_PLUS – specifies that the device is a VFX/40ESCplus or VFX/PCI board
- FT_FAX_EXT_TBL – specifies send fax and receive fax feature support.

On **Springware boards**, if this bit is turned on and the FT_SENDFAX_TXFILE_ASCII bit (in `ft_send`) is turned on, then the device supports DSP Fax (also known as Softfax).

On **DM3 boards**, if the `ft_fax` field contains the bitmask `FT_FAX | FT_VFX40 | FT_VFX40E | FT_VFX40E_PLUS`, then this device supports fax.

**ft_front_end**
Contains a bitmask of the front-end features supported on the specified device.

On **DM3 boards**, one or more of the following may be returned:
- FT_ANALOG_CID – returned by the Intel® Dialogic® DMV160LP board
- FT_CAS – supports CAS
- FT_ISDN – supports ISDN
- FT_R2MF – supports R2/MF signaling
• FT_ROUTEABLE – supports flexible routing configuration
For fixed routing, the FT_ROUTEABLE is not set, so none of the other bits is set. For flexible routing, the FT_ROUTEABLE bit is set, and the other three bits are set based on cluster contents.

For example, if the ft_front_end bitmask is FT_ROUTEABLE | FT_ISDN | FT_CAS, then the channel is capable of flexible routing and can also work with an ISDN or a CAS (T1) frontend. In this example, R2/MF is missing, so the channel cannot work with a front-end that is R2/MF (E1 CAS) capable. As another example, FT_ROUTEABLE | FT_ISDN | FT_CAS | FT_R2MF indicates support for flexible routing plus all three front-end capabilities, including R2/MF.

For more information on flexible and fixed routing configurations, see the Voice API Programming Guide.

Note: On DM3 analog boards, use dx_getctinfo( ) rather than dx_getfeaturelist( ) to return information about the type of front end or network interface on the board. The network interface information is contained in the ct_nettype field of CT_DEVINFO.

On Springware boards, one or more of the following may be returned:
• FT_ANALOG – supports analog interface
• FT_EARTH_RECALL – supports earth recall

ft_misc
Contains a bitmask of miscellaneous features supported on the specified device.
• FT_CALLERID – supports caller ID
• FT_CSPEXTRASLOT – reserves extra transmit time slot for continuous speech processing
• FT_GAIN_AND_LAW – TDM ASIC supports AGC and law conversion
• 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
Contains a bitmask of extended play features supported on the specified device.
• FT_TRUSPEECH – supports TrueSpeech decoding (supported on DM3 boards only)

ft_record_ext
Contains a bitmask of extended record features supported on the specified device.
• FT_TRUSPEECH – supports TrueSpeech encoding (supported on DM3 boards only)

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

typedef struct {
    unsigned long   sc_numts;
    long           *sc_tsarrayp;
} SC_TSINFO;

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 ag_ (analog), dt_ (digital network interface), dx_ (voice), fx_ (fax), and ms_ (modular station interface).

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.

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

typedef struct {
    unsigned short tg_dflag;  /* Dual Tone - 1, Single Tone - 0 */
    unsigned short tg_freq1;  /* Frequency for Tone 1 (HZ) */
    unsigned short tg_freq2;  /* Frequency for Tone 2 (HZ) */
    short tg_ampl1;  /* Amplitude for Tone 1 (dB) */
    short tg_ampl2;  /* Amplitude for Tone 2 (dB) */
    short tg_dur;    /* Duration of the Generated Tone */
    /* Units = 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()`.
cadenced tone generation template — TN_GENCAD

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.
On DM3 boards, valid values are 1 to 40 cycles.
On Springware boards, valid values are from 1 to 255 (255 = infinite repetitions).

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.
Tone[4]

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_SEG substructure as 6 instances called toneseg */
} 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 includes the TONE_SEG substructure as 6 instances called toneseg.

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_DATA — tone information

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().
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_BADPARAM**
  - 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_CLIDBLK
   Caller ID is blocked, or private, or withheld (other information may be available using
dx_gtextcallid( ))

EDX_CLIDINFO
   Caller ID information is not sent or caller ID information invalid

EDX_CLIDOOA
   Caller ID is out of area (other information may be available using dx_gtextcallid( ))

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 on a DM3 board

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, such as when a function is not supported on a DM3 board

EDX_SH_BADCMD
   Command is not supported in current bus configuration
Error Codes

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 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_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 either SV_SPEEDTBL or SV_VOLUMETBL

EDX_SV_ADJBLKS
Invalid number of speed/volume adjustment blocks

EDX_SVMTRANGE
Entry out of range in speed/volume modification table, SV_SVMT

EDX_SVMTSIZE
Invalid table size specified

EDX_SYSTEM
Error from operating system; use dx_fileerrno() to obtain error value

EDX_TIMEOUT
I/O function timed out

EDX_TONEID
Invalid tone template ID

EDX_TNMSGSTATUS
Invalid message status setting
Error Codes

EDX_UNSUPPORTED
Function is not supported

EDX_WTRINGSTOP
Wait-for-Rings stopped by user

EDX_XBPARAM
Bad XPB structure
Supplementary Reference

This chapter provides reference information on the following topics:

- DTMF and MF Tone Specifications ........................................ 539
- DTMF and MF Detection Errors ............................................. 540

### 6.1 DTMF and MF Tone Specifications

Table 27 provides information on DTMF specifications. Table 28 provides information on MF tone specifications.

**Table 27. 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, 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>
Table 28. MF Tone Specifications (CCITT R1 Tone Plan)

<table>
<thead>
<tr>
<th>Code</th>
<th>Tone Pair Frequencies (Hz)</th>
<th>Default Length (msec)</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>700, 900</td>
<td>60</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>700, 1100</td>
<td>60</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>900, 1100</td>
<td>60</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>700, 1300</td>
<td>60</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>900, 1300</td>
<td>60</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>1100, 1300</td>
<td>60</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>700, 1500</td>
<td>60</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>900, 1500</td>
<td>60</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1100, 1500</td>
<td>60</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td>1300, 1500</td>
<td>60</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1100, 1700</td>
<td>60</td>
<td>KP</td>
</tr>
<tr>
<td>#</td>
<td>1500, 1700</td>
<td>60</td>
<td>ST</td>
</tr>
<tr>
<td>a</td>
<td>900, 1700</td>
<td>60</td>
<td>ST1</td>
</tr>
<tr>
<td>b</td>
<td>1300, 1700</td>
<td>60</td>
<td>ST2</td>
</tr>
<tr>
<td>c</td>
<td>700, 1700</td>
<td>60</td>
<td>ST3</td>
</tr>
</tbody>
</table>

* The standard length of a KP tone is 100 msec

6.2 DTMF and MF Detection Errors

Some MF digits use approximately the same frequencies as DTMF digits (see Table 27 and Table 28). 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 29 and Table 30 show the digits that are detected when each type of detection is enabled. Table 29 shows which digits are detected when MF digits are sent. Table 30 shows which digits are detected when DTMF digits are sent.
Table 29. Detecting MF Digits

<table>
<thead>
<tr>
<th>MF Digit Sent</th>
<th>Only MF Detection Enabled</th>
<th>Only DTMF Detection Enabled</th>
<th>MF and DTMF Detection Enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2'</td>
<td>4,2'</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>3'</td>
<td>7,3'</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
</tbody>
</table>

* = detection error

Table 30. Detecting DTMF Digits

<table>
<thead>
<tr>
<th>DTMF Digit Sent</th>
<th>Only DTMF Detection Enabled</th>
<th>Only MF Detection Enabled</th>
<th>DTMF and MF Detection Enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4*</td>
<td>4,2'</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>7*</td>
<td>7,3'</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>4*</td>
<td>4,5'</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>7*</td>
<td>7,6'</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>5*</td>
<td>5,8'</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>8*</td>
<td>8,9'</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>5*</td>
<td>5,0'</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

* = detection error
Table 30. 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>#</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 E-1 areas. See also 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.

ADSI (Analog Display Services Interface): A Bellcore standard defining a protocol for the flow of information between a switch, a server, a voice mail system, a service bureau, or a similar device and a subscriber’s telephone, PC, data terminal, or other communicating device with a screen. ADSI adds words to a system that usually only uses touch tones. It displays information on a screen attached to a phone. ADSI’s signaling is DTMF and standard Bell 202 modem signals from the service to a 202-modem-equipped phone.

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: A board-level object that can be manipulated by a physical library. Board devices can be real physical boards, such as a D/41JCT-LS, or virtual boards. See virtual board.
board locator technology (BLT): Operates in conjunction with a rotary switch to determine and set non-conflicting slot and IRQ interrupt-level parameters, thus eliminating the need to set confusing jumpers or DIP switches.

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

bus: An electronic path 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. On DM3 boards, 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 facility 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. 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()` function, 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 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.

**E-1**: A CEPT digital telephony format devised by the CCITT, used in Europe and other countries around the world. A digital transmission channel that carries data at the rate of 2.048 Mbps (DS-1 level). CEPT stands for the Conference of European Postal and Telecommunication Administrations. Contrast with T-1.

**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.

**emulated device**: A virtual device whose software interface mimics the interface of a particular physical device, such as a D/4x boards that is emulated by a D/12x board. On a functional level, a D/12x board is perceived by an application as three D/4x boards. Contrast with physical device.

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

**event handler**: A portion of 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 hook state.

**frequency shift keying (FSK)**: A frequency modulation technique used to send digital data over voice band telephone lines.
**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.

**hook state:** A general term for the current line status of the channel: either on-hook or off-hook. A telephone station is said to be on-hook when the conductor loop between the station and the switch is open and no current is flowing. When the loop is closed and current is flowing, the station is off-hook. These terms are derived from the position of the old fashioned telephone set receiver in relation to the mounting hook provided for it.

**hook switch:** The circuitry that controls the on-hook and off-hook state of the voice device telephone interface.

**I/O:** Input-Output

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

**in-band:** The use of robbed-bit signaling (T-1 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.

**in-band signaling:** (1) In an analog telephony circuit, in-band refers to signaling that occupies the same transmission path and frequency band used to transmit voice tones. (2) In digital telephony, in-band means signaling transmitted within an 8-bit voice sample or time slot, as in T-1 “robbed-bit” signaling.

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

**loop:** The physical circuit between the telephone switch and the voice processing board.

**loop current:** The current that flows through the circuit from the telephone switch when the voice device is off-hook.

**loop current detection:** A voice driver feature that returns a connect after detecting a loop current drop.

**loop start:** In an analog environment, an electrical circuit consisting of two wires (or leads) called tip and ring, which are the two conductors of a telephone cable pair. The CO provides voltage (called “talk battery” or just “battery”) to power the line. When the circuit is complete, this voltage produces a current called loop current. The circuit provides a method of starting (seizing) a telephone line or trunk by sending a supervisory signal (going off-hook) to the CO.

**loop-start interfaces:** Devices, such as an analog telephones, that receive an analog electric current. For example, taking the receiver off-hook closes the current loop and initiates the calling process.

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

**off-hook:** The state of a telephone station when the conductor loop between the station and the switch is closed and current is flowing. When a telephone handset is lifted from its cradle (or an equivalent condition occurs), the telephone line state is said to be off-hook. See also hook state.
**on-hook:** Condition or state of a telephone line when a handset on the line is returned to its cradle (or an equivalent condition occurs). See also **hook state**.

**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.

**physical device:** A device that is an actual piece of hardware, such as a D/4x board; not an emulated device. See **emulated device**.

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

**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.

**robbed-bit signaling:** The type of signaling protocol implemented in areas using the T-1 telephony standard. In robbed-bit signaling, signaling information is carried in-band, within the 8-bit voice samples. These bits are later stripped away, or “robbed,” to produce the signaling information for each of the 24 time slots.

**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.

**SCR:** See **silence compressed record**.

**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 compressed record:** A recording that eliminates or limits the amount of silence in the recording without dropping the beginning of words that activate recording.
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 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.

T-1: The digital telephony format used in North America and Japan. In T-1, 24 voice conversations are time-division multiplexed into a single digital data stream containing 24 time slots. Signaling data are carried “in-band”; as all available time slots are used for conversations, signaling bits are substituted for voice bits in certain frames. Hardware at the receiving end must use the “robbed-bit” technique for extracting signaling information. T-1 carries data at the rate of 1.544 Mbps (DS-1 level).

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, piece-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.

**transparent signaling:** The mode in which a network interface device accepts signaling data from a resource device transparently, or without modification. In transparent signaling, outgoing T-1 signaling bits are generated by a TDM bus resource device. In effect the resource device performs signaling to the network.

**virtual board:** 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. For example, a D120JCTLS has 12 channels of voice processing and contains three 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.

**wink:** In T-1 or E-1 systems, a signaling bit transition from on to off, or off to on, and back again to the original state. In T-1 systems, the wink signal can be transmitted on either the A or B signaling bit. In E-1 systems, the wink signal can be transmitted on either the A, B, C, or D signaling bit. Using either system, the choice of signaling bit and wink polarity (on-off-on or off-on-off hook) is configurable through DTI/xxx board download parameters.
Index

A
ACLIP
  message types 265
adjusting speed and volume
  explicitly 133
  using conditions 401
  using digits 401
adjustment conditions
digits 402
  maximum number 402
  setting 401
ADPCM 300, 337
ADSI 20
  functions 20
  two-way 426
ADSI_XFERSTRUC data structure 478
ag_getctinfo( ) 36
ag_getxmitslot( ) 38
ag_listen( ) 41
ag_unlisten( ) 44
AGC 337
ai_close( ) 46
ai_getxmitslot( ) 48
ai_open( ) 50
A-law 300, 520
alowmax 500
analog devices
  connecting to time slot 41
  disconnecting from TDM bus 44
  get time slot number 38
  getting information about 36
Analog Display Services Interface (ADSI) 20
ansrdgl 501
answering machine detection 52
array 510
asynchronous operation
dailing 191
  digit collection 227
  playing 301
  playing tone 315
  recording 338
  setting hook state 384
  stopping I/O functions 417
  wink 436
  ATDX_ functions 27
  ATDX_ANSRSIZ( ) 52
  ATDX_BNAMEP( ) 54
  ATDX_BDTYPE( ) 56
  ATDX_BUFDIGS( ) 58, 130
  ATDX_CHNAMES( ) 60
  ATDX_CHNUM( ) 62
  ATDX_CONNTYPE( ) 64
  ATDX_CPERROR( ) 67
  ATDX_CPTERM( ) 67, 70
  ATDX_CRTNID( ) 73
  ATDX_DEVTYPE( ) 76
  ATDX_DTNFAIL( ) 78
  ATDX_FRQDUR( ) 81
  ATDX_FRQDUR2( ) 83
  ATDX_FRQDUR3( ) 85
  ATDX_FRQHZ( ) 87
  ATDX_FRQHZ2( ) 89
  ATDX_FRQHZ3( ) 91
  ATDX_FQOUT( ) 93
  ATDX_FWVER( ) 95
  ATDX_HOOKST( ) 97, 385
  ATDX_LINEST( ) 99
  ATDX_LONGLOW( ) 101
  ATDX_PHYADDR( ) 103
  ATDX_SHORTLOW( ) 105
  ATDX_SIZEHI( ) 107
  ATDX_STATE( ) 109
  ATDX_TERMINSK( ) 111, 115
  ATDX_TONEID( ) 115
  ATDX_TRCOUNT( ) 118
  audio input functions
    ai_close( ) 46
    ai_getxmitslot( ) 48
    ai_open( ) 50
  audio pulse digits 374
  automated attendant 446
  automatic gain control 337
  backward signal
    specifying 466
base memory address 510
beep tone
  record notification 399
beep tone, pre-record 409
bits per sample 521
blowmax 500
board
  device 76, 292
  device name 54
  parameters 389, 390, 391, 393
  setting 54
  physical address 103
board device
  handle 60
  opening 292
breaking
  connection to a time slot 431
buffer
  firmware digit 173
buffer size
  bulk queue 367
buffer, ADSI data 478
busy channel
  forcing to idle state 417

c
ca_dtn_deboff 502
ca_dtn_npres 502
ca_dtn_pres 502
ca_lowerfrq 93
ca_maxintering 503
ca_noanswer 498, 503
ca_pamd_failtime 498, 502
ca_pamd_minrning 502
ca_pamd_qtemp 503
ca_pamd_spdval 498, 502
ca_upperfrq 93
cached prompt management functions
  dx_getcachesize( ) 151, 219
cached prompt management functions 21
cached prompts
  downloading voice data 151
  playing 310
  playing, example 313
  size of on-board memory 219
cadence 52
  repetition for user-defined tones 140
cadenced tone
  playing 319
call progress analysis 52, 70, 105, 107, 191
  activating 272
  answering machine detection 52
  cadence 52
  data structure 497
  enabling 189
  enhanced
    activating 272
  errors 67
  frequency detection
    SIT tones(tone 1) 81, 87
    SIT tones(tone 2) 83, 89
    SIT tones(tone 3) 85, 91
  functions 25
    dx_chgdur( ) 154
    dx_chgfreq( ) 158
    dx_chgrepcent( ) 162
  parameter structure 171
  results
    answer duration 192
    busy 70, 192
    call connected 192
    called line answered by 70
    connect 70
    connection type 192, 193
    error 71, 192, 193
    fax machine or modem 192
    frequency detection 193
    frequency out of bounds 193
    initial non-silence 107
    last termination 192, 193
    longer silence 101, 193
    no answer 70, 192
    no dial tone 192
    no ringback 70, 192
    non-silence 193
    operator intercept 70, 192
    shorter silence 105, 193
    stopped 71, 192
    timeout 70
    stopping 193, 418
    termination 70
    tone definitions 154, 158, 162
    using dx_dial( ) 188
call progress analysis functions
  dx_initcallp( ) 272
call progress signals, standard 320
call progress tone 180, 184, 334
call status transition
  DX_CST data structure 504
  event block structure 506
  event handling 379
  synchronously monitoring events 234
call status transition event functions 21
  dx_getevt() 234
  dx_setevtmsk() 377

call status transition structure 504
caller ID
  common message types 263
caller ID functions 26
  dx_gcallid() 258
  dx_gtextcallid() 262
  dx_wtcallid() 440

call
channel
  bulk queue buffer sizing function 367
  current state 109
  device 76, 292
  digit buffer 226
  monitoring activity 99
  names 60
  number 62
  number of processes 234
  parameters 391, 393
  status
    dial 109
    DTMF signal 99
    get digit 109
    idle 109
    no loop current 99
    no ringback 99
    onhook 99
    play 109
    playing tone 109
    record 109
    ringback present 99
    silence 99
    stopped 109
channel device information structure 479
channel parameters 393, 395
CLASS
  message types 264
clearing structures 171, 177
CLIP
  message types 265
close() 166
close() function, Windows 166
closing devices 166
cnosig 497, 498
cnosil 499
coders 520
common message types 263
compelled signaling 466
CON_CAD 64
CON_LPC 64

CON_PAMD 64
CON_PVD 64
configuration functions 16
  dx_clrdbuf() 173
  dx_getparm() 242
  dx_GetRscStatus() 245
  dx_gtsernum() 270
  dx_libinit() 275
  dx_setchxfercnt() 367
  dx_setdigtyp() 374
  dx_sethook() 384
  dx_setparm() 388
  dx_settonelen() 409
  dx_TSFSstatus() 421
  dx_wtring() 443
connect
  event 52
  type 64
continuous speech processing (CSP) 26
convenience functions
  dx_play() 307
  dx_playvox() 324
  dx_recf() 343
  dx_recvox() 351
  dx_recwav() 354
I/O 19
  nr_scroute() 451
  nr_scunroute() 457
R2/MF 23
  speed and volume 24
TDM Routing 22
CR_BUSY 70, 192
CR_CEPT 70, 87, 89, 91, 192
CR_CNCT 64, 70, 192
CR_ERROR 67, 192
CR_FAXTONE 70, 192
CR_LGTUERR 67
CR_MEMERR 67
CR_MXFRQERR 67
CR_NOANS 70, 192
CR_NODIALTONE 70, 192
CR_NORB 70, 192
CR_OVRLPERR 68
CR_STOPD 71, 192
CR_TMOUTOFF 68
CR_TMOUTON 68
CR_UNEXPTN 68
CR_UPFRQERR 68
CS_CALL 109
CS_DIAL 109
CS_GTGID 109
CS_HOOK 109
CS_IDLE 109
CS_PLAY 109
CS_RECE 109
CS_RECVFAX 110
CS_SENDFAX 110
CS_STOPD 109
CS_TONE 109
CS_WINK 110
CSP 26
cst_data 504
cst_event 504
CT bus
references to 22
CT_DEVINFO data structure 36, 221, 479
current parameter settings 242
cycles 531

D
D_APD 374
D_DPD 374
D_DPDZ 374
D_DTMF 374
D_MF 374
data formats 520
data structure
user digit buffer 483
data structures
cadenced tone generation template 531
call progress analysis parameters 497
call status transition 504
clearing 27
echo cancellation resource 508
event block 506
feature information 525
I/O
user-definable 519
I/O transfer table 509
input/output transfer parameter block 520
speed and volume adjustment conditions 514
speed modification table 517
Syntellect license auto attendant 493
TDM bus time slot information 529
termination parameter table 485
tone generation template 530
DE_DIGITS event 474, 504, 506
DE_DIGOFF event 474
DE_LCOFF event 474, 504, 506
DE_LCON event 475, 504, 506
DE_LCREV event 475, 504, 506
DE_RINGS event 475, 504, 506
DE_RNGOFF event 475, 504
DE_SILOFF event 474, 475, 504, 506
DE_SILONN event 474, 475, 504, 506
DE_STOPGETEVT event 474, 475
DE_STOPRINGS event 475
DE_STOPWTRING event 475
DE_TONEOFF event 474, 475, 504, 506
DE_TONEON event 474, 475, 504, 506
DE_WINK event 475, 504, 506
device
opening 50, 292
device handle 16, 56, 292
freeing 166
device information structure 479
device management functions 15
dx_close() 166
dx_open() 292
device names
displaying 60
device type 76
devices
closing 46, 166
multiple processes 166
returning features 525
type 56
DG_DTMF 483
DG_END 483
DG_LPD 483
DG_MAXDIGS 227, 483
DG_MF 483
dg_type 483
DG_USER1 483
dg_value 483
DI_D41BD 56
DI_D41CH 56
dial pulse digit (DPD) 374
dial tone
tone failure 78
dialing
  ASCII string 188
  asynchronous 189, 191
  DTMF 190, 191
  enabling call progress analysis 189
  flash 190, 191
  flash character 191
  MF 190, 191
  pause 190, 191
  pause character 191
  pulse 190, 191
  specifying dial string 188, 190
  stopping 193
  synchronous 189, 191
  synchronous termination 191
  termination events
    TDX_CALLP 191, 472
    TDX_DIAL 191, 472
  with call progress analysis 189, 191

digit buffer 226, 227
  flushing 173

digit buffer, user 483

digit collection 226
  asynchronous 227
  DTMF digits 226
  MF digits 226
  synchronous 227
  termination 227
  user-defined digits 226

digit detection 226
  audio pulse 374
  dial pulse 374
  disabling 197
  DPD, zero-train 374
  DTMF 374
  DTMF vs. MF tones 375
  errors 540
  mask 374
  MF 374
  multiple types 375
  setting digit types 374

digits
  adjustment conditions 402
  collecting 58
  defines for user-defined tones 125
  detecting 58
  speed and volume 130

disabling detection
  user-defined tones 197

disconnecting
  voice receive channel 431

DLL version number functions
  dx_GetDllVersion() 232

DM_DIGITS 378
DM_DIGOFF 378
DM_LCON 378
DM_LCREV 378
DM_RINGS 378, 443
DM_RNGOFF 378
DM_SILOF 378
DM_SILON 378
DM_UNDERRUN 378
DM_WINK 378

DPD
  support 375

DSP fax 245, 527
DT_DXBD 76
DT_DXCH 76

DTMF 541
  detection errors 540
  tone specifications 539

DTMF digits 374
  collection 226
  overlap with MF digits 227

DV_DIGIT data structure 226, 483
  specifying 226

DV_TPT data structure 485
  clearing 177
  contiguous 177
  last entry in 177
  linked 177

dx_addspddig() 120
ndx_addtone() 124
ndx_addvoldig() 129
ndx_adjsv() 133

DX_ATTENDANT data structure 493

dx_blddt() 136
ndx_blddtcad() 139
ndx_bldst() 142
ndx_bldstcad() 145
ndx_bldtngen() 148
ndx_cacheprompt() 151

DX_CAP data structure 497
  clearing 171

dx_chgdur() 154
ndxchgfreq() 158
ndx_chgrecpt() 162
ndx_close() 166
ndx_CloseStream() 169
ndx_clrcap() 171
ndx_clrdigbuf() 58, 173, 227
dx_clrsvcond( ) 175, 401
dx_clrpt( ) 177
dx_createtone( ) 179
DX_CST data structure 504
hook state terminations 385
dx_deltones( ) 186
dx_dial( ) 52, 112, 171, 194, 417
dx_distone( ) 124, 197
DX_EBLK data structure 234, 506
DX_ECRCT data structure 508
dx_enbtone( ) 124, 200
dx_fileclose( ) 203
dx_fileerrno( ) 205
dx_fileopen( ) 208
dx_filedread( ) 210
dx_fileseek( ) 213
dx_filewrit( ) 216
dx_getcachesize( ) 219
dx_getctinfo( ) 221
dx_getcursv( ) 223
dx_getdig( ) 58, 173, 226, 483
dx_GetDllVersion( ) 232
dx_getevt( ) 234, 379, 443, 506
dx_getfeaturelist( ) 237
FEATURE_TABLE data structure 525
dx_getparm( ) 242, 301, 337, 389
dx_GetRscStatus( ) 245
dx_GetStreamInfo( ) 247
dx_getsvmt( ) 249
dx_getxmitslot( ) 252
dx_getxmitslotecr( ) 255
dx_gtcallid( ) 258
dx_gtextcallid( ) 262
dx_getsernum( ) 270
dx_initcallp( ) 272
DX_IOTT data structure 509
dx_list( ) 277
dx_listencre( ) 280
dx_listencrex( ) 283
DX_ECRCT data structure 508
dx_mreciottdata( ) 286
DX_OFFHOOK event 97, 444, 475, 504, 506
DX_ONHOOK event 97, 444, 475, 504, 506
dx_open( ) 292
dx_OpenStreamBuffer( ) 295
dx_pause( ) 297
dx_play( ) 173, 299, 307, 510
dx_playf( ) 307
dx_playiottdata( ) 310
dx_playtone( ) 315
dx_playtoneEx( ) 319
dx_playvox( ) 324
dx_playwav( ) 327
dx_PutStreamData( ) 330
dx_query( ) 333
dx_querytone( ) 183
dx_rec( ) 173, 336, 510
dx_reccf( ) 343
dx_reciottdata( ) 347
dx_recvox( ) 351
dx_recwav( ) 354
dx_ResetStreamBuffer( ) 357
dx_resume( ) 359
dx_RxIottData( ) 361
dx_sendevt( ) 365
dx_setchxfercnt( ) 367
dx_setdevuio( ) 369
dx_setdigbuf( ) 372
dx_setdigtyp( ) 226
dx_setevtmask( ) 234, 377, 443
dx_setgtdamp( ) 382
dx_sethook( ) 112, 384, 443, 493
dx_setparam( ) 301, 337, 388
dx_SetRecordNotifyBeepTone( ) 399
dx_setsvcond( ) 401
dx_setsvmt( ) 405
dx_settonelen( ) 409
dx_setuio( ) 367, 412
dx_SetWaterMark( ) 415
dx_stopch( ) 193, 336, 361, 417
DX_STREAMSTAT data structure 512
DX_SVCB data structure 401, 514
DX_SVMT data structure 405, 517
dx_TSStatus( ) 421
dx_TxIottData( ) 423
dx_TxRxIottData( ) 426
DX_UIO data structure 519
used by dx_setdevuio( ) 369
dx_unlisten( ) 431
dx_unlistencre( ) 433
dx_wink( ) 436
dx_wtcallid( ) 440
dx_wtring( ) 377, 443
Extended attribute functions

- ATDX_ANSRSIZ( ) 52
- ATDX_BDNAMEP( ) 54
- ATDX_BDTYPE( ) 56
- ATDX_BUFDIGS( ) 58
- ATDX_CHNAMES( ) 60
- ATDX_CHNUM( ) 62
- ATDX_CONNTYPE( ) 64
- ATDX_CPTERM( ) 70
- ATDX_CPTERM( ) 70
- ATDX_CRTSID( ) 73
- ATDX_DEVTYPE( ) 76
- ATDX_DTNF( ) 78
- ATDX_FQDUR( ) 81
- ATDX_FQDUR( ) 83
- ATDX_FQDUR( ) 85
- ATDX_FQHZ( ) 87
- ATDX_FQHZ2( ) 89
- ATDX_FQHZ3( ) 91
- ATDX_FQOUT( ) 93
- ATDX_FWVER( ) 95
- ATDX_HOOKST( ) 97
- ATDX_LINEST( ) 99
- ATDX_LONGLOW( ) 101
- ATDX_PHYADDR( ) 103
- ATDX_SHORTLOW( ) 105
- ATDX_SIZEHI( ) 107
- ATDX_STAT( ) 109
- ATDX_TERMSK( ) 111
- ATDX_TONEID( ) 115
- ATDX_TRCOUNT( ) 118

Extended attribute functions category 27

Fax

- Fax 110
- Fax resource 245

Feature information data structure 526

FEATURE_TABLE data structure 525

File format 520

File manipulation functions 26

- dx_fileclose( ) 203
- dx_fileerrno( ) 205
- dx_fileopen( ) 208
- dx_fileread( ) 210
- dx_fileread( ) 213
- dx_filewrite( ) 216

Firmware

- Buffer 58
- Emulated D/4x version number 95
- Returning version number 95

Firmware digit buffer 173

Fixed length string 243
flash character 191
flexible routing configuration
  bitmask 527
flushing digit buffer 173
forward signal
  specifying 462
FSK
  two-way 426
full-duplex connection 41, 451, 457
functions
  ADSI 20
  ATDX_ 27
  cached prompt management 21
  call progress analysis 25
  call status transition Event 21
  caller ID 26
  configuration 16
  device management 15
  extended attribute 27
  global tone detection 23
  global tone generation 23
  I/O 17
  I/O convenience 19
  R2/MF convenience 24
  speed and volume 24
  speed and volume convenience 24
  structure clearance 27
  TDM routing 21
  Windows
    close(_) 166

global tone detection
  adding a tone 124
  deleting tones 186
  disabling 197
  dual frequency cadence tones 139
  dual frequency tones 136
  enabling 200
  enabling detection 124
  functions 23
    dx_addtone( ) 124
    dx_bldtd( ) 136
    dx_bldtdcad( ) 139
    dx_bldst( ) 142
    dx_bldstcad( ) 145
    dx_deltones( ) 186
    dx_dnstone( ) 197
    dx_enbtone( ) 200
    dx_setgtdamp( ) 382
  removing tones 186
  single frequency cadence tones 145
  single frequency tones 142
  global tone generation
  functions 23
    dx_bldtngen( ) 148
    dx_playtone() 315
    dx_playtoneEx( ) 319
  playing a cadenced tone 319
  playing a tone 315
  template 530
GSM voice coder 520, 521
GTD Frequency Amplitude
  setting 382

G

G.711 PCM voice coder 520
G.721 voice coder 520
G.726 voice coder 520
  global dial pulse detection
  support 375

H

half-duplex connection 41, 451, 457
hedge 499
hilbmax 499
hilceil 501
hiltolb 499
hiltolb 499
higltch 500
hisiz 500
hook state 97, 166
  setting 384
  termination events 385
hookstate 384
I/O function 111
transfer parameter block structure 520
transfer table 509
user-defined structure for 519
I/O convenience functions 19
I/O functions 17
dx_dial() 188
dx_getdig() 226
dx_mreciottdata() 286
dx_play() 299
dx_playouttdata() 310
dx_rec() 336
dx_reciottdata() 347
dx_RxIottData() 361
dx_setdigbuf() 372
dx_stopch() 417
dx.TxIottData() 423
dx.TxRxIottData() 426
dx_wink() 436
IMA ADPCM voice coder 521
inter-process event communication 365
intflg 497, 500
io_bufp 510
IO_CACHED 509
IO_CONT 177, 510
IO_DEV 509
IO_EOT 177, 509, 510
io_fhandle 510
io_length 510
IO_LINK 177, 510
IO_MEM 509
io_nextp 510
io_offset 510
io_prevp 510
IO_STREAM 509
io_type 509
IO_UIO 509, 510
IO_USEOFFSET 510

M
maxansr 501
MD_ADPCM 300, 337
MD_GAIN 337
MD_NOGAIN 337
MD_PCM 300, 337
message type ID 262
message types
ACLIP (multiple data message) 265
CLASS (multiple data message) 264
CLIP 265
common to CLASS, ACLIP, and CLIP 263
JCLIP (multiple data message) 266
MF
detection 541
detection errors 540
digit detection 374
digits
collection 226
support 191, 375
tone specifications 539
MF digits
overlap with DTMF digits 227
monitor channels 234
monitoring events 234

mu-law 520
mxtime2frq 502
mxtime3frq 502
mxtimefrq 501

N
names
   board device  54
nbrbeg  501
nbrdna  498
non-standard I/O devices
   dx_setdevuio( ) 369
dx_setuio( ) 412
nr_scroute( ) 451
nr_scunroute( ) 457
nsbusy  500
numsegs  531

O
offset  510
offtime  531
off-hook  97
off-hook state  384
OKI ADPCM voice coder  520
on-hook  97
on-hook state  384
open( ) function  293
opening devices  50, 292
operator intercept  81

P
parameter settings
   getting current  242
parameters
   board and channel  389, 391, 393, 395
call progress analysis  171
sizes  243
pause character  191
pause play  297
PBX call progress signals  320
physical address  103
physical board device
   closing  166
closing, example  167
opening  292
opening, example  293

play
asynchronous  301
currency function  307
default algorithm  300
default rate  301
encoding algorithm  300
mode  301
pausing  297, 515
resuming  359, 515
specifying mode  300
specifying number of bytes  510
synchronous  302
termination  302
   TDX_PLAY  301
termination events  301
tone
   asynchronous  315
   asynchronous termination events  315
   synchronous operation  316
transmitting tone before  300
voice data  324
play and record functions
   dx_mreciottdata( )  286
dx_pause( )  297
dx_play( )  299
dx_playf( )  307
dx_playvox( )  324
dx_rec( )  336
dx_recf( )  343
dx_reciottdata( )  347
dx_recox( )  351
dx_recwav( )  354
dx_resume( )  359
playback
   bytes transferred  118
playing
   see play  301
playing voice data  310
PM_ADSI  300
PM_BYTE  243
PM_FLSTR  243
PM_INT  243
PM_LONG  243
PM_SHORT  243
PM_SR6  300
PM_SR8  300
PM_TONE  300, 409
PM_VLSTR  243
positive answering machine detection  64
positive voice detection  64
pre-record beep  409
processes per channel  234
Pulse Code Modulation  300, 337

R

R2/MF
  compelled signaling  466
  convenience functions  24
  enabling signal detection  462
  playing backward signal  466
  specifying forward signal  462
  user-defined tone IDs  462, 463
R2/MF convenience functions  24
  r2_creatfsig()  462
  r2_playbsig()  466
r2_creatfsig()  462
r2_playbsig()  466
record notification beep tone  399

recording
  algorithm  337
  asynchronous  338
  asynchronous termination event
    TDX_RECORD  338
  bytes transferred  118
  convenience function  343
  default algorithm  337
  default gain setting  337
  default sampling rate  337
  gain control  337
  mode  337, 338
  sampling rate  337
  specifying mode  337
  specifying number of bytes  510
  stopping  336
  synchronous  339
  synchronous termination  339
  voice data  336, 347, 351
  WAVE data  354
  with A-law  337
  with tone  337

resources
  DSP fax  245
  echo cancellation  255, 280, 283, 433

rings
  wait for specified number  443

RLS_DTMF  99
RLS_HOOK  99
RLS_LCSENSE  99
RLS_RING  99
RLS_RINGBK  99
RLS_SILENCE  99
RM_ALAW  337
RM_SR6  337
RM_SR8  337
RM_TONE  337, 409

routing convenience functions
  nr_scroute( )  451
  nr_scunroute( )  457

routing functions
  ag_getctinfo( )  36
  ag_getxmitslot( )  38
  ag_listen( )  41
  ag_unlisten( )  44
  dx_getctinfo( )  221
  dx_getxmitslot( )  252
  dx_listen( )  277
  dx_listenevt()  280
  dx_unlisten( )  431

S

sampling rates  521
SC_TSINFO data structure  38, 41, 529
SCbus
  references to  22
sctools.c  451, 457
serial number
  retrieving  270
setting hook state  384
  asynchronous  384
  synchronous  385
short messaging services (SMS)  20
SIGALRM  444
sigset( )  444
silicon serial number
  retrieving  270
SIT tones
  detection  83, 85, 87, 89, 91
Softfax  527
  using dx_GetRsCStatus( )  245

speed
  adjusting  120
  adjustment conditions  401
  explicitly adjusting  133
  retrieving current  223

speed and volume
  current  134
  data structure  514
  last modified  134
  modification table
    setting  517
    resetting to origin  134
speed and volume convenience functions
   dx_addspddig() 120
   dx_addvoldig() 129

speed and volume function
   dx_setsvmt() 405

speed and volume functions 24
   dx_adjsv() 133
   dx_clrsvcond() 175
   dx_getcursv() 223
   dx_getsvmt() 249
   dx_setsvcond() 401

speed and volume modification table
   resetting to defaults 405, 406
   retrieving contents 249
   specifying speed 405
   specifying volume 405
   updating 405

speed control 517
   sr_getevtdatap() 379

stop I/O functions
   dial 417
   termination reason
      TM_USRSTOP 417
   wink 418

stopping call progress analysis 418

stopping I/O functions
   synchronous 417

streaming to board
   creating stream buffer 295
   deleting stream buffer 169
   DX_STREAMSTAT data structure 512
   function summary 19
   getting status info 247
   putting data in buffer 330
   resetting internal data 357
   setting water mark 415

structure clearance functions 27
   dx_clrcap() 171
   dx_clrtpt() 177

structures
   clearing 171, 177
   digit buffer 226
   DV_DIGIT 226
   DX_CAP 171
   DX_EBLK 234
   DX_IOTT 299
   event block 234
   SV_ABSPOS 134
   SV_CURLASTMOD 134
   SV_CURORIGIN 134
   SV_RELCURPOS 134
   SV_RESETORIG 134
   SV_SPEEDTBL 133
   SV_TOGGLE 134
   SV_TOGORIGIN 134
   SV_VOLUMETBL 133

synchronous operation
   dial 191
      digit collection 227
      play 302
      playing tone 316
      record 339
      setting hook state 385
      stopping I/O functions 417, 418
      wink 436

Syntellect license automated attendant functions
   li_attendant() 446
   li_islicensed() 450

syntellect.c 447
syntellect.h 493

T

TDM bus
   time slot information structure 529

TDM bus routing
   echo cancellation resource 255, 280, 283, 433

TDM bus routing functions 21
   dx_getctinfo() 221
   dx_getxmitslot() 252
   dx_listen() 277
   dx_listenecc() 280
   dx_unlisten() 431

   TDX_CACHEPROMPT event 472
   TDX_CALLP event 191, 472
   TDX_CREATETONE event 472
   TDX_CREATETONE_FAIL event 472
   TDX_CST event 472
   TDX_DELETETONE event 472
   TDX_DELETETONE_FAIL 472
   TDX_DIAL event 191, 472
   TDX_ERROR event 472
   TDX_GETDIG event 472
   TDX_HIGHWATER event 473
   TDX_LOWWATER event 473
   TDX_NOSTOP event 472
   TDX_PLAY event 301, 472
   TDX_PLAYTONE event 315, 320, 472
   TDX_QUERYTONE event 472
   TDX_QUERYTONE_FAIL event 472
   TDX_RECORD event 338, 473
TDX_RXDATA event 428, 473
TDX_SETHOOK 506
TDX_SETHOOK event 385, 473, 504
TDX_TXDATA event 428, 473
TDX_UNDERRUN event 473
TDX_WINK event 473

termination
call progress analysis 70
stop I/O function 417
synchronous record 339

termination conditions 17
termination events 471
DX_CST data structure 385
TDX_SETHOOK 385
TDX_WINK 436

termination parameter table structure 485

terminations
asynchronous play 301
ATDX_TERMMSK( ) 111
end of data 111, 112, 361, 423, 428
function stopped 112, 362, 423, 428
I/O device error 111, 112, 361, 423, 428
I/O function 111
I/O functions 417
inter-digit delay 111, 112
loop current off 112
maximum DTMF count 111, 112
maximum function time 111, 112, 361, 423, 428
maximum period of non-silence 111, 112
maximum period of silence 111, 112
normal termination 112
pattern matched 112
specific digit received 111, 112
synchronous play 302
tone-on/off event 112
tg_dflag 530
tg_freq1 530
TID_BUSY1 73
TID_BUSY2 73
TID_DIAL_IN 73
TID_DIAL_LCL 73
TID_DIAL_XTRA 73
TID_DISCONNECT 73
TID_FAX1 73
TID_FAX2 73
TID_RINGBK1 74
TID_RINGBK2 74
time slot device information structure 479
time2frq 502
time3frq 502

timefrq 501
TM_DIGIT termination 111, 112
TM_EOD termination 111, 112, 361, 423, 428
TM_ERROR termination 111, 112, 361, 423, 428
TM_IDDTIME termination 111, 112
TM_LCOFF termination 112
TM_MAXDATA termination 111, 112, 361, 423, 428
TM_MAXDTMF termination 112
TM_MAXDTMFtermination 111
TM_MAXNOSIL termination 112
TM_MAXNOSSILtermination 111
TM_MAXSIL termination 111, 112
TM_MAXTIME termination 111, 112, 361, 423, 428
TM_NORMTERM termination 112
TM_PATTERN termination 112
TM_TONE termination 112
TM_USRSTOP termination 112, 362, 423, 428
TN_GEN data structure 399, 530, 531
TN_GENCAD data structure 399, 530
	
tone 532
adding 124
enabling detection 124
tone definitions 148, 154, 158, 162
tone generation template 399, 530
tone ID 115, 136, 180, 184, 334
tone identifier 73
tone, pre-record beep 409
tone, record notification beep tone 399
TONE_DATA data structure 179, 333
trailing edge notification
user-defined tones 136
transaction record feature 286
TrueSpeech voice coder 520
TSF function 421
two-way FSK 426
T1 436

U
unsolicited events 473
upper2frq 501
upper3frq 502
upperfrq 501
user digit buffer 483
user-defined
cadence 140
user-defined digits
collection 226
user-defined functions
installing 369, 412
user-defined input/output data structure 519
user-defined Tone ID
R2/MF 462
user-defined tone ID 115
user-defined tones 124
cadence repetition 140
disabling detection 197
dual frequency 136
dual frequency cadence 139
enabling detection 200
first frequency 136
first frequency deviation 136
ID 136
leading or trailing edge notification 136
playing 319
also see playing tone 315
removing 186
second frequency 136
second frequency deviation 136
single frequency 142
single frequency cadence 145
tone ID 463

V
variable length string 243
version number
firmware 95
product 232
voice coders 520
elements (DM3) 521
elements (Springware) 523
volume
adjusting 129
adjustment conditions 401
explicitly adjusting 133
retrieving current 223
volume control 517

W
water mark 415
WAVE files
playing 327
Windows functions
close(_ ) 166
wink 436
asynchronous 436
on non-E&M line 436
synchronous 436
termination event 436

Z
zero-train DPD 374