



Dialogic[®] Voice API

Library Reference

December 2010

Copyright and Legal Notice

Copyright © 2004-2010, Dialogic Corporation. All Rights Reserved. You may not reproduce this document in whole or in part without permission in writing from Dialogic Corporation at the address provided below.

All contents of this document are furnished for informational use only and are subject to change without notice and do not represent a commitment on the part of Dialogic Inc. and its affiliates or subsidiaries ("Dialogic"). Reasonable effort is made to ensure the accuracy of the information contained in the document. However, Dialogic does not warrant the accuracy of this information and cannot accept responsibility for errors, inaccuracies or omissions that may be contained in this document.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH DIALOGIC® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN A SIGNED AGREEMENT BETWEEN YOU AND DIALOGIC, DIALOGIC ASSUMES NO LIABILITY WHATSOEVER, AND DIALOGIC DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF DIALOGIC PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY INTELLECTUAL PROPERTY RIGHT OF A THIRD PARTY.

Dialogic products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

Due to differing national regulations and approval requirements, certain Dialogic products may be suitable for use only in specific countries, and thus may not function properly in other countries. You are responsible for ensuring that your use of such products occurs only in the countries where such use is suitable. For information on specific products, contact Dialogic Inc. at the address indicated below or on the web at www.dialogic.com.

It is possible that the use or implementation of any one of the concepts, applications, or ideas described in this document, in marketing collateral produced by or on web pages maintained by Dialogic may infringe one or more patents or other intellectual property rights owned by third parties. Dialogic does not provide any intellectual property licenses with the sale of Dialogic products other than a license to use such product in accordance with intellectual property owned or validly licensed by Dialogic and no such licenses are provided except pursuant to a signed agreement with Dialogic. More detailed information about such intellectual property is available from Dialogic's legal department at 926 Rock Avenue, San Jose, California 95131 USA. Dialogic encourages all users of its products to procure all necessary intellectual property licenses required to implement any concepts or applications and does not condone or encourage any intellectual property infringement and disclaims any responsibility related thereto. These intellectual property licenses may differ from country to country and it is the responsibility of those who develop the concepts or applications to be aware of and comply with different national license requirements.

Dialogic, Dialogic Pro, Dialogic Blue, Veraz, Brooktrout, Diva, Diva ISDN, Making Innovation Thrive, Video is the New Voice, Diastar, Cantata, TruFax, SwitchKit, SnowShore, Eicon, Eicon Networks, NMS Communications, NMS (stylized), Eiconcard, SIPcontrol, TrustedVideo, Exnet, EXS, Connecting to Growth, Fusion, Vision, PowerMedia, PacketMedia, BorderNet, inCloud9, I-Gate, Hi-Gate, NaturalAccess, NaturalCallControl, NaturalConference, NaturalFax and Shiva, among others as well as related logos, are either registered trademarks or trademarks of Dialogic Inc. and its affiliates or subsidiaries. Dialogic's trademarks may be used publicly only with permission from Dialogic. Such permission may only be granted by Dialogic's legal department at 926 Rock Avenue, San Jose, California 95131 USA. Any authorized use of Dialogic's trademarks will be subject to full respect of the trademark guidelines published by Dialogic from time to time and any use of Dialogic's trademarks requires proper acknowledgement. The names of actual companies and products mentioned herein are the trademarks of their respective owners. This document discusses one or more open source products, systems and/or releases. Dialogic is not responsible for your decision to use open source in connection with Dialogic products (including without limitation those referred to herein), nor is Dialogic responsible for any present or future effects such usage might have, including without limitation effects on your products, your business, or your intellectual property rights.

Microsoft, Windows, Windows Server and Windows 7 are registered trademarks of Microsoft Corporation in the United States and/or other countries. Other names of actual companies and product mentioned herein are the trademarks of their respective owners. Other names of actual companies and product mentioned herein are the trademarks of their respective owners.

Publication Date: December 2010

Document Number: 05-2333-007

Contents

Revision History	9
About This Publication	12
Purpose	12
Applicability	12
Intended Audience	13
How to Use This Publication	13
Related Information	13
1 Function Summary by Category	14
1.1 Device Management Functions	14
1.2 Configuration Functions	15
1.3 I/O Functions	16
1.4 I/O Convenience Functions	17
1.5 Streaming to Board Functions	18
1.6 Analog Display Services Interface (ADSI) Functions	18
1.7 Transaction Record Functions	19
1.8 Call Status Transition (CST) Event Functions	19
1.9 TDM Routing Functions	20
1.10 Global Tone Detection (GTD) Functions	21
1.11 Global Tone Generation (GTG) Functions	22
1.12 Speed and Volume Functions	22
1.13 Call Progress Analysis Functions	23
1.14 Caller ID Functions	24
1.15 File Manipulation Functions	24
1.16 Structure Clearance Functions	24
1.17 Extended Attribute Functions	25
2 Function Information	27
2.1 Function Syntax Conventions	27
ag_getctinfo() – get information about an analog device	28
ag_getxmitslot() – get TDM bus time slot number of analog transmit channel	30
ag_listen() – connect analog receive channel to TDM bus time slot	34
ag_unlisten() – disconnect analog receive channel from TDM bus	38
ATDX_ANSRSIZ() – return the duration of the answer	40
ATDX_BDNAMEP() – return a pointer to the board device name	42
ATDX_BDTYPE() – return the board type for the device	44
ATDX_BUFDIGS() – return the number of uncollected digits	46
ATDX_CHNAMES() – retrieve all channel names for a board	48
ATDX_CHNUM() – return the channel number	50
ATDX_CONNTYPE() – return the connection type for a completed call	52
ATDX_CPERROR() – return the call progress analysis error	55
ATDX_CPTERM() – return the last result of call progress analysis termination	58
ATDX_CRTNID() – return the last call progress analysis termination	61

Contents

ATDX_DEVTYPE() – return the device type	67
ATDX_DTNFAIL() – return character for dial tone	69
ATDX_FRQDUR() – return the duration of the first SIT sequence	72
ATDX_FRQDUR2() – return the duration of the second SIT sequence	74
ATDX_FRQDUR3() – return the duration of the third SIT sequence	76
ATDX_FRQHZ() – return the frequency of the first SIT sequence	78
ATDX_FRQHZ2() – return the frequency of the second SIT sequence	80
ATDX_FRQHZ3() – return the frequency of the third SIT sequence	82
ATDX_FRQOUT() – return percentage of time SIT tone was out of bounds	84
ATDX_FWVER() – return the voice firmware version number	86
ATDX_HOOKST() – return the current hook-switch state	89
ATDX_LINEST() – return the current activity on the channel	91
ATDX_LONGLOW() – return duration of longer silence detected	93
ATDX_PHYADDR() – return the physical board address	95
ATDX_SHORTLOW() – return duration of shorter silence detected	97
ATDX_SIZEHI() – return duration of initial non-silence	99
ATDX_STATE() – return the current state of the channel	101
ATDX_TERMMSK() – return the reason for the last I/O function termination	103
ATDX_TONEID() – return user-defined tone ID that terminated I/O function	107
ATDX_TRCOUNT() – return the byte count for the last I/O transfer	110
dx_addspddig() – set a DTMF digit to adjust speed	112
dx_addtone() – add a user-defined tone	116
dx_addvoldig() – set a DTMF digit to adjust volume	121
dx_adjsv() – adjust speed or volume immediately	125
dx_blddt() – define a user-defined dual-frequency tone	129
dx_blddtcad() – define a user-defined dual frequency cadenced tone	132
dx_bldst() – define a user-defined single-frequency tone	135
dx_bldstcad() – define a user-defined single-frequency cadenced tone	138
dx_bldtngen() – define a tone for generation	141
dx_chgdur() – change the duration definition for a tone	144
dx_chgfreq() – change the frequency definition for a tone	148
dx_chgrepcnt() – change the repetition definition for a tone	152
dx_close() – close a channel or board device handle	155
dx_CloseStreamBuffer() – delete a circular stream buffer	157
dx_clrncap() – clear all fields in a DX_CAP structure	159
dx_clrdigbuf() – clear all digits in the firmware digit buffer	161
dx_clrsvcond() – clear all speed or volume adjustment conditions	163
dx_clrtp() – clear all fields in a DV_TPT structure	165
dx_createtone() – create a new tone definition for a specific call progress tone	167
dx_deletetone() – delete a specific call progress tone	171
dx_deltone() – delete all user-defined tones	174
dx_dial() – dial an ASCII string	177
dx_dialtp() – dial an outbound call using a TPT	185
dx_distone() – disable detection of a user-defined tone	188
dx_enbtone() – enable detection of a user-defined tone	191
dx_fileclose() – close a file	194

dx_fileerrno() – return the system error value	196
dx_fileopen() – open a file	199
dx_fileread() – read data from a file.	201
dx_fileseek() – move a file pointer.	204
dx_filewrite() – write data from a buffer into a file	207
dx_getctinfo() – get information about a voice device	210
dx_getcursv() – return the specified current speed and volume settings	212
dx_getdig() – collect digits from a channel digit buffer	215
dx_getdigEx() – collect more than 31 digits from a channel digit buffer	221
dx_getevt() – monitor channel events synchronously	224
dx_getfeaturelist() – retrieve feature support information for the device	227
dx_getparm() – get the current parameter settings	231
dx_GetRscStatus() – return assignment status of a shared resource.	234
dx_GetStreamInfo() – retrieve information about the circular stream buffer	236
dx_getsvmt() – return the current speed or volume modification table	238
dx_getxmitslot() – get TDM bus time slot number of voice transmit channel	241
dx_gtcallid() – return the calling line Directory Number.	244
dx_gtextcallid() – retrieve a caller ID message	250
dx_initcallp() – initialize and activate call progress analysis	260
dx_listen() – connect a voice listen channel to TDM bus time slot	263
dx_listenEx() – connect a voice listen channel to TDM bus time slot	266
dx_mreciottdata() – record voice data from two TDM bus time slots	270
dx_open() – open a voice device and return a unique device handle	278
dx_OpenStreamBuffer() – create and initialize a circular stream buffer	281
dx_play() – play recorded voice data.	283
dx_playf() – synchronously play voice data.	291
dx_playiottdata() – play back recorded voice data from multiple sources.	294
dx_playtone() – play tone defined by TN_GEN structure	298
dx_playtoneEx() – play the cadenced tone defined by TN_GENCAD.	302
dx_playvox() – play voice data stored in a single VOX file	307
dx_playwav() – play voice data stored in a single WAVE file	310
dx_PutStreamData() – place data into a circular stream buffer.	313
dx_querytone() – get tone information for a specific call progress tone	315
dx_rec() – record voice data from a single channel.	318
dx_recf() – record voice data to a single file	326
dx_reciottdata() – record voice data to multiple destinations	330
dx_recm() – record voice data from two channels.	335
dx_recmf() – record voice data from a single channel.	341
dx_recvox() – record voice data to a single VOX file.	345
dx_recwav() – record voice data to a single WAVE file.	348
dx_resetch() – reset a channel that is hung	351
dx_ResetStreamBuffer() – reset internal data for a circular stream buffer	354
dx_RxlottData() – receive data on a specified channel.	356
dx_sendevt() – allow inter-process event communication.	360
dx_setchxfercnt() – set the bulk queue buffer size	363
dx_setdevuio() – install and retrieve user-defined I/O functions	365

Contents

dx_setdigbuf() – set the digit buffering mode	395
dx_setdigtyp() – control the types of digits detected by the voice channel.	397
dx_setevtmsk() – enable detection of call status transition (CST) events	400
dx_setgtdamp() – set up the tone detection amplitudes.	406
dx_sethook() – provide control of the hook switch status.	408
dx_setparm() – set physical parameters of a channel or board device	412
dx_setsvcond() – set conditions that adjust speed or volume of play	420
dx_setsvmt() – change default values of the speed or volume modification table	424
dx_settone() – adds a GTG tone template.	428
dx_settonelen() – change the duration of the built-in beep tone	430
dx_setuio() – install user-defined I/O functions	433
dx_SetWaterMark() – set water mark for the circular stream buffer.	436
dx_stopch() – force termination of currently active I/O functions	438
dx_TSFStatus() – return the status of tone set file loading	442
dx_TxlottData() – transmit data on a specified channel	444
dx_TxRxlottData() – start a transmit-initiated reception of data	447
dx_unlisten() – disconnect voice receive channel from TDM bus	452
dx_unlistenEx() – disconnect voice receive channel from TDM bus	454
dx_wtcallid() – wait for rings and report caller ID	457
dx_wtring() – wait for a specified number of rings	460
nr_scroute() – make a full or half-duplex connection	463
nr_scuroute() – break a full or half-duplex connection	465
3 Events	467
3.1 Overview of Events.	467
3.2 Termination Events.	467
3.3 Unsolicited Events	469
3.4 Call Status Transition (CST) Events.	470
4 Data Structures	472
ADSI_XFERSTRUC – ADSI 2-way FSK data transfer buffer	474
CT_DEVINFO – channel/time slot device information.	475
DV_DIGIT – user digit buffer	478
DV_DIGITEX – extended digit buffer.	480
DV_TPT – termination parameter table.	481
DX_CAP – call progress analysis parameters.	490
DX_CST – call status transition (CST) information	498
DX_EBLK – call status transition event block	500
DX_IOTT – input/output transfer table.	502
DX_STREAMSTAT – status of stream buffer	505
DX_SVCB – speed and volume adjustment condition block.	507
DX_SVMT – speed and volume modification tables	511
DX_UIO – user-defined input/output	513
DX_XPB – input/output transfer parameter block	514
FEATURE_TABLE – feature information	518
SC_TSINFO – TDM bus time slot information.	523
TN_GEN – tone generation template	524

	TN_GENCAD – cadenced tone generation template	525
	TONE_DATA – tone information	527
5	Error Codes	529
6	Supplementary Reference Information	533
6.1	DTMF and MF Tone Specifications	533
6.2	DTMF and MF Detection Errors	534
	Glossary	537
	Index	544

Tables

1	Valid Dial String Characters (HMP Software)	179
2	Valid Dial String Characters (Springware boards)	179
3	System Error Values	196
4	Caller ID Common Message Types	251
5	Caller ID CLASS Message Types (Multiple Data Message)	252
6	Caller ID ACLIP Message Types (Multiple Data Message)	252
7	Caller ID CLIP Message Types	253
8	Caller ID JCLIP Message Types (Multiple Data Message)	253
9	Caller ID-Related Error Codes Returned for Different Message Types	255
10	Play Mode Selections	285
11	Record Mode Selections	320
1	Voice Board Parameters (HMP Software)	413
2	Voice Board Parameters (Springware boards)	413
3	Voice Channel Parameters (HMP Software)	415
4	Voice Channel Parameters (Springware boards)	416
5	DV_TPT Field Settings Summary	486
6	G.711 Voice Coder Support Fields	515
7	Linear PCM Voice Coder Support Fields	515
8	OKI ADPCM Voice Coder Support Fields	516
9	G.726 Voice Coder Support Fields	516
10	GSM Voice Coder Support Fields	517
11	G.726 Voice Coder Support Fields (Springware boards)	517
12	GSM Voice Coder Support Fields (Springware boards)	517
13	DTMF Tone Specifications	533
14	MF Tone Specifications (CCITT R1 Tone Plan)	534
15	Detecting MF Digits	535
16	Detecting DTMF Digits	535

Revision History

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

Document No.	Publication Date	Description of Revisions
05-2333-007	December 2010	<p>Made global changes to add Dialogic® Springware Architecture board support for Dialogic® HMP Software.</p> <p>About This Publication chapter: Updated the Applicability section.</p> <p>Function Summary by Category chapter: Added Analog Display Services Interface (ADSI) Functions and Caller ID Functions sections.</p> <p>In Configuration Functions, added dx_GetRscStatus(), dx_gtsernum(), dx_setdigbuf(), dx_sethook(), dx_settonelen(), dx_TSFStatus(), dx_wtring().</p> <p>In I/O Functions, added dx_dialtpt(), dx_getdigEx().</p> <p>In Transaction Record Functions, added dx_recv() and dx_recmf().</p> <p>In Call Status Transition (CST) Event Functions, added dx_sendvt().</p> <p>In TDM Routing Functions, added ag_getctinfo(), ag_getxmitslot(), ag_listen(), ag_unlisten().</p> <p>In Global Tone Generation (GTG) Functions, added dx_settone().</p> <p>In Call Progress Analysis Functions, added dx_chgdur(), dx_chgfreq(), dx_chgrepcnt(), dx_initcallp().</p> <p>In Extended Attribute Functions, added 17 functions.</p> <p>Function Information chapter: Updated to include Springware board support. Added 42 functions for Springware board support: ag_getctinfo(), ag_getxmitslot(), ag_listen(), ag_unlisten(), ATDX_ANSRSIZ(), ATDX_DTNFAIL(), ATDX_FRQDUR(), ATDX_FRQDUR2(), ATDX_FRQDUR3(), ATDX_FRQHZ(), ATDX_FRQHZ2(), ATDX_FRQHZ3(), ATDX_FRQOUT(), ATDX_FWVER(), ATDX_HOOKST(), ATDX_LINEST(), ATDX_LONGLOW(), ATDX_PHYADDR(), ATDX_SHORTLOW(), ATDX_SIZEHI(), dx_chgdur(), dx_chgfreq(), dx_chgrepcnt(), dx_dialtpt(), dx_getdigEx(), dx_GetRscStatus(), dx_gtcalled(), dx_gtextcalled(), dx_initcallp(), dx_recv(), dx_recmf(), dx_RxlottData(), dx_sendvt(), dx_setdigbuf(), dx_sethook(), dx_settone(), dx_settonelen(), dx_TSFStatus(), dx_TxlottData(), dx_TxRxlottData(), dx_wtcalled(), dx_wtring().</p> <p>dx_addtone(): Added note about user-defined tones digit reporting in Parameter table, digits parameter description. [IPY00038053]</p> <p>dx_deltone(): Added note for Springware boards. [IPY00079097]</p> <p>dx_getdig(): Added information about the return value for DM3 boards versus Springware boards in Description section. [IPY00038453]</p> <p>dx_rec(), dx_reciottdata(), dx_recvox(), dx_recwav(): Updated caution about channels getting stuck when failing to listen to a TDM bus time slot prior to invoking a record operation; condition now returns an error.</p> <p>dx_setparm(): Added DXCH_DFLAGS for HMP Software (leading edge and trailing edge CST digit detection); added DXCH_DFLAGS example code.</p> <p>Events chapter: Added TDX_RXDATA, TDX_SETHOOK, TDX_TXDATA in Termination Events. Added Call Status Transition Events on Dialogic® Springware Boards.</p> <p>(continued)</p>

Revision History

Document No.	Publication Date	Description of Revisions
05-2333-007 (continued)	December 2010	<p>Data Structures chapter: Updated to include Springware board support. Added 2 data structures for Springware board support: ADSI_XFERSTRUC, DV_DIGITEX.</p> <p>Error Codes chapter: Added EDX_CLIDBLK, EDX_CLIDINFO, EDX_CLIDOOA, EDX_WTRINGSTOP error codes.</p>
05-2333-006	April 2009	<p>Function Summary by Category chapter: Added dx_setchxfercnt() in Configuration Functions and added ATDX_BUFDIGS() in Extended Attribute Functions.</p> <p>ATDX_BUFDIGS() function: Added.</p> <p>dx_OpenStreamBuffer() function: Added caution about calling dx_open() before calling this function. [IPY00045172]</p> <p>dx_setchxfercnt() function: Added.</p> <p>dx_setparm() function: Added DXCH_XFERBUFSIZE.</p> <p>DX_XPB structure: Added GSM 6.10 full-rate coder (Microsoft format and TIPHON format).</p>
05-2333-005	January 2008	<p>Made global changes to reflect Dialogic brand and changed title to "Dialogic® Voice API Library Reference."</p> <p>Function Summary by Category chapter: Added dx_resetch() function to I/O Functions section.</p> <p>dx_getdig() function: Corrected number of digits returned in Synchronous Operation (IPY00038453)</p> <p>dx_reciottdata() function: Added RM_VADNOTIFY and RM_ISCR modes.</p> <p>dx_resetch() function: Added new function.</p> <p>dx_setparm() function: Added DXCH_SCRFEATURE define.</p> <p>Events chapter: Added TDX_VAD event.</p>
05-2333-004	August 2006	<p>Function Summary by Category chapter: Added support for speed control in Speed and Volume Functions section. Added note about enabling speed control in CONFIG file.</p> <p>dx_addspddig() function: Added support for this function in HMP.</p> <p>dx_adjsv() function: Added support for speed control.</p> <p>dx_clrsvcond() function: Added support for speed control.</p> <p>dx_getcursv() function: Added support for speed control.</p> <p>dx_getsvmt() function: Added support for speed control.</p> <p>dx_listenEx() function: Added caution about using this function and dx_unlistenEx() rather than dx_unlisten() and dx_listen().</p> <p>dx_mreciottdata() function: Updated values for mode parameter.</p> <p>dx_setsvcond() function: Added support for speed control.</p> <p>dx_setsvmt() function: Added support for speed control.</p> <p>dx_unlistenEx() function: Added caution about using this function and dx_listenEx() rather than dx_unlisten() and dx_listen().</p> <p>Events chapter: Removed DE_DIGOFF event from Call Status Transition(CST) Events section; not supported.</p> <p>DX_CST data structure: Removed DE_DIGOFF value; not supported.</p> <p>DX_SVCB data structure: Added support for speed control.</p> <p>DX_SVMT data structure: Added support for speed control.</p> <p>Error Codes chapter: Added "speed" to EDX_SPDVOL, EDX_SVADJBLKS, EDX_SVMTRANGE error code descriptions.</p>

Document No.	Publication Date	Description of Revisions
05-2333-003	December 2005	<p>ATDX_CRTNID() function: Added support for this function in HMP.</p> <p>dx_createtone() function: Added note about SIT sequences not supported for toneid in the parameter description table. Also added this information in the Cautions section. Updated example code to show asynchronous mode.</p>
05-2333-003 (cont.)	December 2005	<p>dx_deletetone() function: Added note about SIT sequences not supported for toneid in the parameter description table. Also added this information in the Cautions section.</p> <p>dx_querytone() function: Added note about SIT sequences not supported for toneid in the parameter description table. Also added this information in the Cautions section.</p> <p>dx_reciottdata() function: Added support for MD_NOGAIN for mode parameter (previously missing).</p> <p>dx_setparm() function: Removed the following channel parameters: DXCH_AGC_MAXGAIN, DXCH_AGC_MEMORY_MAXIMUMSIZE, DXCH_AGC_MEMORY_SILENCERESSET, DXCH_AGC_NOISE_THRESHOLD, DXCH_AGC_SPEECH_THRESHOLD, and DXCH_AGC_TARGET_OUTPUTLEVEL. These are not supported on HMP. Added support for the DXCH_EC_ACTIVE channel parameter.</p> <p>CT_DEVINFO data structure: Added CT_NTT1 and CT_NTE1 as supported values for ct_nettype field. Corrected ct_busmode field values: CT_BMH100 (previously CT_H100) and CT_BMH110 (previously CT_H110). Added support for ct_ext_devinfo.ct_net_devinfo.ct_prototype field.</p> <p>DX_XPB data structure: Updated to indicate support for linear PCM 8 kHz 16-bit (128 Kbps) encoding method. In the Field Descriptions section, wDataFormat field was updated. In the Examples section, Linear PCM Voice Coder Support Fields table was updated.</p>
05-2333-002	April 2005	<p>Function Summary by Category chapter: Added Transaction Record Function section. Removed dx_GetDllVersion() and dx_libinit() functions from Configuration Functions section. Added dx_listenEx() and dx_unlistenEx() to TDM Routing Functions section.</p> <p>dx_GetDllVersion() function: Removed; not supported.</p> <p>dx_libinit() function: Removed; not supported.</p> <p>dx_listen() function: Updated Description section and Example code section.</p> <p>dx_listen() function: New TDM routing function that extends and enhances the dx_listen() function.</p> <p>dx_mreciottdata() function: Transaction record now supported in HMP.</p> <p>dx_unlistenEx() function: New TDM routing function that extends and enhances the dx_unlisten() function.</p> <p>Events chapter: Added TDX_LISTEN, TDX_LISTEN_FAIL, TDX_UNLISTEN, TDX_UNLISTEN_FAIL events to Termination Events section.</p>
05-2333-001	September 2004	Initial version of document.

About This Publication

The following topics provide information about this publication:

- [Purpose](#)
- [Applicability](#)
- [Intended Audience](#)
- [How to Use This Publication](#)
- [Related Information](#)

Purpose

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

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

Applicability

This document version is published for Dialogic® Host Media Processing Software Release 3.0WIN Service Update, Dialogic® Host Media Processing Software Release 3.1LIN Service Update, and Dialogic® Host Media Processing Software Release 4.1LIN Service Update.

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

This document may also be applicable to other software releases (including service updates) on Linux or Windows® operating systems. Check the Release Guide for your software release to determine whether this document is supported.

Intended Audience

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

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

How to Use This Publication

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

The information in this guide is organized as follows:

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

A glossary and index are provided for your reference.

Related Information

See the following for additional information:

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

Function Summary by Category

1

This chapter describes the categories into which the Dialogic[®] Voice API library functions can be logically grouped.

Note: Functions may not be supported on all available platforms and operating systems. Platform and operating system support are indicated for each function in [Chapter 2, “Function Information”](#).

- [Device Management Functions](#) 14
- [Configuration Functions](#) 15
- [I/O Functions](#) 16
- [I/O Convenience Functions](#) 17
- [Streaming to Board Functions](#) 18
- [Analog Display Services Interface \(ADSI\) Functions](#). 18
- [Transaction Record Functions](#) 19
- [Call Status Transition \(CST\) Event Functions](#) 19
- [TDM Routing Functions](#) 20
- [Global Tone Detection \(GTD\) Functions](#) 21
- [Global Tone Generation \(GTG\) Functions](#) 22
- [Speed and Volume Functions](#) 22
- [Call Progress Analysis Functions](#) 23
- [Caller ID Functions](#) 24
- [File Manipulation Functions](#) 24
- [Structure Clearance Functions](#). 24
- [Extended Attribute Functions](#) 25

1.1 Device Management Functions

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

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

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

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

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

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

The device management functions are:

dx_close()

closes a board or channel device handle

dx_open()

opens a board or channel device handle

1.2 Configuration Functions

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

returns information about the features supported on the device

dx_getparm()

gets the current parameter settings for an open device

dx_GetRscStatus()

returns the assignment status of a shared resource for the specified channel

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

Function Summary by Category

dx_setparm()

sets physical parameters for the device

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 *Dialogic® Standard Runtime Library API Programming Guide* for more information on asynchronous and synchronous operation.

A set of termination conditions can be specified for I/O functions, except for **dx_stopch()**. These conditions dictate what events will cause an I/O function to terminate. The termination conditions are specified just before the I/O function call is made. Obtain termination reasons for I/O functions by calling the extended attribute function **ATDX_TERMMSK()**. See the *Dialogic® Voice API Programming Guide* for information about I/O terminations.

The I/O functions are:

dx_dial()

dials an ASCIIZ string of digits

dx_dialtpt()

dials an outbound call with the ability to terminate call progress analysis

dx_getdig()

collects digits from a channel digit buffer

dx_getdigEx()

initiates the collection of digits from a channel digit buffer (for up to 127 digits plus the null terminator)

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_resetch()

recovers a channel that is “stuck” (busy or hung) and in a recoverable state, and brings it to an idle and usable state

dx_reciottdata()

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

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

- 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()**

Function Summary by Category

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 *Dialogic® Voice API Programming Guide* for more information about the streaming to board feature.

The streaming to board functions include:

dx_CloseStreamBuffer()

deletes a circular stream buffer

dx_GetStreamInfo()

retrieves information about the circular stream buffer

dx_OpenStreamBuffer()

creates and initializes a circular stream buffer

dx_PutStreamData()

places data into the circular stream buffer

dx_ResetStreamBuffer()

resets internal data for a circular stream buffer

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 *Dialogic® 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 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

[dx_recm\(\)](#)

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

[dx_recmf\(\)](#)

records voice data from two channels to a single file

1.8 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 Dialogic® 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

1.9 TDM Routing Functions

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

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

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

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

The TDM routing functions are:

[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_listenEx\(\)](#)

connects the listen (receive) component of a voice channel to a TDM bus time slot. This function extends and enhances the [dx_listen\(\)](#) function.

[dx_unlisten\(\)](#)

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

dx_unlistenEx()

disconnects the listen (receive) component of a voice channel from TDM bus time slot. This function extends and enhances the **dx_unlisten()** function.

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

1.10 Global Tone Detection (GTD) Functions

The global tone detection (GTD) functions define and enable detection of single and dual frequency tones that fall outside the range of those automatically provided with the voice driver. They include tones outside the standard DTMF range of 0-9, a-d, *, and #.

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

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)

1.11 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.

The **dx_settone()** function adds a GTG tone template, defined by the **TN_GEN** data structure, to the firmware. This definition can be used by the application for tone-initiated record. The customization of record pre-beep lets the user select the frequencies, amplitudes, and duration of the beep being played prior to record.

See the *Dialogic® 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**

dx_playtone()

plays a user-defined tone as defined in **TN_GEN** structure

dx_playtoneEx()

plays the cadenced tone defined by **TN_GENCAD** structure

dx_settone()

adds a user-defined tone template to the firmware (and customizes pre-record beep)

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

1.12 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 *Dialogic® Voice API Programming Guide* for more information about the speed and volume feature, and speed and volume modification tables.

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

1.13 Call Progress Analysis Functions

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

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_chgrepent()

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

1.14 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()`. See the *Dialogic® 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

1.15 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 function. The arguments for these 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

1.16 Structure Clearance Functions

These functions do not affect a device. The `dx_clrcap()` and `dx_clrtppt()` 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_clrtppt()`

clears all fields in a `DV_TPT` structure

1.17 Extended Attribute Functions

Dialogic® Voice API 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

ATDX_BUFDIGS()

returns the number of digits in the firmware since the last **dx_getdig()** or **dx_getdigEx()** 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_CPELOR()

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

Function Summary by Category

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

This chapter provides an alphabetical reference to the functions in the Dialogic® Voice API library. A general description of the function syntax convention is provided before the detailed function information.

Note: The “Platform” line in the function header table of each function indicates the platforms supported. “HMP Software” refers to Dialogic® HMP Software. “Springware boards” refers to Dialogic® Springware architecture boards that are supported with Dialogic® HMP Software. If a function is supported on a specific operating system only, this is also noted.

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()

Name: int ag_getctinfo(chdev, ct_devinfofop)
Inputs: int chdev • valid analog channel device handle
CT_DEVINFO *ct_devinfofop • pointer to device information structure
Returns: 0 on success
-1 on error
Includes: srllib.h
dxxxlib.h
Category: Routing
Mode: synchronous
Platform: Springware boards

■ Description

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

Parameter	Description
chdev	specifies the valid analog channel handle obtained when the channel was opened using dx_open()
ct_devinfofop	specifies a pointer to the data structure CT_DEVINFO

■ 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_BADPARAM**
Parameter error
- EDX_SH_BADEXTTS**
TDM bus time slot is not supported at current clock rate
- EDX_SH_BADINDEX**
Invalid Switch Handler library index number
- EDX_SH_BADTYPE**
Invalid channel type (voice, analog, etc.)

get information about an analog device — ag_getctinfo()

EDX_SH_CMDBLOCK
Blocking command is in progress

EDX_SH_LIBBSY
Switch Handler library is busy

EDX_SH_LIBNOTINIT
Switch Handler library is uninitialized

EDX_SH_MISSING
Switch Handler is not present

EDX_SH_NOCLK
Switch Handler clock fallback failed

EDX_SYSTEM
Error from operating system

■ Example

```
#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

- [dx_getctinfo\(\)](#)

ag_getxmitslot()

Name: int ag_getxmitslot(chdev, sc_tsinfofop)
Inputs: int chdev • valid analog channel device handle
SC_TSINFO *sc_tsinfofop • 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 boards

■ 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 523.

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

Parameter	Description
chdev	specifies the valid analog channel handle obtained when the channel was opened using dx_open()
sc_tsinfofop	specifies a pointer to the data structure SC_TSINFO

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_BADPARAM
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_BADINDEX	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_MISSING	Switch Handler is not present
EDX_SH_NOCLK	Switch Handler clock fallback failed
EDX_SYSTEM	Error from operating system

■ Example

```
#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);
}
```

ag_getxmitslot() — get TDM bus time slot number of analog transmit channel

■ **See Also**

- [ag_listen\(\)](#)
- [dx_listen\(\)](#)

get TDM bus time slot number of analog transmit channel — ag_getxmitslot()

ag_listen() — connect analog receive channel to TDM bus time slot

ag_listen()

Name: int ag_listen (chdev, sc_tsinfofop)

Inputs: int chdev • valid analog channel device handle
SC_TSINFO *sc_tsinfofop • 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 boards

■ 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.

Parameter	Description
chdev	specifies the valid analog channel device handle obtained when the channel was opened using dx_open()
sc_tsinfofop	specifies a pointer to the SC_TSINFO structure. For more information on this structure, see SC_TSINFO , on page 523.

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_BADPARAM
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 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 <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 */
    }
}
```

ag_listen() — connect analog receive channel to TDM bus time slot

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

/* Get TDM bus time slot connected to transmit of 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\(\)](#)
- [ag_unlisten\(\)](#)

connect analog receive channel to TDM bus time slot — ag_listen()

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 boards

■ 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.

Parameter	Description
-----------	-------------

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

■ 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_BADPARAM
Parameter error

EDX_SH_BADCMD
Command is not supported in current bus configuration

EDX_SH_BADINDX
Invalid Switch Handler index number

disconnect analog receive channel from TDM bus — ag_unlisten()

EDX_SH_BADLCLTS	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

```
#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\(\)](#)

ATDX_ANSRSIZ() — return the duration of the answer

ATDX_ANSRSIZ()

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

■ Description

The **ATDX_ANSRSIZ()** function returns the duration of the answer that occurs when **dx_dial()** or **dx_dialtpt()** 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.

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

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


return the duration of the answer — `ATDX_ANSRSIZ()`

```
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

- [dx_dial\(\)](#)
- [dx_dialtpt\(\)](#)
- [DX_CAP](#) data structure

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

ATDX_BDNAMEP()

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: HMP Software, Springware boards

■ 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.

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 a pointer to “Unknown device” if an invalid channel device handle is specified in **chdev**.

■ Example

```
#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()

```
.
/* 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() — return the board type for the device

ATDX_BDTYPE()

Name: long ATDX_BDTYPE(dev)
Inputs: int dev • valid board or channel device handle
Returns: board or channel device type if successful
 AT_FAILURE if error
Includes: srllib.h
 dxxxlib.h
Category: Extended Attribute
Mode: synchronous
Platform: HMP Software, Springware boards

■ 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.

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

Possible return values are the following:

DI_D41BD

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

DI_D41CH

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

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

return the board type for the device — ATDX_BDTYPE()

■ **Example**

```
#include <stdio.h>
#include <srllib.h>
#include <dxxplib.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() — return the number of uncollected digits

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: HMP Software, Springware boards

■ Description

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

Note: This function is supported on HMP Software but must be manually enabled. You must enable the function before the application is loaded in memory.

On Linux, to enable this function, add `SupportForSignalCounting = 1` in `/usr/diallogic/cfg/cheetah.cfg`. To subsequently disable this function, remove this line from the `.cfg` file.

On Windows®, to enable this function, set the Data field of `SupportForSignalCounting` to 1 in Key `HKEY_LOCAL_MACHINE\SOFTWARE\Dialogic\Cheetah\CC`. To subsequently disable this function, set this parameter to 0.

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

■ Cautions

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

■ Errors

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

■ **Example**

```
#include <fcntl.h>
#include <srllib.h>
#include <dxxxxlib.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("dxxxxB1C1",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 */

    /* On Linux only, use open function */
    if((iott.io_fhandle = open("prompt.vox", O_RDONLY)) == -1) {
        /* process error */
    }

    /* On Windows only, use dx_fileopen function */
    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:** HMP Software, Springware boards
-

■ 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.

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

■ 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

```
#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() — return the channel number

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:** HMP Software, Springware boards
-

■ 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: HMP Software, Springware boards

■ 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 *Dialogic® Voice API Programming Guide* for more information about call progress analysis.

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

Possible return values are the following:

CON_CAD
 Connection due to cadence break

CON_LPC (supported on Springware boards only)
 Connection due to loop current

CON_PAMD
 Connection due to positive answering machine detection

CON_PVD
 Connection due to positive voice detection

■ Cautions

None.

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

■ Errors

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

■ Example

```
#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_deltone(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_ERRMSG( 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_CPEERR( dxxxdev ) );
        dx_close( dxxxdev );
        exit( 1 );
    }
}
```

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

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

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

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

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

/*
 * 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
dxxplib.h
- Category:** Extended Attribute
- Mode:** synchronous
- Platform:** HMP Software, Springware boards
-

■ 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 *Dialogic® Voice API Programming Guide* for more information about call progress analysis.

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

■ 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_maxtimefrq field in **DX_CAP**. If the ca_mvertimefrq parameter for each SIT is nonzero, it must have a value greater than or equal to the ca_timefrq 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_mvertimefrq 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

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

main()
{
    int dxxxdev;
    int cares;

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

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


return the call progress analysis error — ATDX_CPERROR()

```
/*
 * 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()

- 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:** HMP Software, Springware boards
-

■ 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 *Dialogic® Voice API Programming Guide* for more information about call progress analysis.

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

Possible return values are the following:

- CR_BUSY**
Called line was busy.
- CR_CEPT**
Called line received Operator Intercept (SIT). Extended attribute functions provide information on detected frequencies and duration.
- CR_CNCT**
Called line was connected.
- CR_FAXTONE**
Called line was answered by fax machine or modem.
- CR_NOANS**
Called line did not answer.
- CR_NODIALTONE**
Timeout occurred while waiting for dial tone. Springware boards only.
- CR_NORB**
No ringback on called line.

return the last result of call progress analysis termination — ATDX_CPTERM()

CR_STOPD

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

CR_ERROR

Call progress analysis error occurred. Use **ATDX_CPELOR()** 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

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

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

        /* Clear DX_CAP structure */
        dx_clrkap(&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 */
```

ATDX_CPTERM() — *return the last result of call progress analysis termination*

```
        .  
        .  
        break;  
        .  
        .  
    case AT_FAILURE: /* Error */  
    }  
}
```

■ **See Also**

- [dx_dial\(\)](#)
- [DX_CAP](#) data structure

return the last call progress analysis termination — ATDX_CRTNID()

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
dxxlib.h

Category: Extended Attribute

Mode: synchronous

Platform: HMP Software, Springware boards

■ 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 *Dialogic® Voice API Programming Guide* for a description of call progress analysis.

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

On HMP Software, 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_DISCONNECT
Disconnect tone (post-connect)
- TID_FAX1
First fax or modem tone
- TID_FAX2
Second fax or modem tone
- TID_RNGBK1
Ringback (detected as single tone)

ATDX_CRTNID() — return the last call progress analysis termination

TID_RNGBK2
Ringback (detected as dual tone)

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

TID_SIT_INEFFECTIVE_OTHER or
TID_SIT_IO
Ineffective other SIT sequence

TID_SIT_NO_CIRCUIT or
TID_SIT_NC
No circuit found SIT sequence

TID_SIT_NO_CIRCUIT_INTERLATA or
TID_SIT_NC_INTERLATA
InterLATA no circuit found SIT sequence

TID_SIT_OPERATOR_INTERCEPT or
TID_SIT_IC
Operator intercept SIT sequence

TID_SIT_REORDER_TONE or
TID_SIT_RO
Reorder (system busy) SIT sequence

TID_SIT_REORDER_TONE_INTERLATA or
TID_SIT_RO_INTERLATA
InterLATA reorder (system busy) SIT sequence

TID_SIT_VACANT_CIRCUIT or
TID_SIT_VC
Vacant circuit SIT sequence

On Springware boards, 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

return the last call progress analysis termination — ATDX_CRTNID()

TID_FAX2
Second fax or modem tone

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.

■ Example1

This example applies to HMP Software.

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

main()
{
    DX_CAP  cap_s;
    int     ddd, car;
    char    *chnam, *dialstrg;
    long    tone_id;
    chnam   = "dxxxB1C1";
    dialstrg = "L1234";
    /*
     * Open channel
     */
    if ((ddd = dx_open( chnam, NULL )) == -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;
        }
    }
}
```

ATDX_CRTNID() — return the last call progress analysis termination

```
        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;
case CR_CEPT:
    printf(" Special tone received at %s\n", dialstrg );
    tone_id = ATDX_CRTNID(ddd); //ddd is handle that is returned by dx_open()

    switch (tone_id) {

case TID_SIT_NC:
    printf("No circuit found special information tone received\n");
    break;
case TID_SIT_IC:
    printf("Operator intercept special information tone received\n");
    break;
case TID_SIT_VC:
    printf("Vacant circuit special information tone received\n");
    break;
case TID_SIT_RO:
    printf("Reorder special information tone received\n");
    break;
case TID_SIT_NC_INTERLATA:
    printf("InterLATA no circuit found special information tone received\n");
    break;
case TID_SIT_RO_INTERLATA:
    printf("InterLATA reorder special information tone received\n");
    break;
case TID_SIT_IO:
    printf("Ineffective other special information tone received\n");
    break;
case TID_SIT_ANY:
    printf("Catch all special information tone received\n");
    break;
    }
    break;
default:
    break;
}

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

dx_close( ddd );
}
```

■ Example 2

This example applies to Springware boards.

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


return the last call progress analysis termination — ATDX_CRTNID()

```
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_deltone(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;
    }
}
```

ATDX_CRTNID() — return the last call progress analysis termination

```
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.

return the device type — ATDX_DEVTYPE()

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
dxxlib.h

Category: Extended Attribute

Mode: synchronous

Platform: HMP Software, Springware boards

■ Description

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

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

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**.

***ATDX_DEVTYPE()* — return the device type**

■ **Example**

```
#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
dxxplib.h

Category: Extended Attribute

Mode: synchronous

Platform: Springware boards

■ Description

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

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

Possible return values are the following:

- L Local dial tone
- I International dial tone
- X Special (“extra”) dial tone

■ Cautions

None.

■ Errors

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

■ Example

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

ATDX_DTNFAIL() — return character for dial tone

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

    chnam   = "dxxxBlC1";
    dialstrg = "L1234";

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

    /*
     * Delete any previous tones
     */
    if ( dx_deltone(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;
    }
}
```

return character for dial tone — ATDX_DTNFAIL()

```
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() — return the duration of the first SIT sequence

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 boards

■ Description

The **ATDX_FRQDUR()** function returns the duration of the first Special Information Tone (SIT) sequence in 10 msec units after **dx_dial()** or **dx_dialtpt()** 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;
    .
    .
}
```


return the duration of the first SIT sequence — ATDX_FRQDUR()

```
/* 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("\nFirst frequency detected - %ld Hz",ATDX_FRQHZ(chdev));
        printf("\nSecond frequency detected - %ld Hz", ATDX_FRQHZ2(chdev));
        printf("\nThird frequency detected - %ld Hz", ATDX_FRQHZ3(chdev));
        printf("\nDuration of first frequency - %ld ms", ATDX_FRQDUR(chdev));
        printf("\nDuration of second frequency - %ld ms", ATDX_FRQDUR2(chdev));
        printf("\nDuration of third frequency - %ld ms", ATDX_FRQDUR3(chdev));
        break;
    case CR_BUSY:
        break;
    .
    .
}
}
```

■ See Also

- [dx_dial\(\)](#)
- [dx_dialtpt\(\)](#)
- [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\(\)](#)

ATDX_FRQDUR2() — return the duration of the second SIT sequence

ATDX_FRQDUR2()

- Name:** long ATDX_FRQDUR2(chdev)
- Inputs:** int chdev • valid channel device handle
- Returns:** second frequency duration in 10 msec units if success
AT_FAILURE if error
- Includes:** srllib.h
dxxxlib.h
- Category:** Extended Attribute
- Mode:** synchronous
- Platform:** Springware boards
-

■ Description

The **ATDX_FRQDUR2()** function returns the duration of the second Special Information Tone (SIT) sequence in 10 msec units after **dx_dial()** or **dx_dialtpt()** 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

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

■ See Also

- **dx_dial()**
- **dx_dialtpt()**
- **ATDX_CPTERM()**
- **DX_CAP** data structure

return the duration of the second SIT sequence — ATDX_FRQDUR2()

- call progress analysis topic in the *Voice API Programming Guide*
- [ATDX_FRQDUR\(\)](#)
- [ATDX_FRQDUR3\(\)](#)
- [ATDX_FRQHZ\(\)](#)
- [ATDX_FRQHZ2\(\)](#)
- [ATDX_FRQHZ3\(\)](#)

ATDX_FRQDUR3() — return the duration of the third SIT sequence

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
 dxxxlib.h
Category: Extended Attribute
Mode: synchronous
Platform: Springware boards

■ Description

The **ATDX_FRQDUR3()** function returns the duration of the third Special Information Tone (SIT) sequence in 10 msec units after **dx_dial()** or **dx_dialtpt()** 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

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

■ See Also

- **dx_dial()**
- **dx_dialtpt()**
- **ATDX_CPTERM()**
- **DX_CAP** data structure

return the duration of the third SIT sequence — ATDX_FRQDUR3()

- call progress analysis topic in *Voice API Programming Guide*
- [ATDX_FRQDUR\(\)](#)
- [ATDX_FRQDUR2\(\)](#)
- [ATDX_FRQHZ\(\)](#)
- [ATDX_FRQHZ2\(\)](#)
- [ATDX_FRQHZ3\(\)](#)

ATDX_FRQHZ() — return the frequency of the first SIT sequence

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
dxxxlib.h
- Category:** Extended Attribute
- Mode:** synchronous
- Platform:** Springware boards
-

■ Description

The **ATDX_FRQHZ()** function returns the frequency in Hz of the first Special Information Tone (SIT) sequence after **dx_dial()** or **dx_dialtpt()** 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*.

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

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

main()
{
    int cares, chdev;
    DX_CAP capp;
    .
```

return the frequency of the first SIT sequence — `ATDX_FRQHZ()`

```
.
/* 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("\nFirst frequency detected - %ld Hz",ATDX_FRQHZ(chdev));
        printf("\nSecond frequency detected - %ld Hz", ATDX_FRQHZ2(chdev));
        printf("\nThird frequency detected - %ld Hz", ATDX_FRQHZ3(chdev));
        printf("\nDuration of first frequency - %ld ms", ATDX_FRQDUR(chdev));
        printf("\nDuration of second frequency - %ld ms", ATDX_FRQDUR2(chdev));
        printf("\nDuration of third frequency - %ld ms", ATDX_FRQDUR3(chdev));
        break;
    case CR_BUSY:
        break;
    .
    .
}
}
```

■ See Also

- [dx_dial\(\)](#)
- [dx_dialtpt\(\)](#)
- [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\(\)](#)

ATDX_FRQHZ2() — return the frequency of the second SIT sequence

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
dxxxlib.h
- Category:** Extended Attribute
- Mode:** synchronous
- Platform:** Springware boards
-

■ Description

The **ATDX_FRQHZ2()** function returns the frequency in Hz of the second Special Information Tone (SIT) sequence after **dx_dial()** or **dx_dialtpt()** 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*.

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

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

■ See Also

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

return the frequency of the second SIT sequence — ATDX_FRQHZ2()

- [ATDX_FRQHZ\(\)](#)
- [ATDX_FRQHZ3\(\)](#)
- [ATDX_FRQDUR\(\)](#)
- [ATDX_FRQDUR2\(\)](#)
- [ATDX_FRQDUR3\(\)](#)

ATDX_FRQHZ3() — return the frequency of the third SIT sequence

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
 dxxxlib.h
Category: Extended Attribute
Mode: synchronous
Platform: Springware boards

■ Description

The **ATDX_FRQHZ3()** function returns the frequency in Hz of the third Special Information Tone (SIT) sequence after **dx_dial()** or **dx_dialtpt()** 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*.

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

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

■ See Also

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

return the frequency of the third SIT sequence — ATDX_FRQHZ3()

- [ATDX_FRQHZ\(\)](#)
- [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 boards
-

■ 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*.

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

■ Cautions

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

■ 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>
```

return percentage of time SIT tone was out of bounds — ATDX_FRQOUT()

```
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:
            break;
        .
        .
    }
}
```

■ See Also

- [dx_dial\(\)](#)
- [dx_dialtpt\(\)](#)
- [ATDX_CPTERM\(\)](#)
- [DX_CAP](#) data structure
- call progress analysis topic in the *Voice API Programming Guide*

ATDX_FWVER() — return the voice firmware version number

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 boards

■ Description

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

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

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

TTTT | MMMM | mmmmmmmmm | AAAAAAAAA | aaaaaaaaa

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

Letter	Description
T	Type of Release. Decimal values have the following meanings (example: 0010 for Alpha release): <ul style="list-style-type: none">• 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”

return the voice firmware version number — ATDX_FWVER()

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

The following is an example on Linux.

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

main()
{
    int bddev;
    long fwver;
    .
    .

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

    /* Display Firmware version number */
    if ((fwver = ATDX_FWVER(bddev)) == AT_FAILURE) {
        /* Process error */
    }
    printf("Firmware version %ld\n",fwver);
    .
    .
}
```

The following is an example on Windows.

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

void GetFwVersion(char *, long);

main()
{
    int bddev;
    char *bdname, FWVersion[50];
    long fwver;
```

ATDX_FWVER() — *return the voice firmware version number*

```
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
 */
GetFwVersion(FWVersion, fwver);

printf("%s\n", FWVersion);

} /* end main */
```

■ **See Also**

None.

ATDX_HOOKST()

- Name:** long ATDX_HOOKST(chdev)
- Inputs:** int chdev • valid channel device handle
- Returns:** current hook state of channel if successful
AT_FAILURE if error
- Includes:** srllib.h
dxxplib.h
- Category:** Extended Attribute
- Mode:** synchronous
- Platform:** Springware boards
-

■ Description

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

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

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

main()
{
    int chdev;
    long hookst;
    /* Open the channel device */
    if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
```

***ATDX_HOOKST()* — return the current hook-switch state**

```
        /* 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 */
    }
    .
    .
}
```

■ See Also

- [dx_sethook\(\)](#)
- [DX_CST](#) structure
- [dx_setevtmsk\(\)](#) 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*

return the current activity on the channel — ATDX_LINEST()

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
dxxlib.h

Category: Extended Attribute

Mode: synchronous

Platform: Springware boards

■ Description

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

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

Possible return values are the following:

RLS_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

```
#include <srllib.h>
#include <dxxlib.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 • valid channel device handle
- Returns:** duration of longer silence if successful
AT_FAILURE if error
- Includes:** srllib.h
dxxplib.h
- Category:** Extended Attribute
- Mode:** synchronous
- Platform:** Springware boards
-

■ Description

The **ATDX_LONGLOW()** function returns duration of longer 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_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
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

```
/* Call progress analysis with user-specified parameters */
#include <stdio.h>
#include <srllib.h>
#include <dxxplib.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);
        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

- [dx_dial\(\)](#)
- [dx_dialtpt\(\)](#)
- [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*

return the physical board address — ATDX_PHYADDR()

ATDX_PHYADDR()

Name: long ATDX_PHYADDR(bddev)

Inputs: int bddev • valid board device handle

Returns: physical address of board if successful
AT_FAILURE if error

Includes: srllib.h
dxxplib.h

Category: Extended Attribute

Mode: synchronous

Platform: Springware boards

■ Description

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

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

■ Cautions

None.

■ Errors

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

■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <dxxplib.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*

```
    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
dxxlib.h
- Category:** Extended Attribute
- Mode:** synchronous
- Platform:** Springware boards
-

■ 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.

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**.

ATDX_SHORTLOW() — return duration of shorter silence detected

■ 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

- [dx_dial\(\)](#)
- [dx_dialtpt\(\)](#)
- [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
dxxplib.h

Category: Extended Attribute

Mode: synchronous

Platform: Springware boards

■ 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.

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

```
/* Call progress analysis with user-specified parameters */
#include <stdio.h>
#include <srllib.h>
#include <dxxplib.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);
        printf("\nDuration of non-silence - %ld ms",ATDX_SIZEHI(chdev)*10);
        break;
    case CR_CEPT:     /* Operator Intercept detected */
        printf("\nFrequency detected - %ld Hz",ATDX_FRQHZ(chdev));
        printf("\n%% of Frequency out of bounds - %ld Hz",ATDX_FRQOUT(chdev));
        break;
    case CR_BUSY:
        .
        .
}
}
```

■ See Also

- [dx_dial\(\)](#)
- [dx_dialtpt\(\)](#)
- [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*

return the current state of the channel — ATDX_STATE()

ATDX_STATE()

Name: long ATDX_STATE(chdev)

Inputs: int chdev • valid channel device handle

Returns: current state of channel if successful
AT_FAILURE if error

Includes: srllib.h
dxxlib.h

Category: Extended Attribute

Mode: synchronous

Platform: HMP Software, Springware boards

■ Description

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

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

Possible return values are the following:

CS_DIAL	Dial state
CS_CALL	Call state
CS_GTDIG	Get Digit state
CS_HOOK	Hook state
CS_IDLE	Idle state
CS_PLAY	Play state
CS_RECD	Record state
CS_STOPD	Stopped state

***ATDX_STATE()* — return the current state of the channel**

CS_TONE
Playing tone 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**

```
#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
dxxplib.h

Category: Extended Attribute

Mode: synchronous

Platform: HMP Software, Springware boards

■ 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 HMP Software, possible return values are the following:

TM_DIGIT	Specific digit received
TM_EOD	End of data reached (on playback, receive)
TM_ERROR	I/O device error
TM_IDDTIME	Inter-digit delay
TM_MAXDTMF	Maximum DTMF count
TM_MAXSIL	Maximum period of silence
TM_MAXTIME	Maximum function time exceeded
TM_NORMTERM	Normal termination (for dx_dial())
TM_TONE	Tone-on/off event

ATDX_TERMMSK() — return the reason for the last I/O function termination

TM_USRSTOP	Function stopped by user
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_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.
- On HMP Software, when both `DX_MAXDTMF` and `DX_DIGMASK` termination conditions are specified in the `DV_TPT` structure, and both conditions are satisfied, the `ATDX_TERMMSK()` function will return the `TM_MAXDTMF` termination event only.

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

This behavior differs from Springware boards, where both `TM_MAXDTMF` and `TM_DIGIT`

return the reason for the last I/O function termination — ATDX_TERMMSK()

will be returned when both DX_MAXDTMF and DX_DIGMASK termination conditions are specified in the DV_TPT structure and both are satisfied by the user input.

■ Errors

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

■ Example

```
#include <stdio.h>
#include <fcntl.h>
#include <srllib.h>
#include <dxxplib.h>

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

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

    /* Record a voice file. Terminate on receiving a digit, silence, loop
     * current drop, max time, or reaching a byte count of 50000 bytes.
     */
    /* set up DX_IOTT */
    iott.io_type = IO_DEV|IO_EOT;
    iott.io_bufp = 0;
    iott.io_offset = 0;
    iott.io_length = 50000;

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

    /* set up DV_TPTs for the required terminating conditions */
    dx_clrtpt(tpt,4);
    tpt[0].tp_type = IO_CONT;
    tpt[0].tp_termno = DX_MAXDTMF; /* Maximum digits */
    tpt[0].tp_length = 1; /* terminate on the first digit */
    tpt[0].tp_flags = TF_MAXDTMF; /* Use the default flags */
    tpt[1].tp_type = IO_CONT;
    tpt[1].tp_termno = DX_MAXTIME; /* Maximum time */
    tpt[1].tp_length = 100; /* terminate after 10 secs */
    tpt[1].tp_flags = TF_MAXTIME; /* Use the default flags */
    tpt[2].tp_type = IO_CONT;
    tpt[2].tp_termno = DX_MAXSIL; /* Maximum Silence */
    tpt[2].tp_length = 30; /* terminate on 3 sec silence */
    tpt[2].tp_flags = TF_MAXSIL; /* Use the default flags */
    tpt[3].tp_type = IO_EOT; /* last entry in the table */
    tpt[3].tp_termno = DX_LCOFF; /* terminate on loop current drop */
    tpt[3].tp_length = 10; /* terminate on 1 sec silence */
    tpt[3].tp_flags = TF_LCOFF; /* Use the default flags */

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

ATDX_TERMMSK() — return the reason for the last I/O function termination

```
/* 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 *Dialogic® Standard Runtime Library API Programming Guide* and *Dialogic® Standard Runtime Library API Library Reference*)
- [ATEC_TERMMSK\(\)](#) in the *Dialogic® Continuous Speech Processing API Library Reference*

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

ATDX_TONEID()

Name: long ATDX_TONEID(chdev)

Inputs: int chdev • valid channel device handle

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

Includes: srllib.h
dxxplib.h

Category: Extended Attribute

Mode: synchronous

Platform: HMP Software, Springware boards

■ 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.

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

#define TID_1 101

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

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

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

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

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

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

/*
 * Set up the Terminating Conditions
 */
tpt[0].tp_type = IO_CONT;
tpt[0].tp_termno = DX_TONE;
tpt[0].tp_length = TID_1;
tpt[0].tp_flags = TF_TONE;
tpt[0].tp_data = DX_TONEON;
tpt[1].tp_type = IO_CONT;
tpt[1].tp_termno = DX_TONE;
tpt[1].tp_length = TID_1;
tpt[1].tp_flags = TF_TONE;
tpt[1].tp_data = DX_TONEOFF;
tpt[2].tp_type = IO_EOT;
tpt[2].tp_termno = DX_MAXTIME;
tpt[2].tp_length = 6000;
tpt[2].tp_flags = TF_MAXTIME;

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

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

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

ATDX_TRCOUNT()

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: HMP Software, Springware boards

■ Description

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

Parameter	Description
-----------	-------------

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

■ Cautions

None.

■ Errors

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

■ Example

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

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

    /* Open the channel device */
    if ((chdev = dx_open("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 */
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

dx_addspddig()

Name: int dx_addspddig(chdev, digit, adjval)

Inputs: int chdev • valid channel device handle
 char digit • DTMF digit
 short adjval • speed adjustment value

Returns: 0 if success
 -1 if failure

Includes: srllib.h
 dxxxlib.h

Category: Speed and Volume

Mode: synchronous

Platform: HMP Software, Springware boards

■ Description

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.

Note: On HMP Software, before you can use the speed control feature, you must enable this feature in the [decoder] section of the CONFIG file. For more information, see the Configuration Guide applicable to your release or product.

For more information about speed and volume control as well as speed and volume modification tables, see the *Dialogic® 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()

Parameter	Description
digit	specifies a DTMF digit (0-9, *,#) that will modify speed by the amount specified in adjval
adjval	<p>specifies a speed adjustment value to take effect whenever the digit specified in digit occurs:</p> <p>On HMP Software, valid values are:</p> <ul style="list-style-type: none"> • 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% <p>On Springware boards, valid values are:</p> <ul style="list-style-type: none"> • 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**

- Speed control is not supported for all voice coders. For more information on supported coders, see the speed control topic in the *Dialogic® Voice API Programming Guide*.
- On HMP Software, digits that are used for play adjustment may also be used as a terminating condition. If a digit is defined as both, then both actions are applied upon detection of that digit.
- On 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.
- 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 Dialogic® 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_BADPROD	Function not supported on this board
EDX_SVADJBLK	Invalid number of play adjustment blocks
EDX_SYSTEM	Error from operating system

■ Example

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

/*
 * Global Variables
 */

main()
{
    int dxxxxdev;

    /*
     * Open the Voice Channel Device and Enable a Handler
     */
    if ( ( dxxxxdev = dx_open( "dxxxxB1C1", NULL) ) == -1 ) {
        perror( "dxxxxB1C1" );
        exit( 1 );
    }

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

■ 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 *Dialogic® Voice API Programming Guide*.

Parameter	Description
chdev	specifies the valid channel device handle obtained when the channel was opened using dx_open()
digit	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 dx_getdig() (they can be used in the same way as DTMF digits, for example). If you do not specify a digit, the tone will be indicated by a DE_TONEON event or DE_TONEOFF event. Note: User-defined tones that are associated with an optional digit have digit reporting enabled by default. The user-defined tones digit reporting can be turned off by using dx_setevtmsk() with DM_DIGOFF mask. To reactivate digit reporting, use dx_setevtmsk() with DM_DIGITS mask.

Parameter	Description
digtype	<p>specifies the type of digit the channel will detect</p> <p>On HMP Software, the valid value is:</p> <ul style="list-style-type: none"> • DG_USER1 <p>On Springware boards, valid values are:</p> <ul style="list-style-type: none"> • DG_USER1 • DG_USER2 • DG_USER3 • DG_USER4 • DG_USER5 <p>Up to twenty digits can be associated with each of these digit types.</p> <p><i>Note:</i> 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().</p>

■ 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 *Dialogic® 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 Dialogic® 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_BADPARAM	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_MAXTMPLT	Maximum number of user-defined tones for the board
EDX_SYSTEM	Error from operating system
EDX_TONEID	Invalid tone template ID

■ **Example**

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

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

main()
{
    int dxxxxdev;

    /*
     * Open the Voice Channel Device and Enable a Handler
     */
    if ( ( dxxxxdev = dx_open( "dxxxxB1C1", NULL ) ) == -1 ) {
        perror( "dxxxxB1C1" );
        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( dxxxxdev, NULL, 0 ) == -1 ) {
        printf( "Unable to Bind the Tone %d\n", TID_1 );
        printf( "Lasterror = %d Err Msg = %s\n",
            ATDV_LASTERR( dxxxxdev ), ATDV_ERRMSGP( dxxxxdev ) );
        dx_close( dxxxxdev );
        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 msec and off
 * 990-1010 msec and a cadence of 3.
 */
if ( dx_blddtcad( TID_2, 1000, 50, 500, 25, 20, 1, 100, 1, 3 ) == -1 ) {
    printf("Unable to build a Dual Tone Cadence Template\n" );
}

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

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

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

/*
 * Describe a Single Tone Frequency Tone of 950-1050 Hz.
 * On between 190-210 msec and off 990-1010 msec 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( dxdev ) != 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 *Dialogic® Voice API Programming Guide*
- [dx_getevt\(\)](#)
- [DX_CST](#) data structure
- [sr_getevtdatap\(\)](#) in the *Dialogic® 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`
`dxxlib.h`

Category: Speed and Volume

Mode: synchronous

Platform: HMP Software, Springware boards

■ 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, see the *Dialogic® Voice API Programming Guide*. For information about speed and volume data structures, see the `DX_SVMT` and the `DX_SVCB` data structures.

Parameter	Description
<code>chdev</code>	specifies the valid channel device handle obtained when the channel was opened using <code>dx_open()</code>
<code>digit</code>	specifies a DTMF digit (0-9, *, #) that will modify volume by the amount specified in <code>adjval</code>

dx_addvoldig() — set a DTMF digit to adjust volume

Parameter	Description
adjval	<p>specifies a volume adjustment value to take effect whenever the digit specified in digit occurs</p> <p>On HMP Software, valid values are:</p> <ul style="list-style-type: none">• 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) <p>On Springware boards, valid values are:</p> <ul style="list-style-type: none">• 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 HMP Software, digits that are used for play adjustment may also be used as a terminating condition. If a digit is defined as both, then both actions are applied upon detection of that digit.
- On 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 Dialogic® 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_BADPROD

Function not supported on this board

EDX_SVADJBLKS

Invalid number of play adjustment blocks

EDX_SYSTEM

Error from operating system

■ Example

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

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

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

    /*
     * Add a Speed Adjustment Condition - decrease the
     * playback volume by 2dB whenever DTMF key 2 is pressed.      */
    if ( dx_addvoldig( dxxxdev, '2', SV_SUB2DB ) == -1 ) {
        printf( "Unable to Add a Volume Adjustment" );
        printf( " Condition\n" );
        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_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: HMP Software, Springware boards

■ 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.

Note: On HMP Software, before you can use the speed control feature, you must enable this feature in the [decoder] section of the CONFIG file. For more information, see the Configuration Guide applicable to your release or product.

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 *Dialogic® 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.

dx_adjsv() — *adjust speed or volume immediately*

Parameter	Description
chdev	specifies the valid channel device handle obtained when the channel was opened using dx_open()
tabletype	specifies whether to modify the playback using a value from the speed or the volume modification table <ul style="list-style-type: none">SV_SPEEDTBL – use the speed modification tableSV_VOLUMETBL – use the volume modification table
action	specifies the type of adjustment to make. Set to one of the following: <ul style="list-style-type: none">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 parameterSV_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. <ul style="list-style-type: none">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 Dialogic® 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_BADPROD
Function not supported on this board

EDX_SYSTEM
Error from operating system

■ Example

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

main()
{
    int dxxxdev;

    /*
     * Open the Voice Channel Device and Enable a Handler
     */
    if ( ( dxxxdev = dx_open( "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 *Dialogic® Voice API Programming Guide*

dx_adjsv() — *adjust speed or volume immediately*

- [DX_SVMT](#) data structure

`dx_blddt()`

Name: `int dx_blddt(tid, freq1, fq1dev, freq2, fq2dev, mode)`

Inputs:

<code>unsigned int tid</code>	• tone ID to assign
<code>unsigned int freq1</code>	• frequency 1 in Hz
<code>unsigned int fq1dev</code>	• frequency 1 deviation in Hz
<code>unsigned int freq2</code>	• frequency 2 in Hz
<code>unsigned int fq2dev</code>	• frequency 2 deviation in Hz
<code>unsigned int mode</code>	• leading or trailing edge

Returns: 0 if success
-1 if failure

Includes: `srllib.h`
`dxxlib.h`

Category: Global Tone Detection

Mode: synchronous

Platform: HMP Software, Springware boards

■ 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 *Dialogic® Voice API Programming Guide*.

Parameter	Description
tid	specifies a unique identifier for the tone. See Cautions for more information about the tone ID.
freq1	specifies the first frequency (in Hz) for the tone
frq1dev	specifies the allowable deviation (in Hz) for the first frequency
freq2	specifies the second frequency (in Hz) for the tone
frq2dev	specifies the allowable deviation (in Hz) for the second frequency
mode	specifies whether tone detection notification will occur on the leading or trailing edge of the tone. Set to one of the following: <ul style="list-style-type: none"> • <code>TN_LEADING</code> • <code>TN_TRAILING</code>

***dx_blddt()* — define a user-defined dual-frequency tone**

■ 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.
- On Windows[®], do not use tone IDs 261, 262 and 263; they are reserved for library use.
- When using this function in a multi-threaded application, use critical sections or a semaphore around the function call to ensure a thread-safe application. Failure to do so will result in “Bad Tone Template ID” errors.

■ Errors

If this function returns -1 to indicate failure, call the Dialogic[®] 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

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

#define TID_1 101

main()
{
    int dxxxdev;

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

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

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

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

define a user-defined dual-frequency tone — dx_blddt()

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

■ **See Also**

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

dx_blddtcad() — *define a user-defined dual frequency cadenced tone*

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
dxxxlib.h

Category: Global Tone Detection

Mode: synchronous

Platform: HMP Software, Springware boards

■ 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 *Dialogic® Voice API Programming Guide*.

Parameter	Description
tid	specifies a unique identifier for the tone. See Cautions for more information on the tone ID.
freq1	specifies the first frequency (in Hz) for the tone
frq1dev	specifies the allowable deviation (in Hz) for the first frequency
freq2	specifies the second frequency (in Hz) for the tone

define a user-defined dual frequency cadenced tone — `dx_blddtcad()`

Parameter	Description
<code>frq2dev</code>	specifies the allowable deviation (in Hz) for the second frequency
<code>ontime</code>	specifies the length of time for which the cadence is on (in 10 msec units)
<code>ontdev</code>	specifies the allowable deviation for on time (in 10 msec units)
<code>offtime</code>	specifies the length of time for which the cadence is off (in 10 msec units)
<code>offtdev</code>	specifies the allowable deviation for off time (in 10 msec units)
<code>repcnt</code>	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. `dx_blddtcad()` will replace the previous user-defined tone definition.
- On Windows[®], do not use tone IDs 261, 262 and 263; they are reserved for library use.
- When using this function in a multi-threaded application, use critical sections or a semaphore around the function call to ensure a thread-safe application. Failure to do so will result in “Bad Tone Template ID” errors.

■ Errors

If this function returns -1 to indicate failure, call the Dialogic[®] 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

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

#define TID_2 102

main()
{
    int dxxxdev;

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

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

dx_blddtcad() — *define a user-defined dual frequency cadenced tone*

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

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

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

■ See Also

- global tone detection topic in *Dialogic® Voice API Programming Guide*
- [dx_bldst\(\)](#)
- [dx_blddt\(\)](#)
- [dx_bldstcad\(\)](#)
- [dx_addtone\(\)](#)
- [dx_distone\(\)](#)
- [dx_enbtone\(\)](#)

`dx_bldst()`

Name: `int dx_bldst(tid, freq, fqdev, mode)`

Inputs:

<code>unsigned int tid</code>	• tone ID to assign
<code>unsigned int freq</code>	• frequency in Hz
<code>unsigned int fqdev</code>	• frequency deviation in Hz
<code>unsigned int mode</code>	• leading or trailing edge

Returns: 0 if success
-1 if failure

Includes: `srllib.h`
`dxxlib.h`

Category: Global Tone Detection

Mode: synchronous

Platform: HMP Software, Springware boards

■ 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 *Dialogic® Voice API Programming Guide*.

Parameter	Description
tid	specifies a unique identifier for the tone. See Cautions for more information about the tone ID.
freq	specifies the frequency (in Hz) for the tone
fqdev	specifies the allowable deviation (in Hz) for the frequency
mode	specifies whether detection is on the leading or trailing edge of the tone. Set to one of the following: <ul style="list-style-type: none">• <code>TN_LEADING</code>• <code>TN_TRAILING</code>

■ Cautions

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

***dx_bldst()* — define a user-defined single-frequency tone**

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

■ Errors

If this function returns -1 to indicate failure, call the Dialogic® 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

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

#define TID_3 103

main()
{
    int dxxxxdev;

    /*
     * Open the Voice Channel Device and Enable a Handler
     */
    if ( ( dxxxxdev = dx_open( "dxxxxB1C1", 0 ) ) == -1 ) {
        perror( "dxxxxB1C1" );
        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( dxxxxdev ) != 0 ) {
        perror( "close" );
    }

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

■ See Also

- global tone detection topic in *Dialogic® Voice API Programming Guide*
- [dx_blddtcad\(\)](#)

define a user-defined single-frequency tone — dx_bldst()

- [dx_blddt\(\)](#)
- [dx_bldstcad\(\)](#)
- [dx_addtone\(\)](#)
- [dx_distone\(\)](#)
- [dx_enbtone\(\)](#)

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

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: HMP Software, Springware boards

■ 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 *Dialogic® Voice API Programming Guide*.

Parameter	Description
tid	specifies a unique identifier for the tone. See Cautions for more information about the tone ID.
freq	specifies the frequency (in Hz) for the tone
frqdev	specifies the allowable deviation (in Hz) for the frequency
ontime	specifies the length of time for which the cadence is on (in 10 msec units)
ontdev	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)

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

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

■ Cautions

- Only one tone per application may be defined at any time. `dx_bldstcad()` will replace the previous user-defined tone definition.
- On Springware boards, using `dx_bldstcad()` to define two different user-defined tones with the same frequency and different cadence times may result in the board erroneously reporting `CON_CAD` instead of `CR_NOANS`.
- On Windows[®], do not use tone IDs 261, 262 and 263; they are reserved for library use.
- When using this function in a multi-threaded application, use critical sections or a semaphore around the function call to ensure a thread-safe application. Failure to do so will result in “Bad Tone Template ID” errors.

■ Errors

If this function returns -1 to indicate failure, call the Dialogic[®] 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

```
#include <stdio.h>
#include <srllib.h>
#include <dxxlib.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 msec and off 990-1010 msec 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" );
    }
}
```

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

```
/*
 * 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 *Dialogic® Voice API Programming Guide*
- [dx_blddtcad\(\)](#)
- [dx_blddt\(\)](#)
- [dx_bldst\(\)](#)
- [dx_addtone\(\)](#)
- [dx_distone\(\)](#)
- [dx_enbtone\(\)](#)

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
dxxlib.h

Category: Global Tone Generation

Mode: synchronous

Platform: HMP Software, Springware boards

■ 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 *Dialogic® Voice API Programming Guide*.

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

dx_bldtngen() — define a tone for generation

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**

```
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
main()
{
    TN_GEN    tngen;
    int       dxxxdev;
    /*
     * Open the Voice Channel Device and Enable a Handler
     */
    if ( ( dxxxdev = dx_open( "dxxxBlC1", 0 ) ) == -1 ) {
        perror( "dxxxBlC1" );
        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 msec.
     */
    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 *Dialogic® Voice API Programming Guide*

define a tone for generation — dx_bldtngen()

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 boards

■ 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_deltone()** 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*.

change the duration definition for a tone — `dx_chgdur()`

Parameter	Description
tonetype	specifies the identifier of the tone whose definition is to be modified. It may be one of the following: <ul style="list-style-type: none">• 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) (Windows only)• 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)
ontime	specifies the length of time that the tone is on (10 msec units)
ondev	specifies the maximum permissible deviation from ontime (10 msec units)
offtime	specifies the length of time that the tone is off (10 msec units)
offdev	specifies the maximum permissible deviation from offtime (10 msec units)

■ Cautions

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

■ Errors

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

■ Example

```
#include <stdio.h>
#include <srllib.h>
#include <dxxlib.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 */
    }
}
```

***dx_chgdur()* — change the duration definition for a tone**

```
/*
 * Delete any previous tones
 */
if ( dx_deltone(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_deltone\(\)](#)
- [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 boards

■ 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_deltone()** 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*.

change the frequency definition for a tone — `dx_chgfreq()`

Parameter	Description
tonetype	specifies the identifier of the tone whose definition is to be modified. It may be one of the following: <ul style="list-style-type: none">• 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) (Windows only)• 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)
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

```
#include <stdio.h>
#include <srllib.h>
#include <dxxplib.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 */
    }
}
```

***dx_chgfreq()* — change the frequency definition for a tone**

```
/*
 * Delete any previous tones
 */
if ( dx_deltone(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 frequency definition for a tone — dx_chgfreq()

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

dx_close( ddd );
}
```

■ See Also

- [dx_chgdur\(\)](#)
- [dx_chgrept\(\)](#)
- [dx_deltone\(\)](#)
- [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 boards

■ 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.

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_deltone()** followed by **dx_initcallp()**.

Parameter	Description
tonetype	specifies the identifier of the tone whose definition is to be modified. It may be one of the following: <ul style="list-style-type: none">• 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) (Windows only)• 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_deltone()` is called on that channel followed by `dx_initcallp()`.

■ Errors

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

■ Example

```
#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_deltone(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 */
    }
}
```

***dx_chgrepcnt()* — change the repetition definition for a tone**

```
/*
 * 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_deltone\(\)](#)
- [dx_initcallp\(\)](#)

`dx_close()`

Name: `int dx_close(dev)`

Inputs: `int dev` • valid channel or board device handle

Returns: 0 if successful
-1 if error

Includes: `srllib.h`
`dxxlib.h`

Category: Device Management

Mode: synchronous

Platform: HMP Software, Springware boards

■ Description

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

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

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

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

■ Cautions

- Once a device is closed, a process can no longer act on that device using that device handle.
- Other handles for that device that exist in the same process or other processes will still be valid.
- The only process affected by `dx_close()` is the process that called the function.
- Do not use the operating system `close()` command to close a voice device; unpredictable results will occur.
- The `dx_close()` function discards any outstanding events on that handle.
- If you close a device via `dx_close()` after modifying volume table values using `dx_getsvmt()`, the `dx_getcursv()` function may return incorrect volume settings for the device. This is because the next `dx_open()` resets the volume tables to their default values.

dx_close() — *close a channel or board device handle*

■ **Errors**

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

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

EBADF
Invalid file descriptor

EINTR
A signal was caught

EINVAL
Invalid argument

■ **Example**

This example illustrates how to close a channel device handle.

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

main()
{
    int chdev;          /* channel descriptor */
    .
    .
    .

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

    /* Close channel */
    if (dx_close(chdev) == -1) {
        /* process error */
    }
}
```

■ **See Also**

- [dx_open\(\)](#)

`dx_CloseStreamBuffer()`

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: HMP Software

■ 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.

Parameter	Description
<code>hBuffer</code>	specifies the stream buffer handle obtained from <code>dx_OpenStreamBuffer()</code>

■ 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 Dialogic® 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 <dxxxlib.h>

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

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

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

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

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

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

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

■ See Also

- [dx_OpenStreamBuffer\(\)](#)
- [dx_GetStreamInfo\(\)](#)

dx_clrcap()

Name: void dx_clrcap(capp)

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

Returns: none

Includes: srllib.h
dxxxlib.h

Category: Structure Clearance

Mode: synchronous

Platform: HMP Software, Springware boards

■ 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
capp	pointer to call progress analysis parameter data structure, DX_CAP. For more information on this structure, see DX_CAP , on page 490.

■ 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

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

main()
{
    DX_CAP cap;
    int chdev;

    /* open the channel using dx_open */
    if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
        /* process error */
    }
    .
    .
    /* set call progress analysis parameters before doing call progress analysis */
    dx_clrcap(&cap);
    cap.ca_nbrdna = 5; /* 5 rings before no answer */
}
```

dx_clracap() — clear all fields in a *DX_CAP* structure

```
.  
. /* continue with call progress analysis */  
. .  
}
```

■ **See Also**

- [dx_dial\(\)](#)
- [DX_CAP](#) data structure
- call progress analysis topic in the *Dialogic® Voice API Programming Guide*

`dx_clrdigbuf()`

Name: `int dx_clrdigbuf(chdev)`

Inputs: `int chdev` • valid channel device handle

Returns: 0 if success
-1 if failure

Includes: `srllib.h`
`dxxlib.h`

Category: Configuration

Mode: synchronous

Platform: HMP Software, Springware boards

■ Description

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

Parameter	Description
<code>chdev</code>	specifies the valid channel device handle obtained when the channel was opened using <code>dx_open()</code>

■ Cautions

- The function will fail and return -1 if the channel device handle is invalid or the channel is busy.
- On HMP Software, digits will not always be cleared by the time this function returns, because processing may continue on the board even after the function returns. For this reason, careful consideration should be given when using this function before or during a section where digit detection or digit termination is required; the digit may be cleared only after the function has returned and possibly during the next function call.

■ Errors

If the function returns -1, use the Dialogic® 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_SYSTEM`
Error from operating system

***dx_clrdigbuf()* — clear all digits in the firmware digit buffer**

■ 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()**.

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

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

    /* Clear digit buffer */
    if (dx_clrdigbuf(chdev) == -1) {
        /* process error*/
    }
    .
    .
}
```

■ See Also

None.

dx_clrsvcond()

- Name:** int dx_clrsvcond(chdev)
- Inputs:** int chdev • valid channel device handle
- Returns:** 0 if success
-1 if failure
- Includes:** srllib.h
dxxplib.h
- Category:** Speed and Volume
- Mode:** synchronous
- Platform:** HMP Software, Springware boards
-

■ 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()**.

Note: On HMP Software, before you can use the speed control feature, you must enable this feature in the [decoder] section of the CONFIG file. For more information, see the Configuration Guide applicable to your release or product.

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

■ Cautions

None.

■ Errors

If the function returns -1, use the Dialogic® 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_BADPROD
Function not supported on this board

***dx_clrsvcond()* — clear all speed or volume adjustment conditions**

EDX_SYSTEM

Error from operating system

■ Example

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

main()
{
    int dxxxdev;

    /*
     * Open the Voice Channel Device and Enable a Handler
     */
    if ( ( dxxxdev = dx_open( "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" );
        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_addspddig\(\)](#)
- [dx_addvoldig\(\)](#)
- speed and volume modification tables in *Dialogic® Voice API Programming Guide*
- [DX_SVCB](#) data structure

dx_clrtp()

Name: int dx_clrtp(tp, size)

Inputs: DV_TPT *tp • 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: HMP Software, Springware boards

■ Description

The **dx_clrtp()** 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
tp	points to the first DV_TPT structure to be cleared
size	indicates the number of DV_TPT structures to clear. If size is set to 0, the function will return a 0 to indicate success. For more information on this structure, see DV_TPT , on page 481.

- 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_clrtp()**, 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_clrtp()** 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_clrtp()** only after the links have been set up properly, as illustrated in the Example.

***dx_clrtp()* — 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

```
#include <srllib.h>
#include <dxxxxlib.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_nexttp = &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_clrtp(&tpt1[0],4);
    /* now set up the DV_TPT structures for the next play */
    .
    .
}
```

■ See Also

- [DV_TPT data structure](#)

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

dx_createtone()

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

Inputs:

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

Returns: 0 if successful
-1 if failure

Includes: srllib.h
dxxlib.h

Category: Call Progress Analysis

Mode: Asynchronous or synchronous

Platform: HMP Software

■ 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.

Before creating a new tone definition with **dx_createtone()**, first use **dx_querytone()** to get tone information for the tone ID, then use **dx_deletetone()** to delete that same tone ID. Only tones listed in the **toneid** parameter description are supported for this function. For more information on modifying call progress analysis tone definitions, see the *Dialogic® Voice API Programming Guide*.

When running in asynchronous mode, this 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. The **TONE_DATA** structure should remain in scope until the application receives these events.

By default, this function runs in synchronous mode and returns 0 to indicate completion.

Parameter	Description
brdhdl	specifies a valid board device handle (not a virtual board device) of the format brdBn obtained by a call to dx_open() . To get the board name, use the SRLGetPhysicalBoardName() function. This function and other device mapper functions return information about the structure of the system. For more information, see the <i>Dialogic® Standard Runtime Library API Library Reference</i> .

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

Parameter	Description
toneid	<p>specifies the tone ID of the call progress tone whose definition needs to be modified. Valid values are:</p> <ul style="list-style-type: none">• TID_BUSY1• TID_BUSY2• TID_DIAL_INTL• TID_DIAL_LCL• TID_DISCONNECT• TID_FAX1• TID_FAX2• TID_RNGBK1• TID_RNGBK2• TID_SIT_NC• TID_SIT_IC• TID_SIT_VC• TID_SIT_RO <p><i>Note:</i> The following tone IDs are not supported by this function: TID_SIT_ANY, TID_SIT_NO_CIRCUIT_INTERLATA, TID_SIT_REORDER_TONE_INTERLATA, and TID_SIT_INEFFECTIVE_OTHER.</p>
tonedata	<p>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</p>
mode	<p>specifies the mode in which the function will run. Valid values are:</p> <ul style="list-style-type: none">• EV_ASYNC – asynchronous mode• EV_SYNC – synchronous mode (default)

■ Cautions

- Only the default call progress tones listed in the **toneid** parameter description are supported for this function. The following tone IDs are not supported by this function: TID_SIT_ANY, TID_SIT_NO_CIRCUIT_INTERLATA, TID_SIT_REORDER_TONE_INTERLATA, and TID_SIT_INEFFECTIVE_OTHER.
- If you call **dx_createtone()** prior to calling **dx_deletetone()**, then **dx_createtone()** will fail with an error EDX_TNQUERYDELETE.
- To modify a default tone definition, use the three functions **dx_querytone()**, **dx_deletetone()**, and **dx_createtone()** in this order, for one tone at a time.
- When **dx_createtone()** is issued on a board device in asynchronous mode, and the function is immediately followed by another similar call prior to completion of the previous call on the same device, the subsequent call will fail with device busy.

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

■ Errors

If the function returns -1, use the Dialogic® 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_SYSTEM	error from operating system
EDX_TNPARAM	invalid tone template parameter
EDX_TNQUERYDELETE	tone not queried or deleted prior to create

■ Example

```
#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);
    }

    while (1) {
        if (sr_waitevt(2000) < 0)
            break;
        long evttype = sr_getevttype(0);
        if (evttype == TDX_QUERYTONE)
            printf("TDX_QUERYTONE Event received \n");
        elseif (evttype == TDX_QUERYTONE_FAIL)
            printf("TDX_QUERYTONE_FAIL event received \n");
        else
            printf(" Unknown event received 0x%x \n", evttype);
        break;
    }

    /* 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");
    }
}
```

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

```
/* Perform system error processing */
exit(1);
}
while (1) {
if (sr_waitevt(2000) < 0)
break;
long evttype = sr_getevttype(0);
if (evttype == TDX_DELETETONE)
printf("TDX_DELETETONE Event received \n");
elseif (evttype == TDX_DELETETONE_FAIL)
printf("TDX_DELETETONE_FAIL event received \n");
else
printf(" Unknown event received 0x%x \n", evttype);
break;
}

/* Change call progress default Busy tone */

tonedata.toneseg[0].structver = 0;
tonedata.toneseg[0].numofseg = 1; /* Single segment tone */
tonedata.toneseg[0].tn_rep_cnt = 4;

toneinfo.toneseg[0].structver = 0;
toneinfo.toneseg[0].tn_dflag = 1; /* Dual tone */
toneinfo.toneseg[0].tn1_min = 0; /* Min. Frequency for Tone 1 (in Hz) */
toneinfo.toneseg[0].tn1_max = 450; /* Max. Frequency for Tone 1 (in Hz) */
toneinfo.toneseg[0].tn2_min = 0; /* Min. Frequency for Tone 2 (in Hz) */
toneinfo.toneseg[0].tn2_max = 150; /* Max. Frequency for Tone 2 (in Hz) */
toneinfo.toneseg[0].tn_twinmin = 0;
toneinfo.toneseg[0].tn_twinmax = 0;

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

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

while (1) {
if (sr_waitevt(2000) < 0)
break;
long evttype = sr_getevttype(0);
if (evttype == TDX_CREATETONE)
printf("TDX_CREATETONE Event received \n");
elseif (evttype == TDX_CREATETONE_FAIL)
printf("TDX_CREATETONE_FAIL event received \n");
else
printf(" Unknown event received 0x%x \n", evttype);
break;
}
}
```

■ See Also

- [dx_deletetone\(\)](#)
- [dx_querytone\(\)](#)

`dx_deletetone()`

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

Inputs:

<code>int brdhdl</code>	• a valid board device handle
<code>int toneid</code>	• tone ID of the call progress tone
<code>unsigned short mode</code>	• mode

Returns: 0 if successful
-1 if failure

Includes: `srllib.h`
`dxxlib.h`

Category: Call Progress Analysis

Mode: asynchronous or synchronous

Platform: HMP Software

■ Description

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

Before creating a new tone definition with `dx_createtone()`, first use `dx_querytone()` to get tone information for the tone ID, then use `dx_deletetone()` to delete that same tone ID. Only tones listed in the `toneid` parameter description are supported for this function. For more information on modifying call progress analysis tone definitions, see the *Dialogic® Voice API Programming Guide*.

When running in asynchronous mode, 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.

By default, this function runs in synchronous mode and returns 0 to indicate completion.

Parameter	Description
<code>brdhdl</code>	specifies a valid board device handle (not a virtual board device) of the format <code>brdBn</code> obtained by a call to <code>dx_open()</code> . To get the board name, use the <code>SRLGetPhysicalBoardName()</code> function. This function and other device mapper functions return information about the structure of the system. For more information, see the <i>Dialogic® Standard Runtime Library API Library Reference</i> .

dx_deletetone() — delete a specific call progress tone

Parameter	Description
toneid	specifies the tone ID of the call progress tone. Valid values are: <ul style="list-style-type: none">• TID_BUSY1• TID_BUSY2• TID_DIAL_INTL• TID_DIAL_LCL• TID_DISCONNECT• TID_FAX1• TID_FAX2• TID_RNGBK1• TID_RNGBK2• TID_SIT_NC• TID_SIT_IC• TID_SIT_VC• TID_SIT_RO <p><i>Note:</i> The following tone IDs are not supported by this function: TID_SIT_ANY, TID_SIT_NO_CIRCUIT_INTERLATA, TID_SIT_REORDER_TONE_INTERLATA, and TID_SIT_INEFFECTIVE_OTHER.</p>
mode	specifies the mode in which the function will run. Valid values are: <ul style="list-style-type: none">• EV_ASYNC – asynchronous mode• EV_SYNC – synchronous mode (default)

■ Cautions

- Only the default call progress tones as listed in the **toneid** parameter description are supported for this function. The following tone IDs are not supported by this function: TID_SIT_ANY, TID_SIT_NO_CIRCUIT_INTERLATA, TID_SIT_REORDER_TONE_INTERLATA, and TID_SIT_INEFFECTIVE_OTHER.
- When **dx_deletetone()** is issued on a board device in asynchronous mode, and the function is immediately followed by another similar call prior to completion of the previous call on the same device, the subsequent call will fail with device busy.

■ Errors

If the function returns -1, use the Dialogic® 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_SYSTEM
error from operating system

delete a specific call progress tone — dx_deletetone()

EDX_TONEID
bad tone template ID

■ Example

```
#include "srllib.h"
#include "dxxplib.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: srl.lib.h
dxxx.lib.h

Category: Global Tone Detection

Mode: synchronous

Platform: HMP Software, Springware boards

■ 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_blddt()**, **dx_bldst()**, **dx_bldstcad()**, or **dx_blddtcad()**.

Note: On Springware boards, calling **dx_deltones()** suspends all tone detection on the specified channel, including DTMF tone detection; it resumes upon function return. The user may experience missed or duplicated DTMF digits if **dx_deltones()** is called while a DTMF digit is being detected. It is recommended that you call **dx_clr digbuf()** after **dx_deltones()** to clear any inappropriately detected DTMF digits from internal digit buffer.

Parameter	Description
------------------	--------------------

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

■ 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 Dialogic[®] 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_BADPROD
Function not supported on this board

EDX_SYSTEM
Error from operating system

■ Example

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

main()
{
    int dxxxxdev;

    /*
     * Open the Voice Channel Device and Enable a Handler
     */
    if ( ( dxxxxdev = dx_open( "dxxxxB1C1", 0 ) ) == -1 ) {
        perror( "dxxxxB1C1" );
        exit( 1 );
    }

    /*
     * Delete all Tone Templates
     */
    if ( dx_deltone( dxxxxdev ) == -1 ) {
        printf( "Unable to Delete all the Tone Templates\n" );
        printf( "LastError = %d Err Msg = %s\n",
            ATDV_LASTERR( dxxxxdev ), ATDV_ERRMSG( dxxxxdev ) );
        dx_close( dxxxxdev );
        exit( 1 );
    }

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

    /*
     * Close the opened Voice Channel Device
     */
    if ( dx_close( dxxxxdev ) != 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_deltone() — delete all user-defined tones

- [dx_bldst\(\)](#)
- [dx_bldstcad\(\)](#)
- [dx_blddtcad\(\)](#)

dx_dial()

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

Inputs:

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

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

Includes: srllib.h
 dxllib.h

Category: I/O

Mode: asynchronous or synchronous

Platform: HMP Software, Springware boards

■ 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 *Dialogic® Voice API Programming Guide*. For HMP Software, see also the *Dialogic® 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.

Parameter	Description
chdev	specifies the valid channel device handle obtained when the channel was opened using dx_open()
dialstrp	points to the ASCII dial string. dialstrp must contain a null-terminated string of ASCII characters. For a list of valid dialing and control characters, see Table 1 and Table 2. The maximum dial string size (number of digits) is 275 for HMP Software and 200 for Springware boards.

dx_dial() — dial an ASCII string

Parameter	Description
capp	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 .
mode	specifies whether the ASCII string will be dialed with or without call progress analysis enabled, and whether the function will run asynchronously or synchronously. This parameter is a bit mask that can be set to a combination of the following values: <ul style="list-style-type: none">• DX_CALLP – enables call progress analysis• DX_CNGTONE – generates fax CNG tone after dialing to indicate to the remote side that a fax call is coming. Some fax machines expect a CNG tone before receiving a fax call. Use with DX_CALLP.• EV_ASYNC – runs dx_dial() asynchronously• EV_SYNC – runs dx_dial() synchronously (default) On HMP Software, call progress analysis (PerfectCall) is enabled directly through dx_dial() . The dx_initcallp() function is not supported. 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.

■ Asynchronous Operation

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

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 returns a 0 to indicate that it has completed successfully.

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

■ Valid Dial String Characters

On HMP Software, the following is a list of valid dialing and control characters.

Table 1. Valid Dial String Characters (HMP Software)

Characters	Description	Valid in Dial Mode	
		DTMF	MF
On Keypad			
0 1 2 3 4 5 6 7 8 9	digits	Yes	Yes
*	asterisk or star	Yes	Yes (KP)
#	pound, hash, number, or octothorpe	Yes	Yes (ST)
Not on Keypad			
a		Yes	Yes (ST1) (Windows®) (PST) (Linux)
b		Yes	Yes (ST2)
c		Yes	Yes (ST3)
d		Yes	
Special Control			
,	pause for 2.5 seconds (comma)	Yes	Yes
T	Dial Mode: Tone (DTMF) (default)	Yes	Yes
M	Dial Mode: MF	Yes	Yes

When using **dx_dial()** on HMP Software, be aware of the following considerations:

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

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

Table 2. Valid Dial String Characters (Springware boards)

Characters	Description	Valid in Dial Mode		
		DTMF	MF	Pulse
On Keypad				
0 1 2 3 4 5 6 7 8 9	digits	Yes	Yes	Yes

Table 2. Valid Dial String Characters (Springware boards) (Continued)

Characters	Description	Valid in Dial Mode		
		DTMF	MF	Pulse
*	asterisk or star	Yes	Yes (KP)	
#	pound, hash, number, or octothorpe	Yes	Yes (ST)	
Not on Keypad				
a		Yes	Yes (ST1) (Windows PST) (Linux)	
b		Yes	Yes (ST2)	
c		Yes	Yes (ST3)	
d		Yes		
Special Control				
,	pause (comma)	Yes	Yes	
&	flash (ampersand)	Yes	Yes	
T	Dial Mode: Tone (DTMF) (default)	Yes	Yes	Yes
P	Dial Mode: Pulse	Yes	Yes	Yes
M	Dial Mode: MF	Yes	Yes	Yes
L	call progress analysis: local dial tone	Yes	Yes	Yes
I	call progress analysis: international dial tone	Yes	Yes	Yes
X	call progress analysis: special dial tone	Yes	Yes	Yes

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.
- 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”.

■ 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.
- 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 Dialogic® 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

EDX_SYSTEM
Error from operating system

■ Example

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

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

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

dx_dial() — dial an ASCII string

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

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

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

/*
 * Change call progress analysis default busy cadence
 */
if ( dx_chgdur( TID_BUSY1, 550, 400, 550, 400 ) < 0 ) {
    /* handle error */
}
if ( dx_chgrepcnt( 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_dialtpt\(\)](#)
- [dx_stopch\(\)](#)
- event management functions in the *Dialogic® 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_CAP](#) data structure
- call progress analysis topic in the *Dialogic® Voice API Programming Guide*
- [ATDX_ANSRSIZ\(\)](#)
- [ATDX_CONNTYPE\(\)](#)
- [ATDX_CPERROR\(\)](#)
- [ATDX_FRQDUR\(\)](#)
- [ATDX_FRQDUR2\(\)](#)
- [ATDX_FRQDUR3\(\)](#)
- [ATDX_FRQHZ\(\)](#)
- [ATDX_FRQHZ2\(\)](#)
- [ATDX_FRQHZ3\(\)](#)
- [ATDX_FRQOUT\(\)](#)
- [ATDX_LONGLOW\(\)](#)
- [ATDX_SHORTLOW\(\)](#)
- [ATDX_SIZEHI\(\)](#)

dx_dial() — dial an ASCII string

dx_dialtpt()

Name: int dx_dialtpt(chdev, dialstrp, tptp, capp, mode)

Inputs:

- int chdev • valid channel device handle
- char *dialstrp • pointer to the ASCIIZ dial string
- DV_TPT *tptp • pointer to the Termination Parameter Table structure
- 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: Springware boards Linux

■ Description

The **dx_dialtpt()** function allows an application to dial an outbound call using a DV_TPT (Termination Parameter Table). This function works the same way as **dx_dial()** but with the enhancement that the **tptp** parameter allows termination conditions for call progress analysis to be provided. Once dialing is completed and call progress analysis is in progress, call progress analysis can be terminated if one of the conditions in DV_TPT is satisfied.

After **dx_dialtpt()** terminates, if the return value from **ATDX_CPTERM()** is CR_STOPPED, use the **ATDX_TERMMSK()** function to determine the reason for the termination.

Parameter	Description
chdev	specifies the valid channel device handle obtained when the channel was opened using dx_open()
dialstrp	points to the ASCII dial string. dialstrp must contain a null-terminated string of ASCII characters. For a list of valid dialing and control characters, see Table 1, “Valid Dial String Characters (HMP Software)” , on page 179 and Table 2, “Valid Dial String Characters (Springware boards)” , on page 179.
tpt	points to the Termination Parameter Table structure, DV_TPT, that specifies the termination conditions for the device handle.

dx_dialtpt() — dial an outbound call using a TPT

Parameter	Description
capp	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 .
mode	specifies whether the ASCII string will be dialed with or without call progress analysis enabled, and whether the function will run asynchronously or synchronously. This parameter is a bit mask that can be set to a combination of the following values: <ul style="list-style-type: none">• DX_CALLP – enable Call Progress Analysis.• EV_ASYNC – run dx_dialtpt() asynchronously.• EV_SYNC – run dx_dialtpt() synchronously. (default) To run dx_dialtpt() without call progress analysis, specify only EV_ASYNC or EV_SYNC. On Springware boards, to use call progress analysis (PerfectCall), you must call dx_initcallp() prior to calling dx_dialtpt() . Otherwise, dx_dialtpt() uses basic call progress analysis. If dx_dialtpt() with call progress analysis is performed on a channel that is on-hook, the function will only dial digits. Call progress analysis will not occur.

■ Cautions

- This function will fail if an invalid channel device handle is specified.
- Dialing cannot be terminated using the [DV_TPT](#) structure; only call progress analysis can be terminated.

■ Errors

In asynchronous mode, the function returns immediately and either a TDX_CALLP (with call progress analysis) or a TDX_DIAL (without call progress analysis) event is queued upon completion. If a failure occurs during operation then a TDX_ERROR event will be queued. 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

EDX_SYSTEM
Error from operating system

■ Example

This example demonstrates basic call progress analysis with default DX_CAP parameters (synchronous mode).

dial an outbound call using a TPT — dx_dialtpt()

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

int      chdev;      /* Channel device handle */
DV_TPT  tpt;        /* Termination Parameter Table Structure */
long    term;       /* Reason for termination */

/* Open board 1 channel 1 device */
if ((chdev = dx_open("dxxxB1C1", 0)) == -1) {
    printf("Cannot open channel dxxxB1C1.  errno = %d", errno);
    exit(1);
}

/* Set up DV_TPT */
dx_clrtpt(&tpt, 1);
tpt.tp_type = IO_EOT;
tpt.tp_termno = DX_MAXTIME;
tpt.tp_length = 100;
tpt.tp_flags = TF_MAXTIME;

/* Take the phone off-hook */
if (dx_sethook(chdev, DX_OFFHOOK, EV_SYNC) == -1) {
    printf("Error message = %s", ATDV_ERRMSGP(chdev));
    exit(1);
}

/* Perform outbound dial with TPT and default Call Progress Analysis Parameters */
if (dx_dialtpt(chdev, "5551212", &tpt, (DX_CAP *)NULL, DX_CALLP|EV_SYNC) == -1) {
    printf("Error message = %s", ATDV_ERRMSGP(chdev));
    exit(1);
}

if (ATDX_CPTERM(chdev) == CR_STOPPED) {
    if ((term = ATDX_TERMMSK(chdev)) != AT_FAILURE) {
        if (term == TM_MAXTIME) {
            printf("Call Progress Analysis terminated after max time.\n");
        } else {
            printf("Unknown termination reason %x\n", term);
        }
    } else {
        printf("Error message = %s", ATDV_ERRMSGP(chdev));
        exit(1);
    }
}
}
```

■ See Also

- [dx_dial\(\)](#)

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: HMP Software, Springware boards

■ 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.

Parameter	Description
chdev	specifies the valid channel device handle obtained when the channel was opened using dx_open()
toneid	specifies the user-defined tone identifier for which detection is being disabled To disable detection of all user-defined tones on the channel, set toneid to TONEALL.
evt_mask	specifies whether to disable detection of the user-defined tone going on or going off. Set to one or both of the following using a bitwise-OR () operator. <ul style="list-style-type: none">• DM_TONEON – disable TONE ON detection• DM_TONEOFF – disable TONE OFF detection evt_mask affects the enabled/disabled status of the tone template and remains in effect until dx_distone() or dx_enbtone() is called again to reset it.

■ 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 Dialogic® 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_BADPROD**
Function not supported on this board
- EDX_SYSTEM**
Error from operating system
- EDX_TNMSGSTATUS**
Invalid message status setting
- EDX_TONEID**
Bad tone ID

■ Example

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

#define TID_1 101

main()
{
    int dxxxdev;

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

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

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

`dx_distone()` — disable detection of a user-defined tone

```
/*
 * 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_blddt\(\)](#), [dx_bldst\(\)](#), [dx_blddtcad\(\)](#), [dx_bldstcad\(\)](#)
- [dx_enbtone\(\)](#)
- global tone detection topic in the *Dialogic® Voice API Programming Guide*
- [dx_getevt\(\)](#)
- [DX_CST](#) data structure
- [sr_getevtdatap\(\)](#) in the *Dialogic® Standard Runtime Library API Library Reference*

`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`
 `dxxlib.h`

Category: Global Tone Detection

Mode: synchronous

Platform: HMP Software, Springware boards

■ 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()`.

Parameter	Description
chdev	specifies the valid channel device handle obtained when the channel was opened using <code>dx_open()</code>
toneid	specifies the user-defined tone identifier for which detection is being enabled To enable detection of all user-defined tones on the channel, set toneid to <code>TONEALL</code> .
evt_mask	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 (<code> </code>) operator. <ul style="list-style-type: none"> • <code>DM_TONEON</code> – enable TONE ON detection • <code>DM_TONEOFF</code> – enable TONE OFF detection evt_mask 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.

dx_enbtone() — *enable detection of a user-defined tone*

■ **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 Dialogic[®] 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_BADPROD
Function not supported on this board

EDX_SYSTEM
Error from operating system

EDX_TONEID
Bad tone ID

EDX_TNMSGSTATUS
Invalid message status setting

■ **Example**

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

#define TID_1 101

main()
{
    int dxxxdev;

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

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

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


enable detection of a user-defined tone — `dx_enbtone()`

```
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 *Dialogic® Voice API Programming Guide*
- [dx_getevt\(\)](#)
- [DX_CST](#) data structure
- [sr_getevtdatap\(\)](#) in *Dialogic® Standard Runtime Library API Library Reference*

dx_fileclose()

Name: int dx_fileclose(handle)

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

Returns: 0 if success
-1 if failure

Includes: srllib.h
dxxxlib.h

Category: File Manipulation

Mode: synchronous

Platform: HMP Software Windows, Springware boards Windows

■ 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.

■ Example

```
/*
 * Play a voice file. Terminate on receiving 4 digits or at end of file
 */

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

main()
{
    int chdev;
    DX_IOTT iott;
    DV_TPT tpt;
    DV_DIGIT dig;
    .
    .
}
```

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

/* set up DX_IOTT */
iott.io_type = IO_DEV|IO_EOT;
iott.io_bufp = 0;
iott.io_offset = 0;
iott.io_length = -1; /* play till end of file */
if((iott.io_handle = dx_fileopen("prompt.vox",
    O_RDONLY|O_BINARY)) == -1) {
    /* process error */
}

/* set up DV_TPT */
dx_clrtpt(&tpt,1);
tpt.tp_type = IO_EOT; /* only entry in the table */
tpt.tp_termno = DX_MAXDTMF; /* Maximum digits */
tpt.tp_length = 4; /* terminate on four digits */
tpt.tp_flags = TF_MAXDTMF; /* Use the default flags */

/* clear previously entered digits */
if (dx_clrdigbuf(chdev) == -1) {
    /* process error */
}

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

/* get digit using dx_getdig() and continue processing. */
.
.
if (dx_fileclose(iott.io_handle) == -1) {
    /* process error */
}
}
```

■ See Also

- [dx_fileopen\(\)](#)
- [dx_fileseek\(\)](#)
- [dx_fileread\(\)](#)
- [dx_filewrite\(\)](#)

dx_fileerrno() — return the system error value

dx_fileerrno()

Name: int dx_fileerrno(void)
Inputs: none
Returns: system error value
Includes: srllib.h
dxxxlib.h
Category: File Manipulation
Mode: synchronous
Platform: HMP Software Windows, Springware boards Windows

■ 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 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 3 lists the system error values that may be returned by **dx_fileerrno()**.

Table 3. System Error Values

Value	Description
E2BIG	Argument list too long.
EACCES	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.
EAGAIN	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.

return the system error value — dx_fileerrno()

Table 3. System Error Values

Value	Description
EBADF	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.
EDOM	Math argument.
EEXIST	Files exist. An attempt has been made to create a file that already exists. For example, the <code>_O_CREAT</code> and <code>_O_EXCL</code> flags are specified in an <code>_open</code> call, but the named file already exists.
EINTR	A signal was caught.
EINVAL	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 <code>fseek</code>) 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 <code>pmode</code> argument.
EIO	Error during a Windows open.
EMFILE	Too many open files. No more file handles are available, so no more files can be opened.
ENOENT	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.
ENOMEM	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.
ENOSPC	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).
ERANGE	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.
ESR_TMOUT	Timed out waiting for event.
EXDEV	Cross-device link. An attempt was made to move a file to a different device (using the <code>rename</code> function).

■ **Cautions**

None.

■ **Errors**

None.

■ **Example**

```
rc=dx_fileopen(fileName, O_RDONLY);
if (rc == -1) {
    printf("Error opening %s, system error: %d\n", fileName, dx_fileerrno());
}
```

dx_fileerrno() — return the system error value

■ **See Also**

None.

dx_fileopen()

Name: int dx_fileopen(filep, flags, pmode)

Inputs: const char *filep • filename
int flags • type of operations allowed
int pmode • permission mode

Returns: file handle if success
-1 if failure

Includes: srllib.h
dxxxlib.h

Category: File Manipulation

Mode: synchronous

Platform: HMP Software Windows, Springware boards Windows

■ 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 Microsoft[®] Visual C++[®].

■ Cautions

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

■ Errors

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

■ Example

```
/* 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>
```

***dx_fileopen()* — open a file**

```
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_clrldigbuf(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()

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: HMP Software Windows, Springware boards Windows

■ 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 Microsoft[®] Visual C++[®].

■ Cautions

None.

■ Errors

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

■ Example

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

main()

{

int cd;                /* channel device descriptor */
DX_UIO myio;         /* user definable I/O structure */
```

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

```
/*
 * User defined I/O functions
 */
int my_read(fd,ptr,cnt)
int fd;
char * ptr;
unsigned cnt;

{
    printf("My read\n");
    return(dx_fileread(fd,ptr,cnt));
}

/*
 * my write function
 */
int my_write(fd,ptr,cnt)
int fd;
char * ptr;
unsigned cnt;
{
    printf("My write \n");
    return(dx_filewrite(fd,ptr,cnt));
}

/*
 * my seek function
 */
long my_seek(fd,offset,whence)
int fd;
long offset;
int whence;
{
    printf("My seek\n");
    return(dx_fileseek(fd,offset,whence));
}
void main(argc,argv)
int argc;
char *argv[];
{
    .
    . /* Other initialization */
    .
    DX_UIO uioblk;

    /* Initialize the UIO structure */
    uioblk.u_read=my_read;
    uioblk.u_write=my_write;
    uioblk.u_seek=my_seek;

    /* Install my I/O routines */
    dx_setuio(uioblk);
    vodat_fd = dx_fileopen("JUNK.VOX",O_RDWR|O_BINARY);

    /*This block uses standard I/O functions */
    iott->io_type = IO_DEV|IO_CONT
    iott->io_fhandle = vodat_fd;
    iott->io_offset = 0;
    iott->io_length = 20000;

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

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

/*This block uses my I/O functions */
iott->io_type = IO_DEV|IO_UIO|IO_EOT
iott->io_fhandle = vodat_fd;
iott->io_offset = 10003;
iott->io_length = 20000;
devhandle = dx_open("dxxxB1C1", 0);
dx_sethook(devhandle, DX_ONHOOK, EV_SYNC)
dx_wtring(devhandle, 1, DX_OFFHOOK, EV_SYNC);
dx_clrdigbuf;
    if(dx_rec(devhandle, iott, (DX_TPT*)NULL, RM_TONE|EV_SYNC) == -1) {
        perror("");
        exit(1);
    }
dx_clrdigbuf(devhandle);
    if(dx_play(devhandle, iott, (DX_TPT*)EV_SYNC) == -1 {
        perror("");
        exit(1);
    }
dx_close(devhandle);

}
```

■ **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: HMP Software Windows, Springware boards Windows

■ 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 Microsoft® Visual C++®.

■ Cautions

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

■ Errors

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

■ Example

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

main()
{
    int cd;                /* channel device descriptor */
    DX_UIO myio;          /* user definable I/O structure */
}
```

```

/*
 * User defined I/O functions
 */
int my_read(fd,ptr,cnt)
int fd;
char * ptr;
unsigned cnt;
{
    printf("My read\n");
    return(dx_fileread(fd,ptr,cnt));
}

/*
 * my write function
 */
int my_write(fd,ptr,cnt)
int fd;
char * ptr;
unsigned cnt;
{
    printf("My write \n");
    return(dx_filewrite(fd,ptr,cnt));
}

/*
 * my seek function
 */
long my_seek(fd,offset,whence)
int fd;
long offset;
int whence;
{
    printf("My seek\n");
    return(dx_fileseek(fd,offset,whence));
}

void main(argc,argv)
int argc;
char *argv[];
{
    .
    . /* Other initialization */
    .
    DX_UIO uioblk;

    /* Initialize the UIO structure */
    uioblk.u_read=my_read;
    uioblk.u_write=my_write;
    uioblk.u_seek=my_seek;

    /* Install my I/O routines */
    dx_setuio(uioblk);
    vodat_fd = dx_fileopen("JUNK.VOX",O_RDWR|O_BINARY);

    /*This block uses standard I/O functions */
    iott->io_type = IO_DEV|IO_CONT
    iott->io_fhandle = vodat_fd;
    iott->io_offset = 0;
    iott->io_length = 20000;

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

```

dx_fileseek() — move a file pointer

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

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

}
```

■ See Also

- [dx_fileopen\(\)](#)
- [dx_fileclose\(\)](#)
- [dx_fileread\(\)](#)
- [dx_filewrite\(\)](#)

`dx_filewrite()`

Name: `int dx_filewrite(handle, buffer, count)`

Inputs:

<code>int handle</code>	• handle returned from <code>dx_fileopen()</code>
<code>void *buffer</code>	• data to be written
<code>unsigned int count</code>	• number of bytes

Returns: number of bytes if success
-1 if failure

Includes: `srllib.h`
`dxxlib.h`

Category: File Manipulation

Mode: synchronous

Platform: HMP Software Windows, Springware boards Windows

■ 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 Microsoft® Visual C++®.

■ Cautions

None.

■ Errors

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

■ Example

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

main()
{
    int cd;                /* channel device descriptor */
    DX_UIO myio;          /* user definable I/O structure */
}
```

`dx_filewrite()` — write data from a buffer into a file

```
/*
 * 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;
```


write data from a buffer into a file — `dx_filewrite()`

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

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

■ See Also

- [dx_fileopen\(\)](#)
- [dx_fileclose\(\)](#)
- [dx_fileseek\(\)](#)
- [dx_fileread\(\)](#)

dx_getctinfo()

Name: int dx_getctinfo(chdev, ct_devinfo)

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

Returns: 0 on success
-1 on error

Includes: srllib.h
dxxxlib.h

Category: TDM Routing

Mode: synchronous

Platform: HMP Software, Springware boards

■ Description

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

Parameter	Description
chdev	specifies the valid voice channel handle obtained when the channel was opened using dx_open()
ct_devinfo	specifies a pointer to the CT_DEVINFO structure that will contain the voice channel device information

■ Cautions

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

■ Errors

If the function returns -1, use the Dialogic® 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
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

■ **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("dxxxBlC1", 0)) == -1) {
        /* process error */
    }

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

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

■ **See Also**

- [ag_getctinfo\(\)](#)
- [dt_getctinfo\(\)](#) in the *Digital Network Interface Software Reference*
- [gc_GetCTInfo\(\)](#) in the *Dialogic® Global Call API Library Reference*
- [ipm_GetCTInfo\(\)](#) in the *Dialogic® IP Media Library API Library Reference*

dx_getcursv() — *return the specified current speed and volume settings*

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: srllib.h
 dxxxlib.h

Category: Speed and Volume

Mode: synchronous

Platform: HMP Software, Springware boards

■ 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()**.

Note: On HMP Software, before you can use the speed control feature, you must enable this feature in the [decoder] section of the CONFIG file. For more information, see the Configuration Guide applicable to your release or product.

Parameter	Description
chdev	specifies the valid channel device handle obtained when the channel was opened using dx_open()
curvolp	points to an integer that represents the current absolute volume setting for the channel. This value will be between -30 dB and +10 dB.
curspeedp	points to an integer that represents the current absolute speed setting for the channel. This value will be between -50% and +50%.

■ Cautions

On HMP Software, 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.

return the specified current speed and volume settings — dx_getcursv()

■ Errors

If the function returns -1, use the Dialogic® 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_BADPROD

Function not supported on this board

EDX_SYSTEM

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

■ Example

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

/*
 * Global Variables
 */

main()
{
    int dxxxdev;
    int curspeed, curvolume;

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

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

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

dx_getcursv() — return the specified current speed and volume settings

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

■ **See Also**

- [dx_adjsv\(\)](#)
- [dx_addspddig\(\)](#)
- [dx_addvoldig\(\)](#)
- [dx_setsvmt\(\)](#)
- [dx_getsvmt\(\)](#)
- [dx_setsvcond\(\)](#)
- [dx_clrsvcond\(\)](#)
- speed and volume modification tables in the *Voice API Programming Guide*
- [DX_SVMT](#) data structure

`dx_getdig()`

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

Inputs:

- `int chdev` • valid channel device handle
- `DV_TPT *tptp` • pointer to Termination Parameter Table structure
- `DV_DIGIT *digitp` • pointer to User Digit Buffer structure
- `unsigned short mode` • asynchronous/synchronous setting

Returns: 0 to indicate successful initiation (asynchronous)
number of digits if successful (synchronous)
-1 if failure

Includes: `srllib.h`
`dxxxlib.h`

Category: I/O

Mode: asynchronous or synchronous

Platform: HMP Software, Springware boards

■ 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.

For HMP Software, the return value of `dx_getdig()` in synchronous mode returns 0 instead of 1 when there are no digits in the buffer. The NULL character in the digit string 'dg_value' is no longer counted as a digit. Similarly, when `dx_getdig()` returns the number of digits, the terminating NULL is no longer added to the number of digits. (The NULL was previously counted in the `numdig` return value calculation, but since it is not a digit, the NULL is no longer included.)

For Springware boards, the terminating NULL is included in the number of digits. So for Springware boards, `dx_getdig()` still returns 1 when there are no digits in the buffer.

Parameter	Description
<code>chdev</code>	specifies the valid channel device handle obtained when the channel was opened using <code>dx_open()</code>
<code>tptp</code>	points to the Termination Parameter Table structure, <code>DV_TPT</code> , which specifies termination conditions for this function. For a list of possible termination conditions, see <code>DV_TPT</code> , on page 481.

dx_getdig() — collect digits from a channel digit buffer

Parameter	Description
digitp	points to the User Digit Buffer structure, DV_DIGIT, where collected digits and their types are stored in arrays. For a list of digit types, see DV_DIGIT, on page 478. For more information about creating user-defined digits, see dx_addtone() .
mode	specifies whether to run dx_getdig() asynchronously or synchronously. Specify one of the following: <ul style="list-style-type: none">• EV_ASYNC – run asynchronously• EV_SYNC – run synchronously (default)

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

- Notes:**
1. The maximum size of the digit buffer varies with the board type and technology. Multiple calls to **dx_getdig()** may be required to retrieve all digits in the digit buffer. On Springware boards, you can use [ATDX_BUFDIGS\(\)](#) to see if any digits are left in the digit buffer after a call to **dx_getdig()**.
 2. By default, after the maximum number of digits is received, all subsequent digits will be discarded. On Springware boards, you can use [dx_setdigbuf\(\)](#) with mode parameter set to DX_DIGCYCLIC, which will cause all incoming digits to overwrite the oldest digit in the buffer.
 3. Instead of getting digits from the DV_DIGIT structure using **dx_getdig()**, an alternative method is to enable the DE_DIGITS call status transition event using [dx_setevtmsk\(\)](#) and get them from the DX_EBLK event queue data (ev_data) using [dx_getevt\(\)](#) or from the DX_CST call status transition data (cst_data) using [sr_getevtdata\(\)](#).

■ 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 Dialogic® Standard Runtime Library (SRL) Event Management functions to handle the termination event. For more information, see the *Dialogic® 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. Use [ATDX_TERMMSK\(\)](#) to determine the reason for termination.

If the function is operating synchronously and there are no digits in the buffer, the return value from this function will be 0.

■ Cautions

- Global DPD is supported on Springware boards only.
- Some MF digits use approximately the same frequencies as DTMF digits (see [Section 6.1, “DTMF and MF Tone Specifications”](#), on page 533). 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 Dialogic® 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_BADTPT
Invalid [DV_TPT](#) entry

EDX_BUSY
Channel busy

EDX_SYSTEM
Error from operating system

■ Example 1

This example illustrates how to use [dx_getdig\(\)](#) in synchronous mode.

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

dx_getdig() — collect digits from a channel digit buffer

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

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

    /* initiate the call */
    .
    .

    /* Set up the DV_TPT and get the digits */
    dx_clrtpt(tpt,3);
    tpt[0].tp_type = IO_CONT;
    tpt[0].tp_termno = DX_MAXDTMF; /* Maximum number of digits */
    tpt[0].tp_length = 4; /* terminate on 4 digits */
    tpt[0].tp_flags = TF_MAXDTMF; /* terminate if already in buf. */

    tpt[1].tp_type = IO_CONT;
    tpt[1].tp_termno = DX_LCOFF; /* LC off termination */
    tpt[1].tp_length = 3; /* Use 30 msec (10 msec resolution timer) */
    tpt[1].tp_flags = TF_LCOFF|TF_10MS; /* level triggered, clear history,
     * 10 msec resolution */

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

    /* clear previously entered digits */
    if (dx_clrdigbuf(chdev) == -1) {
        /* process error */
    }
    if ((numdigs = dx_getdig(chdev,tpt, &digp, EV_SYNC)) == -1) {
        /* process error */
    }

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

    /* go to next state */
    .
    .
}
```

■ Example 2

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

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

#define MAXCHAN 24

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

collect digits from a channel digit buffer — dx_getdig()

```
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_enbhdr(), set up handler function to handle dx_getdig()
         * completion events on this channel.
         */
        if (sr_enbhdr(chdev[i], TDX_GETDIG, digit_handler) == -1) {
            /* process error */
        }
        /* initiate the call */
        .
        .

        /* Set up the DV_TPT and get the digits */
        dx_clrtpt(tpt,3);

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

        tpt[1].tp_type = IO_CONT;
        tpt[1].tp_termno = DX_LCOFF; /* LC off termination */
        tpt[1].tp_length = 3; /* Use 30 msec (10 msec resolution timer) */
        tpt[1].tp_flags = TF_LCOFF|TF_10MS; /* level triggered, clear
         * history, 10 msec resolution */

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

        /* clear previously entered digits */
        if (dx_clrdigbuf(chdev[i]) == -1) {
            /* process error */
        }
        if (dx_getdig(chdev[i], tpt, &digp[chdev[i]], EV_ASYNC) == -1) {
            /* process error */
        }
    }

    /* Use sr_waitvt() 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_enbhdr().
     */
    .
    .
}
```

dx_getdig() — collect digits from a channel digit buffer

```
int digit_handler()
{
    int chfd;
    int cnt, numdigs;
    chfd = sr_getevtdev();
    numdigs = strlen(digp[chfd].dg_value);
    for(cnt=0; cnt < numdigs; cnt++) {
        printf("\nDigit received = %c, digit type = %d",
            digp[chfd].dg_value[cnt], digp[chfd].dg_type[cnt]);
    }

    /* Kick off next function in the state machine model. */
    .
    .
    return 0;
}
```

■ See Also

- [ATDX_BUFDIGS\(\)](#)
- [dx_addtone\(\)](#)
- [dx_setdigtyp\(\)](#)
- [DV_DIGIT](#) data structure
- [dx_sethook\(\)](#)

`dx_getdigEx()`

Name: `int dx_getdigEx(chdev, tptp, digitp, mode)`

Inputs:

- `int chdev` • valid channel device handle
- `DV_TPT *tptp` • pointer to Termination Parameter Table structure
- `DV_DIGITEX *digitp` • pointer to Extended 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: Springware boards Linux

■ Description

The `dx_getdigEx()` function collects more than 31 digits and null terminator from an open channel's digit buffer. Use this function instead of `dx_getdig()` to retrieve up to 127 digits and the null terminator. Upon termination of the function, the collected digits are written in ASCIIZ format to the extended digit buffer, which is arranged as a `DV_DIGITEX` structure.

Note: The `DX_MAXTIME` termination condition (specified in the `DV_TPT` structure) is limited to 127.

Parameter	Description
<code>chdev</code>	specifies the valid channel device handle obtained when the channel was opened using <code>dx_open()</code>
<code>tptp</code>	points to the Termination Parameter Table Structure, <code>DV_TPT</code> , which specifies termination conditions for this function. For more information on termination conditions, see <code>DV_TPT</code> , on page 481.
<code>digitp</code>	points to the External Digit Buffer Structure, <code>DV_DIGITEX</code> , where collected digits and their types are stored in arrays. For defines of the digit types, see the <code>DV_DIGIT</code> structure.:
<code>mode</code>	specifies whether to run this function in asynchronous mode (<code>EV_ASYNC</code>) or synchronous mode (<code>EV_SYNC</code>).

■ Cautions

See `dx_getdig()`.

***dx_getdigEx()* — collect more than 31 digits from a channel digit buffer**

■ Errors

In asynchronous mode, the function returns immediately and a TDX_GETDIG event is queued upon completion. If a failure occurs during operation then a TDX_ERROR event will be queued. 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_BADTPT
Invalid DV_TPT entry

EDX_BUSY
Channel busy

EDX_SYSTEM
Error from operating system

■ Example

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

main()
{
    DV_TPT tpt[2];
    DV_DIGITEX digEx;
    int chdev, numdigs, cnt;
    char digval[128];
    char digtype[128];

    /* Open virtual board 1, channel 4 */
    if ((chdev = dx_open("dxxxB1C4",NULL)) == -1) {
        printf("Error opening dxxxB1C4\n");
        exit(1);
    }

    /* Set offhook */
    if (dx_sethook(chdev, DX_OFFHOOK,EV_SYNC) == -1) {
        printf("Error setting channel offhook: %s\n", ATDV_ERRMSGP(chdev));
        dx_close(chdev);
        exit(1);
    }

    /* Set up the tpt structure */
    dx_clrtpt(tpt, 2);

    tpt[0].tp_type = IO_CONT;
    tpt[0].tp_termno = DX_MAXDTMF;
    tpt[0].tp_length = 63; /* Can specify up to 128 */
    tpt[0].tp_flags = TF_MAXDTMF;

    tpt[1].tp_type = IO_EOT;
    tpt[1].tp_termno = DX_MAXTIME;
    tpt[1].tp_length = 600;
    tpt[1].tp_flags = TF_MAXTIME;
}
```

collect more than 31 digits from a channel digit buffer — `dx_getdigEx()`

```
/* Clear previously entered digits */
if (dx_clrldigbuf(chdev) == -1) {
    printf("Error clearing digit buffer: %s\n", ATDV_ERRMSGP(chdev));
    dx_close(chdev);
    exit(1);
}

/* Set digit detection type */
if (dx_setdigtyp(chdev,DM_DTMF) == -1) {
    printf("Error setting digit type: %s\n", ATDV_ERRMSGP(chdev));
    dx_close(chdev);
    exit(1);
}

digEx.numdigits = sizeof(digval);
digEx.dg_valuep = digval;
digEx.dg_typep = digtype;

if ((numdigs = dx_getdigEx(chdev, tpt, &digEx, EV_SYNC)) == -1) {
    printf("Error getting digits: %s\n", ATDV_ERRMSGP(chdev));
    dx_close(chdev);
    exit(1);
}

printf("Number of digits received is (excluding NULL): %d\n",
       (numdigs-1));

for (cnt=0; cnt < numdigs-1; cnt++) {
    printf("Digit received = %c, digit type = %d\n",
          digval[cnt], digtype[cnt]);
}

dx_close(chdev);
}
```

■ See Also

- [dx_getevt\(\)](#)

dx_getevt()

Name: int dx_getevt(chdev, eblkp, timeout)

Inputs: int chdev • valid channel device handle
DX_EBLK *eblkp • pointer to Event Block structure
int timeout • timeout value in seconds

Returns: 0 if success
 -1 if failure

Includes: srllib.h
 dxxxlib.h

Category: Call Status Transition Event

Mode: synchronous

Platform: HMP Software, Springware boards

■ Description

The **dx_getevt()** function monitors channel events synchronously for possible call status transition events in conjunction with **dx_setevtmsk()**. The **dx_getevt()** function blocks and returns control to the program after one of the events set by **dx_setevtmsk()** occurs on the channel specified in the **chdev** parameter. The **DX_EBLK** structure contains the event that ended the blocking.

Parameter	Description
chdev	specifies the valid channel device handle obtained when the channel was opened using dx_open()
eblkp	points to the Event Block structure DX_EBLK , which contains the event that ended the blocking
timeout	specifies the maximum amount of time in seconds to wait for an event to occur. timeout can have one of the following values: <ul style="list-style-type: none">• 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.

- Notes:**
1. When the time specified in **timeout** expires, **dx_getevt()** will terminate and return an error. Use the Standard Attribute function **ATDV_LASTERR()** to determine the cause of the error, which in this case is **EDX_TIMEOUT**.
 2. On Linux, an application can stop the **dx_getevt()** function from within a process or from another process.

From within a process, a signal handler may issue a **dx_stopch()** with the handle for the device waiting in **dx_getevt()**. The **mode** parameter to **dx_stopch()** should be OR'ed with the **EV_STOPGETEVT** flag to stop **dx_getevt()**. In this case **dx_getevt()** will successfully return

with the event DE_STOPGETEVT. The EV_STOPGETEVT flag influences **dx_getevt()** only. It does not affect the existing functionality of **dx_stopch()**. Specifically, if a different function besides **dx_getevt()** is in progress when **dx_stopch()** is called with the EV_STOPGETEVT mode, that function will be stopped as usual. EV_STOPGETEVT will be ignored if **dx_getevt()** is not in progress.

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

■ 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 Dialogic® 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_SYSTEM
Error from operating system

EDX_TIMEOUT
Timeout time limit is reached

■ Example

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

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

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

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

dx_getevt() — monitor channel events synchronously

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

switch (eblk.ev_event) {
case DE_RINGS:
    printf("Ring event occurred.\n");
    break;
case DE_WINK:
    printf("Wink event occurred.\n");
    break;
}
.
.
}
```

■ See Also

- [dx_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`
`dxxplib.h`

Category: Configuration

Mode: synchronous

Platform: HMP Software, Springware boards

■ Description

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

Parameter	Description
<code>dev</code>	specifies the valid device handle obtained when a board (in the format <code>dxxxBn</code>) or channel (<code>dxxxBnCm</code>) was opened using <code>dx_open()</code> . <i>Note:</i> Specifying a board device handle is not supported on Springware boards. <i>Note:</i> On HMP Software, 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.
<code>feature_tablep</code>	specifies a pointer to the <code>FEATURE_TABLE</code> 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 <code>FEATURE_TABLE</code> , on page 518.

■ Cautions

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

■ Errors

If the function returns -1, use the Dialogic® 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`
Parameter error

***dx_getfeaturelist()* — retrieve feature support information for the device**

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

■ **Example**

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

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

    if ((dev = dx_open(chname, 0)) == -1) {
        printf("Error opening \"%s\"\n", chname);
        exit(1);
    }

    if (dx_getfeaturelist(dev, &feature_table) == -1) {
        printf("%s: Error %d getting featurelist\n", chname, ATDV_LASTERR(dev));
        exit(2);
    }

    printf("\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 ");
    }
}
```

retrieve feature support information for the device — dx_getfeaturelist()

```
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 ");
}

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

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

printf("\n\n%s: Tone Features:-\n", chname);
if (feature_table.ft_tone & FT_GTDENABLED) {
    printf("GTDENABLED ");
}
}
```

dx_getfeaturelist() — retrieve feature support information for the device

```
    if (feature_table.ft_tone & FT_GTGENABLED) {
        printf("GTGENABLED ");
    }

    if (feature_table.ft_tone & FT_CADENCE_TONE) {
        printf("CADENCE_TONE");
    }

    printf("\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_SYNTTELLECT) {
        printf("SYNTTELLECT");
    }

    printf("\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%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%s: Miscellaneous Features:-\n", chname);

    if (feature_table.ft_misc & FT_CALLERID) {
        printf("CALLERID");
    }

    printf("\n");

    dx_close(dev);
}
```

■ See Also

- [dx_getctinfo\(\)](#)

`dx_getparm()`

Name: `int dx_getparm(dev, parm, valuep)`

Inputs:

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

Returns: 0 if success
-1 if failure

Includes: `srllib.h`
`dxxlib.h`

Category: Configuration

Mode: synchronous

Platform: HMP Software, Springware boards

■ 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 affect the specified channel only.

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

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

dx_getparm() — *get the current parameter settings*

Parameter	Description
parm	<p>Specifies the define for the parameter type whose value is to be returned in the variable pointed to by valuep.</p> <p>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.</p> <p>For HMP Software, see Table 1, “Voice Board Parameters (HMP Software)”, on page 413 and Table 3, “Voice Channel Parameters (HMP Software)”, on page 415.</p> <p>For Springware boards, see Table 2, “Voice Board Parameters (Springware boards)”, on page 413 and Table 4, “Voice Channel Parameters (Springware boards)”, on page 416.</p>
valuep	<p>Points to the variable where the value of the parameter specified in parm should be returned.</p> <p>Note: You must use a void* cast on the returned parameter value, as demonstrated in the Example section code for this function.</p> <p>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.</p>

■ **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 Dialogic® 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

■ **Example**

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


get the current parameter settings — `dx_getparm()`

```
main()
{
    int bddev;
    unsigned short parmval;

    /* open the board using dx_open(). Obtain board device descriptor in
     * bddev
     */
    if ((bddev = dx_open("dxxxB1",NULL)) == -1) {
        /* process error */
    }

    parmval = 0;    /* CLEAR parmval */

    /* get the number of channels on the board. DXBD_CHNUM is of type
     * unsigned short as specified by the PM_SHORT define in the definition
     * for DXBD_CHNUM in dxxxlib.h. The size of the variable parmval is
     * sufficient to hold the value of DXBD_CHNUM.
     */
    if (dx_getparm(bddev, DXBD_CHNUM, (void *)&parmval) == -1) {
        /* process error */
    }

    printf("\nNumber of channels on board = %d",parmval);
    .
    .
}
```

■ See Also

- [dx_setparm\(\)](#)

dx_GetRscStatus() — return assignment status of a shared resource

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 boards

■ Description

The **dx_GetRscStatus()** function returns the assignment status of the shared resource for the specified channel.

Parameter	Description
chdev	specifies the valid channel device handle obtained when the channel was opened using dx_open()
rsctype	specifies the type of shared resource: <ul style="list-style-type: none">• RSC_FAX – shared fax resource (DSP-based Group 3 Fax, also known as Softfax)
status	points to the data that represents the assignment status of the resource: <ul style="list-style-type: none">• RSC_ASSIGNED – a shared resource of the specified rsctype is assigned to the channel• RSC_NOTASSIGNED – a shared resource of the specified rsctype is not assigned to the channel

■ 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_BADPARAM
Invalid parameter

return assignment status of a shared resource — dx_GetRscStatus()

EDX_SYSTEM

Error from operating system

■ Example

```
/* Check whether a shared Fax resource is assigned to the voice channel */
#include <stdio.h>
#include <srllib.h>
#include <dxxxxlib.h>

main()
{
    int chdev ; /* Fax channel device handle */
    int status;

    /*Open the Voice channel resource (device) using dx_open(). */
    if((chdev = dx_open("dxxxxB1C1", 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_ERRMSGF(chdev), ATDV_LASTERR(chdev));
        if(ATDV_LASTERR(chdev) == EDX_SYSTEM) {
            /* Perform system error processing */
        }
    }
    else {
        printf("The resource status ::%d\n", status);
    }
}
```

■ See Also

None.

dx_GetStreamInfo()

Name: int dx_GetStreamInfo(hBuffer, &StreamStatStruct)

Inputs: int hBuffer • stream buffer handle
DX_STREAMSTAT • pointer to stream status structure
StreamStatStruct

Returns: 0 if successful
-1 if failure

Includes: srllib.h
dxxxlib.h

Category: streaming to board

Mode: synchronous

Platform: HMP Software

■ 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 505.

■ Cautions

None.

■ Errors

Unlike other Dialogic[®] 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

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

main()
{
    int nBuffSize = 32768;
    int hBuffer = -1;
    DX_STREAMSTAT streamStat;
```

retrieve information about the circular stream buffer — `dx_GetStreamInfo()`

```
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.version, streamStat.bytesIn, streamStat.bytesOut, streamStat.headPointer,
           streamStat.tailPointer, streamStat.currentState, streamStat.numberOfBufferUnderruns,
           streamStat.numberOfBufferOverruns, streamStat.BufferSize, streamStat.spaceAvailable,
           streamStat.highWaterMark, streamStat.lowWaterMark);
}
if (dx_CloseStreamBuffer(hBuffer) < 0)
{
    printf("Error closing stream buffer \n");
}
}
```

■ See Also

- [dx_OpenStreamBuffer\(\)](#)

dx_getsvmt() — return the current speed or volume modification table

dx_getsvmt()

Name: int dx_getsvmt(chdev, tabletype, svmtp)

Inputs: int chdev • valid channel device handle
 unsigned short tabletype • type of table to retrieve (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
 dxxxlib.h

Category: Speed and Volume

Mode: synchronous

Platform: HMP Software, Springware boards

■ Description

The **dx_getsvmt()** function returns the current speed or volume modification table to the **DX_SVMT** structure.

Note: On HMP Software, before you can use the speed control feature, you must enable this feature in the [decoder] section of the CONFIG file. For more information, see the Configuration Guide applicable to your release or product.

Parameter	Description
chdev	specifies the valid channel device handle obtained when the channel was opened using dx_open()
tabletype	specifies whether to retrieve the speed or the volume modification table: <ul style="list-style-type: none">• 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 Dialogic® 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

return the current speed or volume modification table — dx_getsvmt()

EDX_BADPROD

Function not supported on this board

EDX_SPDVOL

Must specify either SV_SPEEDTBL or SV_VOLUMETBL

EDX_SYSTEM

Error from operating system

■ Example

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

/*
 * Global Variables
 */

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

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

    /*
     * Get the Current Volume Modification Table
     */
    memset( &svmt, 0, sizeof( DX_SVMT ) );
    if (dx_getsvmt( dxxxdev, SV_VOLUMETBL, &svmt ) == -1){
        printf( "Unable to Get the Current Volume" );
        printf( " Modification Table\n" );
        printf( "Lasterror = %d  Err Msg = %s\n",
            ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
        dx_close( dxxxdev );
        exit( 1 );
    } else {
        printf( "Volume Modification Table is:\n" );
        for ( index = 0; index < 10; index++ ) {
            printf( "decrease[ %d ] = %d\n", index, svmt.decrease[ index ] );
        }

        printf( "origin = %d\n", svmt.origin );
        for ( index = 0; index < 10; index++ ) {
            printf( "increase[ %d ] = %d\n", index, svmt.increase[ index ] );
        }
    }

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

***dx_getsvmt()* — return the current speed or volume modification table**

```
/*
 * Close the opened Voice Channel Device
 */
if ( dx_close( dxdev ) != 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 *Dialogic® Voice API Programming Guide*
- [DX_SVMT](#) data structure

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

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

■ **Example**

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

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

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

    /* Fill in the TDM bus time slot information */
    sc_tsinfo.sc_numts = 1;
    sc_tsinfo.sc_tsarray = &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);
}
```

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

■ **See Also**

- [dx_listen\(\)](#)

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 boards

■ 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.

Parameter	Description
chdev	specifies the valid channel device handle obtained when a channel was opened using dx_open()
bufferp	pointer to buffer where calling line Directory Number (DN) is returned <i>Note:</i> 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_gttextcallid()** 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.

return the calling line Directory Number — dx_gtcallid()

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_BADPARAM

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

■ **Example 1**

The following example is for Linux only.

```
/*$ dx_gtcallid( ) example (Linux only example) $*/

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

int main()
{
    unsigned char buffer[21];    /* char buffer */
    DX_EBLK eblk;               /* event block struct */
    int timeout;                /* timeout for function */
    int chdev;                  /* channel descriptor */
    unsigned short parmval;     /* parameter value */
```

***dx_gtcallid()* — return the calling line Directory Number**

```
/* 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 */
}

/* set RINGS as events to wait on */
if (dx_setevtmsk(chdev, DM_RINGS) == -1) {
    /* process error */
}

timeout = 5;      /* 5 seconds */

if (dx_getevt(chdev, &eblk, timeout) == -1) {
    /* process error */
}

/* Upon receiving ring event, check event data (bit 0) to see if caller ID
 * is available
 */
if (eblk.ev_event == DE_RINGS) {
    if ((eblk->ev_data & 0x0001) == 0)
        exit(0);
    if (dx_gtcallid(chdev, buffer) == -1) {
        /* process error */
    }
    printf("The calling line directory number is %5\n", buffer);
}

/* get caller ID */
if (dx_gtcallid(chdev,buffer) == -1) {
    printf("Error getting caller ID: 0x%x\n",
        ATDV_LASTERR(chdev));
    /* process error */
}
printf("Caller ID = %s\n", buffer);
}
```

■ Example 2

The following example is for Windows only.

```
/*$ dx_gtcallid( ) example (Windows only example) $*/

#include <windows.h>
#include <sys/types.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

/* Dialogic(r) Includes */
#include "srllib.h"
#include "dxxxlib.h"
```

return the calling line Directory Number — dx_gtcallid()

```
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_CLIDOOA) {
            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.
    * *****/
}
```

***dx_gtcallid()* — return the calling line Directory Number**

```
if ( dx_gttextcallid(chdev,CLIDINFO_FRAMETYPE, buffer) != -1) {
    if(buffer[0] == CLASSFRAME_MDM) {
        /* Get and print the date and time */
        if (dx_gttextcallid(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_gttextcallid(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_gttextcallid(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_gttextcallid\(\)](#)
- [dx_wtcallid\(\)](#)
- [dx_setparm\(\)](#)
- [dx_setevtmsk\(\)](#)
- [dx_getevt\(\)](#)
- [DX_EBLK](#) data structure

return the calling line Directory Number — dx_gtcalled()

dx_gtextcallid() — retrieve a caller ID message

dx_gtextcallid()

Name: int dx_gtextcallid (chdev, infotype, bufferp)

Inputs: int chdev • valid channel device handle
int infotype • message type ID
unsigned char *bufferp • pointer to buffer where the requested caller ID message is returned

Returns: 0 success
-1 error return code

Includes: srllib.h
dxxxlib.h

Category: Caller ID

Mode: synchronous

Platform: Springware boards

■ 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.

Parameter	Description
chdev	the valid channel device handle obtained when a channel was opened using dx_open()
infotype	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 4, “Caller ID Common Message Types” , on page 251, Table 5, “Caller ID CLASS Message Types (Multiple Data Message)” , on page 252, Table 6, “Caller ID ACLIP Message Types (Multiple Data Message)” , on page 252, Table 7, “Caller ID CLIP Message Types” , on page 253, and Table 8, “Caller ID JCLIP Message Types (Multiple Data Message)” , on page 253.
bufferp	pointer to buffer where the requested caller ID message is to be stored. All returns are NULL terminated.

■ Common Message Types

The following standard Message Types are available for:

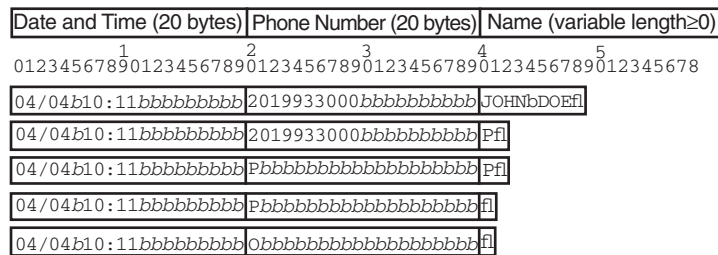
- CLASS (Single Data Message)
- CLASS (Multiple Data Message)
- ACLIP (Single Data Message)
- ACLIP (Multiple Data Message)
- CLIP
- JCLIP

All returns are NULL terminated.

Table 4. Caller ID Common Message Types

Value	Definition>Returns
CLIDINFO_CMPLT	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.
CLIDINFO_GENERAL	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.
CLIDINFO_CALLID	Caller ID (phone number). Can produce EDX_CLIDINFO, EDX_CLIDOOA, and EDX_CLIDBLK errors.
CLIDINFO_FRAME TYPE	Indicates caller ID frame. Does not apply to CLIP. Can produce EDX_CLIDINFO error. Values (depending upon service type): <ul style="list-style-type: none"> • CLASSFRAME_SDM • CLASSFRAME_MDM • ACLIPFRAME_SDM • ACLIPFRAME_MDM • JCLIPFRAME_MDM

Figure 1. Format of General Caller ID Information



Legend:
 b=Blank
 f=NULL
 O=Out of Area
 P=Private

***dx_gtextcallid()* — retrieve a caller ID message**

■ **Message Types for CLASS (Multiple Data Message)**

See Table 4 for the standard Message Types that can also be used. Table 5 lists Message Types that can produce an EDX_CLIDINFO error. All returns are NULL terminated.

Table 5. Caller ID CLASS Message Types (Multiple Data Message)

Value	Definition>Returns
MCLASS_DATETIME	Date and Time (as sent by CO without format characters / and :)
MCLASS_DN	Calling line directory number (digits only)
MCLASS_DDN	Dialed number (digits only)
MCLASS_ABSENCE1	Reason for absence of caller ID (only available if caller name is absent): O = out of area, P = private
MCLASS_REDIRECT	Call forward: 0 = universal; 1 = busy; 2 = unanswered
MCLASS_QUALIFIER	L = long distance call
MCLASS_NAME	Calling line subscriber name
MCLASS_ABSENCE2	Reason for absence of name (only available if caller name is absent): O = out of area, P = private

■ **Message Types for ACLIP (Multiple Data Message)**

See Table 4, “Caller ID Common Message Types”, on page 251 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 ACLIP Message Types (Multiple Data Message)

Value	Definition>Returns
MACLIP_DATETIME	Date and Time (as sent by CO without format characters / and :)
MACLIP_DN	Calling line directory number (digits only)
MACLIP_DDN	Dialed number (digits only)
MACLIP_ABSENCE1	Reason for absence of caller ID (only available if caller name is absent): O = out of area, P = private
MACLIP_REDIRECT	Call forward: 0 = universal; 1 = busy; 2 = unanswered
MACLIP_QUALIFIER	L = long distance call
MACLIP_NAME	Calling line subscriber name
MACLIP_ABSENCE2	Reason for absence of name (only available if caller name is absent): O = out of area, P = private

■ **Message Types for CLIP**

See Table 4, “Caller ID Common Message Types”, on page 251 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 CLIP Message Types

Value	Definition>Returns
CLIP_DATETIME	Date and Time (as sent by CO without format characters / and :)
CLIP_DN	Calling line directory number (digits only)
CLIP_DDN	Dialed number (digits only)
CLIP_ABSENCE1	Reason for absence of caller ID (only available if caller name is absent): O = out of area, P = private
CLIP_NAME	Calling line subscriber name
CLIP_ABSENCE2	Reason for absence of name (only available if caller name is absent): O = out of area, P = private
CLIP_CALLTYPE	1 = voice call, 2 = ring back when free call, 129 = message waiting call
CLIP_NETMSG	Network Message System status: number of messages waiting

■ **Message Types for JCLIP (Multiple Data Message)**

See Table 4, “Caller ID Common Message Types”, on page 251 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 JCLIP Message Types (Multiple Data Message)

Value	Definition>Returns
JCLIP_DN	Calling line directory number (digits only)
JCLIP_DDN	Dialed number (digits only)
JCLIP_ABSENCE1	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)
JCLIP_ABSENCE2	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)

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_DDN can be used to get the dialed number for CLASS MDM (digits only)
- MACLIP_DDN 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

***dx_gtextcallid()* — retrieve a caller ID message**

- `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).

■ **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_BADPARAM`
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_CLIDOOA

Caller ID is out of area (**infotype** = CLIDINFO_CALLID)

EDX_SYSTEM

Error from operating system

All Message Types (**infotype**) can produce an EDX_CLIDINFO error. Message Type CLIDINFO_CALLID can also produce EDX_CLIDOOA and EDX_CLIDBLK errors. Table 9 indicates which caller ID-related error codes are returned for the different Message Types.

Table 9. Caller ID-Related Error Codes Returned for Different Message Types

Message Type ID	Error Codes		
	EDX_CLIDINFO	EDX_CLIDBLK	EDX_CLIDOOA
xx_CMPLT (all formats)	✓		
xx_GENERAL (all formats)	✓		
xx_CALLID (all formats)	✓	✓	✓
xx_FRAMETYPE (all formats)	✓		
MCLASS_xx (CLASS MDM only)	✓		

■ **Example 1**

```

/*$ dx_gtextcallid( ) example to obtain all available caller ID information $*/

#include <sys/types.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
/* Dialogic 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);
    }
}

```

dx_gtextcallid() — retrieve a caller ID message

```
/******
 * 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_FRAME_TYPE, 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");
    /* 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));
    }
}
```


retrieve a caller ID message — dx_gtextcallid()

```

/*****
 * 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));
}
}
}
dx_close(chdev);
return(0);
}

```

■ Example 2

```

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

main()
{
    unsigned char buffer[81];    /* char buffer */
    DX_EBLK eblk;               /* event block struct */
    int timeout;                /* timeout for 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 */
    }

    /* set RINGS as events to wait for */
    if (dx_setevtmsk(chdev, DM_RINGS) == -1) {
        /* process error */
    }

    timeout = 5;                /* 5 seconds */
    if (dx_getevt(chdev, &eblk, timeout) == -1) {
        /* process error */
    }

    /* Upon receiving ring event, check event data (bit 0) to see if caller ID
     * is available
     */
    if ((eblk->ev_data & 0x0001) == 0)
        exit(0);

    /* get caller's name (use equates specific to your caller ID implementation) */
    if (dx_gtextcallid(chdev, MCLASS_NAME, buffer) == -1) {
        printf("Error getting caller's name: 0x%x\n", ATDV_LASTERR(chdev));
        /* process error */
    }

    printf("Caller Name = %s\n", buffer);
}

```

dx_gtextcallid() — retrieve a caller ID message

```
/* get general information - date & time, number, and name */
if (dx_gtextcallid(chdev,CLIDINFO_GENERAL,buffer) == -1) {
    printf("Error getting date&time, number, and name.\n");
    /* process error */
}
printf("Date&time, number, and name = %s\n", buffer);
}
```

■ See Also

- [dx_gtcallid\(\)](#)
- [dx_wtcallid\(\)](#)

retrieve a caller ID message — dx_gtextcallid()

dx_initcallp()

Name: int dx_initcallp(chdev)

Inputs: int chdev • valid channel device handle

Returns: 0 if successful
-1 if failure

Includes: srllib.h
dxxxlib.h

Category: Call Progress Analysis

Mode: synchronous

Platform: Springware boards

■ Description

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 Springware boards, to use call progress analysis, **dx_initcallp()** must be called prior to using **dx_dial()** or **dx_dialtpt()** on the specified channel. If **dx_dial()** or **dx_dialtpt()** 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_chgrepent()** 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_deltone()**) and re-initializing with another call to **dx_initcallp()**. **dx_deltone()** also disables call progress analysis. Note, however, that **dx_deltone()** will erase all user-defined tones from the channel (including any global tone detection information), and not just the call progress analysis tones.

■ Cautions

When you issue this function, the channel must be idle.

■ 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

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

main()
{
    DX_CAP    cap_s;
    int       ddd, car;
    char      *chnam, *dialstrg;
    chnam     = "dxxxxB1C1";
    dialstrg  = "L1234";

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

    /*
     * Delete any previous tones
     */
    if ( dx_deltone(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 */
    }
}
```

***dx_initcallp()* — initialize and activate call progress analysis**

```
/*
 * 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_chgrept\(\)**](#)
- [**dx_deltone\(\)**](#)
- [**dx_TSFStatus\(\)**](#)

`dx_listen()`

Name: `int dx_listen(chdev, sc_tsinfo)`

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

Returns: 0 on success
-1 on error

Includes: `srllib.h`
`dxxlib.h`

Category: TDM Routing

Mode: synchronous

Platform: HMP Software, Springware boards

■ Description

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

The `dx_listen()` function returns immediately with success before the operation is completed. After the operation is completed, the voice receive channel is 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.

Note: The `dx_listenEx()` function extends and enhances the `dx_listen()` function. See the `dx_listenEx()` function reference for more information.

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

Parameter	Description
<code>chdev</code>	specifies the voice channel device handle obtained when the channel was opened using <code>dx_open()</code>
<code>sc_tsinfo</code>	specifies a pointer to the <code>SC_TSINFO</code> structure

■ Cautions

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

***dx_listen()* — connect a voice listen channel to TDM bus time slot**

■ **Errors**

If the function returns -1, use the Dialogic[®] 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

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

Error from operating system

connect a voice listen channel to TDM bus time slot — `dx_listen()`

■ Example

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

main()
{
    int dxdev, ipdev;          /* Channel device handles */
    SC_TSINFO sc_tsinfo;      /* Time slot information structure */
    long scts;                /* TDM bus time slot */

    /* Open IP channel ipmB1C1 */
    if((ipdev = ipm_Open("ipmB1C1", NULL, EV_SYNC)) == -1) {
        /* process error */
    }
    /* Open voice channel dxxxB1C1 */
    if ((dxdev = 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 transmit time slot of IP channel ipmB1C1 */
    if (ipm_GetXmitSlot(ipdev, &sc_tsinfo, EV_SYNC) == -1) {
        /* process error */
    }

    /* Connect the receive timeslot of voice channel dxxxB1C1 to the transmit time slot
    ...of IP channel ipmB1C1 */
    if (dx_listen(dxdev, &sc_tsinfo) == -1) {
        printf("Error message = %s", ATDV_ERRMSGP(dxdev));
        exit(1);
    }
}
```

■ See Also

- [ag_getxmitslot\(\)](#)
- [dx_getxmitslot\(\)](#)
- [dx_unlisten\(\)](#)
- [dx_listenEx\(\)](#)
- [dx_unlistenEx\(\)](#)
- [ipm_Open\(\)](#) in *IP Media Library API Library Reference*
- [ipm_GetXmitSlot\(\)](#) in *IP Media Library API Library Reference*

dx_listenEx() — connect a voice listen channel to TDM bus time slot

dx_listenEx()

Name: int dx_listenEx(chdev, sc_tsinfo, mode)

Inputs: int chdev • valid channel device handle
SC_TSINFO *sc_tsinfo • pointer to TDM bus time slot information structure
unsigned short mode • mode flag

Returns: 0 on success
-1 on error

Includes: srllib.h
dxxxlib.h

Category: TDM Routing

Mode: asynchronous or synchronous

Platform: HMP Software

■ Description

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

The **dx_listenEx()** function extends and enhances the **dx_listen()** function in two ways. First, it adds support for the asynchronous mode of operation and provides event notification upon successful completion or failure of the routing. Second, it enhances the synchronous functionality by blocking the call until the listen action is completed.

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.

Parameter	Description
chdev	specifies the voice channel device handle obtained when the channel was opened using dx_open()
sc_tsinfo	specifies a pointer to the SC_TSINFO structure
mode	specifies the mode of operation: <ul style="list-style-type: none">• EV_SYNC – synchronous mode (default)• EV_ASYNC – asynchronous mode

In synchronous mode, the voice channel is connected to the TDM bus time slot upon return from the **dx_listenEx()** function. By default, this function runs in synchronous mode and returns a 0 to indicate that it has completed successfully. If a failure occurs, this function returns -1.

connect a voice listen channel to TDM bus time slot — dx_listenEx()

In asynchronous mode, a TDX_LISTEN event is queued upon successful completion of the routing. If a failure occurs during routing, a TDX_LISTEN_FAIL event is queued. In some limited cases, such as when invalid arguments are passed to the library, the function may fail before routing is attempted. In such cases, the function returns -1 immediately to indicate failure and no event is queued.

■ **Cautions**

- This function fails when an invalid channel device handle is specified or when an invalid TDM bus time slot number is specified.
- When using this function in asynchronous mode, do not issue another listen operation on the same channel using either **dx_listen()** or **dx_listenEx()** until the TDX_LISTEN event is received. If you attempt to do this, the listen function will return failure.
- It is recommended that you use **dx_listenEx()** and **dx_unlistenEx()** in your application, rather than **dx_listen()** and **dx_unlisten()**. In particular, do not use both pairs of functions on the same channel. Doing so may result in unpredictable behavior.

■ **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

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

***dx_listenEx()* — connect a voice listen channel to TDM bus time slot**

EDX_SH_NOCLK
Switch Handler clock fallback failed

EDX_SYSTEM
Error from operating system

■ **Example 1: Synchronous Mode**

This example code for ***dx_listenEx()*** illustrates the synchronous mode of operation.

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

main()
{
    int dxdev, ipdev;          /* Channel device handles */
    SC_TSINFO sc_tsinfo;     /* Time slot information structure */
    long scts;                /* TDM bus time slot */

    /* Open IP channel ipmB1C1 */
    if((ipdev = ipm_Open("ipmB1C1", NULL, EV_SYNC)) == -1) {
        /* process error */
    }

    /* Open voice channel dxxxB1C1 */
    if ((dxdev = 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 transmit time slot of IP channel ipmB1C1*/
    if (ipm_GetXmitSlot(ipdev, &sc_tsinfo, EV_SYNC) == -1) {
        /* process error */
    }

    /* Connect the receive time slot of voice channel dxxxB1C1 to the transmit time slot
    ...of IP channel ipmB1C1 */
    if (dx_listenEx(dxdev, &sc_tsinfo, EV_SYNC) == -1) {
        printf("Error message = %s", ATDV_ERRMSGP(dxdev));
        exit(1);
    }
}
```

■ **Example 2: Asynchronous Mode**

This example code for ***dx_listenEx()*** illustrates the asynchronous mode of operation.

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

main()
{
    int dxdev, ipdev;          /* Channel device handles */
    SC_TSINFO sc_tsinfo;     /* Time slot information structure */
    long scts;                /* TDM bus time slot */
    int srlmode;

    /* ... */
}
```

connect a voice listen channel to TDM bus time slot — dx_listenEx()

```
/* Set SRL to run in polled mode. */
srlmode = SR_POLLMODE;
if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1) {
    /* process error */
}

/* Open IP channel ipmB1C1 */
if (ipmdev = ipm_Open("ipmB1C1", NULL, EV_SYNC) == -1) {
    /* process error */
}

/* Open voice channel dxxxB1C1 */
if ((dxdev = 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 transmit time slot of IP channel ipmB1C1 */
if (ipm_GetXmitSlot(ipdev, &sc_tsinfo, EV_SYNC) == -1) {
    /* process error */
}

/* Connect the receive time slot of voice channel dxxxB1C1 to the transmit time slot
...of IP channel ipmB1C1 */
if (dx_listenEx(dxdev, &sc_tsinfo, EV_ASYNC) == -1) {
    printf("Error message = %s", ATDV_ERRMSGP(dxdev));
    exit(1);
}

/* Use sr_waitevt to wait for the TDX_LISTEN event */
}
```

■ See Also

- [dx_unlistenEx\(\)](#)
- [dx_unlisten\(\)](#)
- [dx_listen\(\)](#)
- [ipm_Open\(\)](#) in *Dialogic® IP Media Library API Library Reference*
- [ipm_GetXmitSlot\(\)](#) in *Dialogic® IP Media Library API Library Reference*

dx_mreciottdata() — record voice data from two TDM bus time slots

dx_mreciottdata()

Name: dx_mreciottdata (devd, iotp, tptp, xpb, mode, sc_tsinfo)

Inputs:

int devd	• valid channel device handle
DX_IOTT *iotp	• pointer to I/O transfer table
DV_TPT *tptp	• pointer to termination control block
DX_XPB *xpb	• pointer to I/O transfer parameter block
unsigned short mode	• switch to set audible tone, or DTMF termination
SC_TSINFO *sc_tsinfo	• pointer to time slot information structure

Returns: 0 success
-1 error return code

Includes: srllib.h
dxxxlib.h

Category: I/O

Mode: asynchronous or synchronous

Platform: HMP Software, Springware boards Windows

■ Description

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.

On Dialogic® Springware boards on Linux, use the **dx_recm()** and **dx_recmf()** functions for transaction record.

record voice data from two TDM bus time slots — dx_mreciottdata()

Parameter	Description
devd	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.
iotp	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 502.
tptp	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 481.
xpb	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 514.
mode	specifies the attributes of the recording mode. One or more of the following values listed below may be selected in the bitmask using bitwise OR: Choose one only: <ul style="list-style-type: none">• EV_ASYNC – asynchronous mode• EV_SYNC – synchronous mode Choose one or more: <ul style="list-style-type: none">• 0 – standard record mode• MD_NOGAIN – record without automatic gain control (AGC). AGC is on by default.• RM_NOTIFY – (Windows® only) generate record notification beep tone.• RM_TONE – transmit a 200 msec tone before initiating record.
sc_tsinfo	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_tsarray 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.

dx_mreciottdata() — record voice data from two TDM bus time slots

- Voice data files that are specified in the [DX_IOTT](#) structure must be opened with the `O_BINARY` flag.
- 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 function uses [dx_listen\(\)](#) to connect the channel to the first specified time slot, any error returned from [dx_listen\(\)](#) will terminate the function with the error indicated.
- This function connects the channel to the time slot specified in the [SC_TSINFO](#) data structure `sc_tsarrayp[0]` field and remains connected after the function has completed. Both `sc_tsarrayp[0]` and `sc_tsarrayp[1]` must be within the range allowed in [SC_TSINFO](#). No checking is done to verify that `sc_tsarrayp[0]` or `sc_tsarrayp[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 HMP Software, 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_tsarrayp[0]` field.

■ Errors

If the function returns -1, use the Dialogic® 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_BADPARAM`
Invalid parameter passed

EDX_BADTPT
Invalid DV_TPT entry

EDX_BUSY
Busy executing I/O function

EDX_SYSTEM
Error from operating system

■ Example 1

The following example is for Linux applications.

```
#include <fcntl.h>
#include <srllib.h>
#include <xxxxlib.h>

#include <stdio.h>
#include <stdlib.h>

#define MAXLEN 100000

/* Define logging macro */
#define log_rc(B, F) \
    printf (" %-60.60s: ", #B); \
    fflush (stdout); \
    retval = B; \
    printf ("RC=%d\n", retval); \
    if ( retval F ) { printf ("Fatal error!\n"); exit (1); }

main(int argc, char *argv[])
{
    int playerOne, playerTwo, recorder;
    DX_IOTT playOneiott={0}, playTwoiott={0}, recordiott={0};
    DV_TPT playOnetpt={0}, playTwotpt={0}, recordtpt ={0};
    DX_XPB recordxpb={0}, playOnexpb={0}, playTwoxpb={0};

    SC_TSINFO playOnetsinfo, playTwotsinfo, recordtsinfo;
    long playOnescts, playTwoscts;
    long mRectslots[32];

    /* open two play channels and one record channel */
    if ((playerOne = dx_open(argv[3], NULL)) == -1) {
        printf("Could not open %s\n", argv[3]);
        exit (1);
    }

    if ((playerTwo = dx_open(argv[4], NULL)) == -1) {
        printf("Could not open %s\n", argv[4]);
        exit (1);
    }

    if ((recorder = dx_open(argv[5], NULL)) == -1) {
        printf("Could not open %s\n", argv[5]);
        exit (1);
    }

    dx_clrtpt (&playOnetpt, 1);
    dx_clrtpt (&playTwotpt, 1);
    dx_clrtpt (&recordtpt, 1);

    log_rc (playTwoiott.io_fhandle = open (argv[2], O_RDONLY), == -1)
    log_rc (playOneiott.io_fhandle = open (argv[1], O_RDONLY), == -1)
```

dx_mreciottdata() — record voice data from two TDM bus time slots

```
playOneiott.io_type = IO_DEV | IO_EOT;
playOneiott.io_offset = 0;
playOneiott.io_length = -1;

playOnexpb.wFileFormat = FILE_FORMAT_VOX;
playOnexpb.wDataFormat = DATA_FORMAT_MULAW;
playOnexpb.nSamplesPerSec = DRT_8KHZ;
playOnexpb.wBitsPerSample = 8;

playTwoiott.io_type = IO_DEV | IO_EOT;
playTwoiott.io_offset = 0;
playTwoiott.io_length = -1;

playTwoxpb.wFileFormat = FILE_FORMAT_VOX;
playTwoxpb.wDataFormat = DATA_FORMAT_MULAW;
playTwoxpb.nSamplesPerSec = DRT_8KHZ;
playTwoxpb.wBitsPerSample = 8;

/* Get channels' external time slots and fill in mRectslots[] array */
playOnetsinfo.sc_numts = 1;
playOnetsinfo.sc_tsarrayp = &playOnescts;
if (dx_getxmitslot (playerOne, &playOnetsinfo) == -1) {
    /* Handle error */
}

playTwotsinfo.sc_numts = 1;
playTwotsinfo.sc_tsarrayp = &playTwoscts;
if (dx_getxmitslot (playerTwo, &playTwotsinfo) == -1 ) {
    /* Handle error */
}

mRectslots[1] = playTwoscts;
mRectslots[0] = playOnescts;

/* Set up SC_TSINFO structure */
recordtsinfo.sc_numts = 2;
recordtsinfo.sc_tsarrayp = &mRectslots[0];

log_rc (recordiott.io_fhandle = open(argv[6], O_CREAT | O_RDWR, 0666), == -1);
recordiott.io_type = IO_EOT|IO_DEV;
recordiott.io_offset = 0;
recordiott.io_length = MAXLEN;
recordiott.io_bufp = 0;
recordiott.io_nextp = NULL;

recordxpb.wFileFormat = FILE_FORMAT_VOX;
recordxpb.wDataFormat = DATA_FORMAT_MULAW;
recordxpb.nSamplesPerSec = DRT_8KHZ;
recordxpb.wBitsPerSample = 8;

/* Play user-supplied files */
log_rc (dx_playiottdata(playerOne, &playOneiott, NULL, &playOnexpb, EV_ASYNC), ==-1)
log_rc (dx_playiottdata(playerTwo, &playTwoiott, NULL, &playTwoxpb, EV_ASYNC), ==-1)

/* And record from both play channels */
printf("\n Starting dx_mreciottdata");
if (dx_mreciottdata(recorder, &recordiott, NULL, &recordxpb, EV_SYNC|RM_TONE,
    &recordtsinfo) == -1) {
    printf("Error recording from dxxxB1C1 and dxxxB1C2\n");
    printf("error = %s\n", ATDV_ERRMSGP(recorder));
    exit(2);
}
printf("\n Finished dx_mreciottdata\n");

/* Display termination condition value */
printf ("The termination value = %d\n", ATDX_TERMMASK(playerOne));
```

record voice data from two TDM bus time slots — dx_mreciottdata()

```
/* Close two play channels and one record channel */
if (dx_close(recorder) == -1){
    printf("Error closing recorder \n");
    printf("errno = %d\n", errno);
    exit(3);
}
if (dx_close(playerTwo) == -1 ){
    printf("Error closing playerTwo\n");
    printf("errno = %d\n", errno);
    exit (3);
}
if (dx_close(playerOne) == -1) {
    printf("Error closing playerOne\n");
    printf("errno = %d\n", errno);
    exit (3);
}
if (close(recordiott.io_fhandle) == -1){
    printf("File close error \n");
    exit(1);
}
if (close(playOneiott.io_fhandle) == -1){
    printf("File close error \n");
    exit(1);
}
if (close(playTwoiott.io_fhandle) == -1){
    printf("File close error \n");
    exit(1);
}
/* And finish */
return 1;
}
```

■ Example 2

The following example is for Windows[®] applications.

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

#include <stdio.h>
#include <stdlib.h>

#define MAXLEN 100000

/* Define logging macro */
#define log_rc(B, F) \
    printf (" %-60.60s: ", #B); \
    fflush (stdout); \
    retval = B; \
    printf ("RC=%d\n", retval); \
    if ( retval F ) { printf ("Fatal error!\n"); exit (1); }

main(int argc, char *argv[])
{
    int playerOne, playerTwo, recorder;
    DX_IOTT playOneiott={0}, playTwoiott={0}, recordiott={0};
    DV_TPT playOnetpt={0}, playTwotpt={0}, recordtpt ={0};
    DX_XPB recordxpb={0}, playOnexpb={0}, playTwoxpb={0};

    SC_TSINFO playOnetsinfo, playTwotsinfo, recordtsinfo;
    long playOnescts, playTwoscts;
    long mRectslots[32];
```

dx_mreciottdata() — record voice data from two TDM bus time slots

```
/* open two play channels and one record channel */
if ((playerOne = dx_open(argv[3], NULL)) == -1) {
    printf("Could not open %s\n", argv[3]);
    exit (1);
}

if ((playerTwo = dx_open(argv[4], NULL)) == -1) {
    printf("Could not open %s\n", argv[4]);
    exit (1);
}

if ((recorder = dx_open(argv[5], NULL)) == -1) {
    printf("Could not open %s\n", argv[5]);
    exit (1);
}

dx_clrtpt (&playOnetpt, 1);
dx_clrtpt (&playTwotpt, 1);
dx_clrtpt (&recordtpt, 1);

log_rc (playTwoiott.io_fhandle = dx_fileopen (argv[2], O_RDONLY|O_BINARY), == -1)
log_rc (playOneiott.io_fhandle = dx_fileopen (argv[1], O_RDONLY|O_BINARY), == -1)

playOneiott.io_type = IO_DEV | IO_EOT;
playOneiott.io_offset = 0;
playOneiott.io_length = -1;

playOnexpb.wFileFormat = FILE_FORMAT_VOX;
playOnexpb.wDataFormat = DATA_FORMAT_MULAW;
playOnexpb.nSamplesPerSec = DRT_8KHZ;
playOnexpb.wBitsPerSample = 8;

playTwoiott.io_type = IO_DEV | IO_EOT;
playTwoiott.io_offset = 0;
playTwoiott.io_length = -1;

playTwoxpb.wFileFormat = FILE_FORMAT_VOX;
playTwoxpb.wDataFormat = DATA_FORMAT_MULAW;
playTwoxpb.nSamplesPerSec = DRT_8KHZ;
playTwoxpb.wBitsPerSample = 8;

/*
 * Get channels' external time slots and fill in mRectslots[] array
 */
playOnetsinfo.sc_numts = 1;
playOnetsinfo.sc_tsarrayp = &playOnescts;
if (dx_getxmitslot (playerOne, &playOnetsinfo) == -1 ){
    /* Handle error */
}

playTwotsinfo.sc_numts = 1;
playTwotsinfo.sc_tsarrayp = &playTwoscts;
if (dx_getxmitslot (playerTwo, &playTwotsinfo) == -1 ) {
    /* Handle error */
}

mRectslots[1] = playTwoscts;
mRectslots[0] = playOnescts;

/* Set up SC_TSINFO structure */
recordtsinfo.sc_numts = 2;
recordtsinfo.sc_tsarrayp = &mRectslots[0];

log_rc (recordiott.io_fhandle = dx_fileopen(argv[6], O_RDWR|O_BINARY|O_CREAT), == -1);
recordiott.io_type = IO_EOT|IO_DEV;
recordiott.io_offset = 0;
recordiott.io_length = MAXLEN;
```

record voice data from two TDM bus time slots — dx_mreciottdata()

```
recordiott.io_bufp = 0;
recordiott.io_nextp = NULL;

recordxpb.wFileFormat = FILE_FORMAT_VOX;
recordxpb.wDataFormat = DATA_FORMAT_MULAW;
recordxpb.nSamplesPerSec = DRT_8KHZ;
recordxpb.wBitsPerSample = 8;

/* Play user-supplied files */
log_rc (dx_playiottdata(playerOne, &playOneiott, NULL, &playOnexpb, EV_ASYNC), ==-1)
log_rc (dx_playiottdata(playerTwo, &playTwoiott, NULL, &playTwoxpb, EV_ASYNC), ==-1)

/* And record from both play channels */
printf("\n Starting dx_mreciottdata");
if (dx_mreciottdata(recorder, &recordiott, NULL, &recordxpb, EV_SYNC|RM_TONE,
    &recordtsinfo) == -1) {
    printf("Error recording from dxxxB1C1 and dxxxB1C2\n");
    printf("error = %s\n", ATDV_ERRMSGP(recorder));
    exit(2);
}
printf("\n Finished dx_mreciottdata\n");

/* Display termination condition value */
printf ("The termination value = %d\n", ATDX_TERMMSK(playerOne));

/* Close two play channels and one record channel */
if (dx_close(recorder) == -1){
    printf("Error closing recorder \n");
    printf("errno = %d\n", errno);
    exit(3);
}
if (dx_close(playerTwo) == -1 ){
    printf("Error closing playerTwo\n");
    printf("errno = %d\n", errno);
    exit (3);
}
if (dx_close(playerOne) == -1) {
    printf("Error closing playerOne\n");
    printf("errno = %d\n", errno);
    exit (3);
}
if (dx_fileclose(recordiott.io_fhandle) == -1){
    printf("File close error \n");
    exit(1);
}
if (dx_fileclose(playOneiott.io_fhandle) == -1){
    printf("File close error \n");
    exit(1);
}
if (dx_fileclose(playTwoiott.io_fhandle) == -1){
    printf("File close error \n");
    exit(1);
}
/* And finish */
return 1;
}
```

■ See Also

- [dx_rec\(\)](#)
- [dx_play\(\)](#)
- [dx_reciottdata\(\)](#)
- [dx_playiottdata\(\)](#)

dx_open() — *open a voice device and return a unique device handle*

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: HMP Software, Springware boards

■ Description

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

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

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

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

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

■ Cautions

- Do not use the operating system **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.
- On HMP Software, two processes cannot open and access the same device. On Springware boards, a device can be opened more than once by any number of processes.
- In Linux, if STDOUT has been closed and a Dialogic® device is then opened, the device may get the same handle as STDOUT. Subsequent calls to **printf()** (which goes to STDOUT) may cause a kernel panic.
- On Springware boards in Linux, when developing an application for a large system (more than 350 devices), the application should open all the voice devices (board and/or channel) first, and then open all other devices.

■ Errors

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

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

EBADF

Invalid file descriptor

EINTR

A signal was caught

EINVAL

Invalid argument

EIO

Error during a Linux STREAMS open

This function will fail and return -1 if:

- The device name is invalid.
- A hardware error on the board or channel is discovered.

■ Example

This example illustrates how to open a channel device.

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

main()
{
    int chdev;          /* channel descriptor */
    .
    .
    .
}
```

dx_open() — *open a voice device and return a unique device handle*

```
/* Open Channel */
if ((chdev = dx_open("dxxxB1C1",0)) == -1) {
    /* process error */
}
.
.
}
```

■ See Also

- [dx_close\(\)](#)

`dx_OpenStreamBuffer()`

Name: `int dx_OpenStreamBuffer(BufferSize)`

Inputs: `int BufferSize` • size in bytes of circular stream buffer

Returns: stream buffer handle if successful
-1 if failure

Includes: `srllib.h`
`dxxplib.h`

Category: streaming to board

Mode: synchronous

Platform: HMP Software

■ Description

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

Parameter	Description
<code>BufferSize</code>	specifies the size in bytes of the circular stream buffer to allocate

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 *Dialogic® 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.
- Before calling `dx_OpenStreamBuffer()`, you must call `dx_open()` on a board, channel, or physical board. Otherwise, the DM3 library will not load, and `dx_OpenStreamBuffer()` will fail.

■ Errors

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

dx_OpenStreamBuffer() — create and initialize a circular stream buffer

Unlike other Dialogic[®] 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

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

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

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

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

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

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

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

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

■ See Also

- [dx_CloseStreamBuffer\(\)](#)
- [dx_SetWaterMark\(\)](#)

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

Returns: 0 if success
-1 if failure

Includes: srllib.h
dxxlib.h

Category: I/O

Mode: asynchronous or synchronous

Platform: HMP Software, Springware boards

■ Description

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

For a single file synchronous play, **dx_playf()** is more convenient because you do not have to set up a **DX_IOTT** structure. See the **dx_playf()** function description for more information.

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

Parameter	Description
chdev	Specifies the valid channel device handle obtained when the channel was opened using dx_open() .
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 502, for information about the data structure.
tptp	Points to the Termination Parameter Table structure, DV_TPT , which specifies termination conditions for playing. For more information on this structure, see DV_TPT , on page 481. <i>Note:</i> 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.

dx_play() — play recorded voice data

Parameter	Description
mode	<p>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).</p> <p>Choose one only:</p> <ul style="list-style-type: none">• EV_ASYNC – run asynchronously• EV_SYNC – run synchronously (default) <p>On HMP Software, choose one or more of the following:</p> <ul style="list-style-type: none">• 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 200 msec tone before initiating play <p>On Springware boards, choose one or more of the following:</p> <ul style="list-style-type: none">• 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.

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

Feature(s)	ADPCM (24 kbps)	ADPCM (32 kbps)	PCM (48 kbps)	PCM (64 kbps)
Tone	PM_TONE PM_SR6 {MD_ADPCM}	PM_TONE PM_SR8 {MD_ADPCM}	PM_TONE PM_ALAW* PM_SR6 MD_PCM	PM_TONE PM_ALAW* PM_SR8 MD_PCM
No Tone	PM_SR6 {MD_ADPCM}	PM_SR8 {MD_ADPCM}	PM_SR6 MD_PCM	PM_SR8 MD_PCM
{ } = Default modes. * = Select if file was encoded using A-law				

■ **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 Dialogic® 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.

dx_play() — *play recorded voice data*

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()`.
- If A-law encoding is selected (`PM_ALAW`), the A-law parameter must be passed each time the play function is called or the setting will return to mu-law (the default).
- On HMP Software, when playing a file that contains DTMFs, the same voice device might detect the DTMFs as incoming ones and process the DTMFs as a termination condition. The louder the recorded DTMFs in the file being played out, the more likely the chances of those DTMFs to be detected as incoming ones. It's been observed that the problem can be avoided if the amplitude of the DTMFs being played is below -6.5 dB; but this should only be taken as a guideline since environment conditions are also a factor.

■ **Errors**

If the function returns -1, use the Dialogic® 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_BADIOTT`
Invalid `DX_IOTT` entry

`EDX_BADTPT`
Invalid `DV_TPT` entry

`EDX_BUSY`
Busy executing I/O function

`EDX_SYSTEM`
Error from operating system

■ **Example 1**

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

```
/* Play a voice file. Terminate on receiving 4 digits or at end of file */
#include <fcntl.h>
#include <srllib.h>
#include <dxxxlib.h>
```

```
main()
{
    int      chdev;
    DX_IOTT  iott;
    DV_TPT   tpt;
    DV_DIGIT dig;
    .
    .

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

    /* set up DX_IOTT */
    iott.io_type = IO_DEV|IO_EOT;
    iott.io_bufp = 0;
    iott.io_offset = 0;
    iott.io_length = -1; /* play till end of file */
    if((iott.io_fhandle = dx_fileopen("prompt.vox", O_RDONLY|O_BINARY))
        == -1) {
        /* process error */
    }

    /* set up DV_TPT */
    dx_clrtpt(&tpt,1);
    tpt.tp_type = IO_EOT; /* only entry in the table */
    tpt.tp_termno = DX_MAXDTMF; /* Maximum digits */
    tpt.tp_length = 4; /* terminate on four digits */
    tpt.tp_flags = TF_MAXDTMF; /* Use the default flags */

    /* 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>

#define MAXCHAN 24

int play_handler();
DX_IOTT prompt[MAXCHAN];
DV_TPT tpt;
DV_DIGIT dig;
```

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

```
main()
{
    int chdev[MAXCHAN], index, index1;
    char *chname;
    int i, srlmode, voxfd;

    /* Set SRL to run in polled mode. */
    srlmode = SR_POLLMODE;
    if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1) {
        /* process error */
    }

    /* initialize all the DX_IOTT structures for each individual prompt */
    .
    .

    /* For Windows applications: open the vox file to play; the file descriptor will be used
     * by all channels.
     */
    if ((voxfd = dx_fileopen("prompt.vox", O_RDONLY|O_BINARY)) == -1) {
        /* process error */
    }

    /* For Linux applications, open the vox file to play; the file descriptor will be used
     * by all channels.
     */
    if ((voxfd = open("prompt.vox", O_RDONLY)) == -1) {
        /* process error */
    }

    /* For each channel, open the device using dx_open(), set up a DX_IOTT
     * structure for each channel, and issue dx_play() in asynchronous mode. */
    for (i=0; i<MAXCHAN; i++) {

        /* Set chname to the channel name, e.g., 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_enbhdr() to set up handler function to handle play
         * completion events on this channel.
         */
        if (sr_enbhdr(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_clrtppt(&tpt, 1);
        tpt.tp_type = IO_EOT; /* only entry in the table */
        tpt.tp_termno = DX_MAXDTMF; /* Maximum digits */
        tpt.tp_length = 1; /* terminate on the first digit */
        tpt.tp_flags = TF_MAXDTMF; /* Use the default flags */
        prompt[i].io_type = IO_DEV|IO_EOT; /* play from file */
        prompt[i].io_bufp = 0;
        prompt[i].io_offset = 0;
        prompt[i].io_length = -1; /* play till end of file */
        prompt[i].io_nextp = NULL;
        prompt[i].io_fhandle = voxfd;
    }
}
```



```
/* play the data */
if (dx_play(chdev[i], &prompt[i], &tpt, EV_ASYNC) == -1) {
    /* process error */
}
}

/* Use sr_waitevt to wait for the completion of dx_play().
 * On receiving the completion event, TDX_PLAY, control is transferred
 * to the handler function previously established using sr_enbhdrlr().
 */
.
.
}

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

For Springware boards on Windows applications, 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 <dxlib.h>
#include <windows.h>

int parm;
DV_TPT tpt[2];
DV_DIGIT digit;
TN_GEN tngen;
DX_IOTT iott;

main(argc, argv)
    int argc;
    char* argv[];
{
    int chfd;
    char channame[12];
    parm = SR_POLLMODE;
    sr_setparm(SRL_DEVICE, SR_MODEID, &parm);

    /*
     * Open the channel using the command line arguments as input
     */
    sprintf(channame, "%sC%s", argv[1], argv[2]);

    if ((chfd = dx_open(channame, NULL)) == -1) {
        printf("Board open failed on device %s\n", channame);
    }
}
```

dx_play() — *play recorded voice data*

```
        exit(1);
    }
    printf("Devices open and waiting .....\\n");

    /*
    * Take the phone off-hook to talk to the ADSI phone
    * This assumes we are connected through a Skutch Box.
    */

    if (dx_sethook( chfd, DX_OFFHOOK, EV_SYNC) == -1) {
        printf("sethook failed\\n");
        while (1) {
            sleep(5);
            dx_clrdigbuf( chfd );
            printf("Digit buffer cleared ..\\n");

            /*
            * Generate the alert tone
            */
            iott.io_type =IO_DEV|IO_EOT;
            iott.io_fhandle = dx_fileopen("message.asc",O_RDONLY);
            iott.io_length = -1;
            parm = DM_D

            if (dx_setparm (chfd, DXCH_DTINITSET, (void *)parm) ==-1){
                printf ("dx_setparm on DTINITSET failed\\n");
                exit(1);
            }

            if (dx_play(chfd,&iott,(DV_TPT *)NULL, PM_ADSSIALERT|EV_SYNC) ==-1) {
                printf("dx_play on the ADSI file failed\\n");
                exit(1);
            }
        }
    }
    dx_close(chfd);
    exit(0);
}
```

■ See Also

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

`dx_playf()`

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

Inputs:

<code>int chdev</code>	• valid channel device handle
<code>char *fnamep</code>	• pointer to name of file to play
<code>DV_TPT *tptp</code>	• pointer to Termination Parameter Table structure
<code>unsigned short mode</code>	• playing mode bit mask for this play session

Returns: 0 if success
-1 if failure

Includes: `srllib.h`
`dxxlib.h`

Category: I/O Convenience

Mode: synchronous

Platform: HMP Software, Springware boards

■ Description

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

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

Parameter	Description
<code>chdev</code>	specifies the valid channel device handle obtained when the channel was opened using <code>dx_open()</code>
<code>fnamep</code>	points to the file from which voice data will be played
<code>tptp</code>	points to the Termination Parameter Table structure, <code>DV_TPT</code> , which specifies termination conditions for playing. For more information on this structure, see <code>DV_TPT</code> , on page 481.
<code>mode</code>	specifies the mode. This function supports <code>EV_SYNC</code> (synchronous mode) only.

■ Cautions

On HMP Software, when playing a file that contains DTMFs, the same voice device might detect the DTMFs as incoming ones and process the DTMFs as a termination condition. The louder the recorded DTMFs in the file being played out, the more likely the chances of those DTMFs to be detected as incoming ones. It's been observed that the problem can be avoided if the amplitude of the DTMFs being played is below -6.5 dB; but this should only be taken as a guideline since environment conditions are also a factor.

dx_playf() — *synchronously play voice data*

■ Errors

If the function returns -1, use the Dialogic[®] 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_BADIOTT
Invalid **DX_IOTT** entry

EDX_BADTPT
Invalid **DX_TPT** entry

EDX_BUSY
Busy executing I/O function

EDX_SYSTEM
Error from operating system

■ Source Code

```
/*
 * NAME: int dx_playf(devd,filep,tptp,mode)
 * DESCRIPTION: This function opens and plays a
 *              named file.
 * INPUTS: devd - channel descriptor
 *          tptp - pointer to the termination control block
 *          filep - pointer to file name
 * OUTPUTS: Data is played.
 * RETURNS: 0 - success -1 - failure
 * CALLS: open() dx_play() close()
 * CAUTIONS: none.
 */
*****/

int dx_playf(devd,filep,tptp,mode)
int devd;
char *filep;
DV_TPT *tptp;
USHORT mode;

{
    DX_IOTT iott;
    int rval;

    /*
     * If Async then return Error
     * Reason: IOTT's must be in scope for the duration of the play
     */
    if ( mode & EV_ASYNC ) {
        return( -1 );
    }

    /* Open the File */
    if ((iott.io_fhandle = open(filep,O_RDONLY)) == -1) {
        return -1;
    }
}
```

```
/* Use dx_play() to do the Play */
iott.io_type = IO_EOT | IO_DEV;
iott.io_offset = (unsigned long)0;
iott.io_length = -1;

rval = dx_play(devd,&iott,tptp,mode);

if (close(iott.io_fhandle) == -1) {
    return -1;
}

return rval;
}
```

■ Example

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

main()
{
    int chdev;
    DV_TPT tpt[2];

    /* Open the channel using dx_open( ). Get channel device descriptor in
     * chdev.
     */
    if ((chdev = dx_open("dxxxxB1C1",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: HMP Software, Springware boards

■ **Description**

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

The file format for the files to be played is specified in the **wFileFormat** field of the **DX_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.

play back recorded voice data from multiple sources — dx_playiottdata()

Parameter	Description
chdev	Specifies the valid channel device handle obtained when the channel was opened using dx_open() .
iottp	Points to the I/O Transfer Table structure, DX_IOTT , which specifies the order of playback and the location of voice data. See DX_IOTT , on page 502, 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 481.
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 514. For information about supported data formats, see the <i>Dialogic® Voice API Programming Guide</i> .
mode	Specifies the play mode and synchronous/asynchronous mode. For a list of all valid values, see the dx_play() function description. <ul style="list-style-type: none">• PM_TONE – transmit a 200 msec tone before initiating play• 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.
- On HMP Software, playing an empty WAVE file results in an invalid offset error. To play a silent WAVE file successfully, ensure that there is at least one byte of silence data (0xFF) in the payload.
- When set to play WAVE files, all other fields in the [DX_XPB](#) are ignored.
- When set to play WAVE files, this function will fail if an unsupported data format is attempted to be played. For information about supported data formats, see the description for [DX_XPB](#) and the *Dialogic® Voice API Programming Guide*.
- On HMP Software, when playing a file that contains DTMFs, the same voice device might detect the DTMFs as incoming ones and process the DTMFs as a termination condition. The louder the recorded DTMFs in the file being played out, the more likely the chances of those DTMFs to be detected as incoming ones. It's been observed that the problem can be avoided if

***dx_playiottdata()* — play back recorded voice data from multiple sources**

the amplitude of the DTMFs being played is below -6.5 dB; but this should only be taken as a guideline since environment conditions are also a factor.

■ **Errors**

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

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

EDX_BADIOTT
Invalid **DX_IOTT** setting

EDX_BADWAVFILE
Invalid WAVE file

EDX_BUSY
Channel is busy

EDX_SH_BADCMD
Unsupported command or WAVE file format

EDX_SYSTEM
Error from operating system

EDX_XBPARM
Invalid **DX_XPB** setting

■ **Example**

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

```
#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 */
    .
    .
    .
}
```


play back recorded voice data from multiple sources — dx_playiottdata()

```
/* 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;

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

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

/* Set up DX_IOTT */
iott.io_fhandle = fd;
iott.io_bufp = 0;
iott.io_offset = 0;
iott.io_length = -1;
iott.io_type = IO_DEV | IO_EOT;

/*
 * Specify VOX file format for ADPCM at 8KHz
 */
xpb.wFileFormat = FILE_FORMAT_VOX;
xpb.wDataFormat = DATA_FORMAT_DIALOGIC_ADPCM;
xpb.nSamplesPerSec = DRT_8KHZ;
xpb.wBitsPerSample = 4;

/* Wait forever for phone to ring and go offhook */
if (dx_wtring(chdev,1,DX_OFFHOOK,-1) == -1) {
    printf("Error waiting for ring - %s\n", ATDV_LASTERR(chdev));
    exit(3);
}

/* Start playback */
if (dx_playiottdata(chdev,&iott,&tpt,&xpb,EV_SYNC)==-1) {
    printf("Error playing file - %s\n", ATDV_ERRMSGP(chdev));
    exit(4);
}
}
```

■ See Also

- [dx_play\(\)](#)
- [dx_playf\(\)](#)
- [dx_playwav\(\)](#)
- [dx_playvox\(\)](#)
- [dx_setuio\(\)](#)

dx_playtone()

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: HMP Software, Springware boards

■ 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.

Parameter	Description
chdev	specifies the valid channel device handle obtained when the channel was opened using dx_open()
tngenp	points to the TN_GEN structure, which defines the frequency, amplitude, and duration of a single- or dual-frequency tone. For more information, see TN_GEN , on page 524. You can use the dx_bldtngen() function to set up the structure.
tptp	points to the DV_TPT data structure, which specifies a terminating condition for this function. For more information, see DV_TPT , on page 481.
mode	specifies whether to run this function asynchronously or synchronously. Set to one of the following: <ul style="list-style-type: none">• EV_ASYNC – asynchronous mode• EV_SYNC – synchronous mode (default)

■ Asynchronous Operation

To run this function asynchronously, set the **mode** parameter to **EV_ASYNC**. This function returns 0 to indicate it has initiated successfully, and generates a **TDX_PLAYTONE** termination event to indicate completion. Use the Dialogic® Standard Runtime Library (SRL) Event Management functions to handle the termination event; see the *Dialogic® Standard Runtime Library API Library Reference* for more information.

play tone defined by TN_GEN structure — dx_playtone()

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

■ Synchronous Operation

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

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

■ Cautions

- The channel must be idle when calling this function.
- If the tone generation template contains an invalid `tg_dflag`, or the specified amplitude or frequency is outside the valid range, `dx_playtone()` will generate a `TDX_ERROR` event if asynchronous, or -1 if synchronous.
- On HMP Software, 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_BADPARAM`

Invalid parameter

`EDX_BADPROD`

Function not supported on this board

`EDX_BADTPT`

Invalid `DV_TPT` entry

`EDX_BUSY`

Busy executing I/O function

`EDX_FLAGGEN`

Invalid `tn_dflag` field in `TN_GEN` structure

`EDX_FREQGEN`

Invalid frequency component in `TN_GEN` structure

`EDX_SYSTEM`

Error from operating system

***dx_playtone()* — play tone defined by *TN_GEN* structure**

■ Example

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

#define TID_1 101

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

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

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

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

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

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

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

play tone defined by TN_GEN structure — dx_playtone()

```
tpt[0].tp_data = DX_TONEON;

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

tpt[2].tp_type = IO_EOT;
tpt[2].tp_termno = DX_MAXTIME; /* On HMP Software, DX_MAXTIME not supported */
tpt[2].tp_length = 6000;
tpt[2].tp_flags = TF_MAXTIME;

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

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

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

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

■ See Also

- [dx_bldtngen\(\)](#)
- [TN_GEN](#) data structure
- global tone generation topic in *Dialogic® Voice API Programming Guide*
- event management functions in *Dialogic® Standard Runtime Library API Library Reference*
- [DV_TPT](#) data structure (to specify a termination condition)
- [ATDX_TERMMSK\(\)](#)

dx_playtoneEx() — *play the cadenced tone defined by TN_GENCAD*

dx_playtoneEx()

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

Inputs:

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

Returns: 0 if success
-1 if failure

Includes: srllib.h
dxxxlib.h

Category: Global Tone Generation

Mode: asynchronous or synchronous

Platform: HMP Software, Springware boards

■ 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()**. On HMP Software, valid values 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).

play the cadenced tone defined by TN_GENCAD — dx_playtoneEx()

Parameter	Description
chdev	specifies the valid channel device handle obtained when the channel was opened using dx_open()
tnrencadp	<p>On HMP Software, points to a TN_GENCAD structure (which defines a signal by specifying a cycle and its number of repetitions).</p> <p>On Springware boards, points to a TN_GENCAD structure (which defines a signal by specifying a cycle and its number of repetitions), or specifies one of the following predefined, standard, PBX call progress signals:</p> <ul style="list-style-type: none">• CP_DIAL – dial tone• CP_REORDER – reorder tone (paths-busy, all-trunks-busy, fast busy)• CP_BUSY – busy tone (slow busy)• CP_RINGBACK1 – audible ring tone 1 (ringback tone)• CP_RINGBACK2 – audible ring tone 2 (slow ringback tone)• CP_RINGBACK1_CALLWAIT – special audible ring tone 1• CP_RINGBACK2_CALLWAIT – special audible ring tone 2• CP_RECALL_DIAL – recall dial tone• CP_INTERCEPT – intercept tone• CP_CALLWAIT1 – call waiting tone 1• CP_CALLWAIT2 – call waiting tone 2• CP_BUSY_VERIFY_A – busy verification tone (Part A)• CP_BUSY_VERIFY_B – busy verification tone (Part B)• CP_EXEC_OVERRIDE – executive override tone• CP_FEATURE_CONFIRM – confirmation tone• CP_STUTTER_DIAL – Stutter dial tone (same as message waiting dial tone)• CP_MSG_WAIT_DIAL – message waiting dial tone (same as stutter dial tone)
tptp	points to the DV_TPT data structure, which specifies one or more terminating conditions for this function. For more information on this structure, see DV_TPT , on page 481.
mode	specifies whether to run this function asynchronously or synchronously. Set to one of the following: <ul style="list-style-type: none">• EV_ASYNC – asynchronous mode• EV_SYNC – synchronous mode (default)

To run this function asynchronously, set the **mode** parameter to [EV_ASYNC](#). When running asynchronously, this function will return 0 to indicate that it has initiated successfully, and will generate a [TDX_PLAYTONE](#) termination event to indicate successful termination.

By default, this function will run synchronously, and will return a 0 to indicate successful termination of synchronous play.

■ **Cautions**

- The channel must be idle when calling this function.

dx_playtoneEx() — play the cadenced tone defined by TN_GENCAD

- 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 HMP Software, the **DX_MAXTIME** termination condition is not supported by tone generation functions, which include **dx_playtoneEx()**.

■ **Errors**

If the function returns -1, use the Dialogic® 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_BADPARAM

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

■ **Example**

```
/*$ dx_playtoneEx( ) example $*/

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

main()
{
    TN_GEN      tngen;
    TN_GENCAD   tngencad;
    DV_TPT      tpt[ 2 ];
    int         dxxxdev;
    long        term;

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


play the cadenced tone defined by TN_GENCAD — dx_playtoneEx()

```
/*
 * 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 HMP, DX_MAXTIME not supported */
tpt[1].tp_length = 450;
tpt[1].tp_flags = TF_MAXTIME;

/*
 * Build a custom cadence dial tone to indicate that a priority message is waiting.
 * Signal cycle has 4 segments & repeats forever (cycles=255) until tpt termination:
 * Note that cycles = 255 is supported on Springware but not on HMP Software.
 * 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 dial tone.
 */
if (dx_playtoneEx( dxxxxdev, &tngencad, tpt, EV_SYNC ) == -1 ) {
    printf( "Unable to Play the Cadenced Tone\n" );
    printf( "Lasterror = %d Err Msg = %s\n",
        ATDV_LASTERR( dxxxxdev ), ATDV_ERRMSGP( dxxxxdev ) );
    dx_close( dxxxxdev );
    exit( 1 );
}

/*
 * Examine termination reason in bitmap.
 * If time-out caused termination, play reorder tone.
 */
if( (term = ATDX_TERMSK(dxxxxdev)) == 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( dxxxxdev, CP_REORDER, tpt, EV_SYNC ) == -1 ) {
        printf( "Unable to Play the Cadenced Tone\n" );
        printf( "Lasterror = %d Err Msg = %s\n",
```

dx_playtoneEx() — *play the cadenced tone defined by TN_GENCAD*

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

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

■ See Also

- [dx_playtone\(\)](#)
- [dx_bldtngen\(\)](#)
- [TN_GEN](#) data structure
- [TN_GENCAD](#) data structure

dx_playvox()

Name: int dx_playvox(chdev, filenamep, tptp, xpbp, mode)

Inputs:

int chdev	• valid channel device handle
char *filenamep	• pointer to name of file to play
DV_TPT *tptp	• pointer to Termination Parameter Table structure
DX_XPB *xpbp	• pointer to I/O Transfer parameter block structure
unsigned short mode	• play mode

Returns: 0 if successful
-1 if failure

Includes: srllib.h
dxxlib.h

Category: I/O Convenience

Mode: synchronous

Platform: HMP Software, Springware boards

■ Description

The **dx_playvox()** convenience function plays voice data stored in a single VOX file. This function calls **dx_playiottdata()**.

Parameter	Description
chdev	specifies the valid channel device handle obtained when the channel was opened using dx_open()
filenamep	points to name of VOX file to play
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 481.
xpbp	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 514. If xpbp is set to NULL, this function interprets the data as 6 kHz linear ADPCM.
mode	specifies the play mode. The following two values can be used individually or ORed together: <ul style="list-style-type: none">• PM_TONE – transmit a 200 msec tone before initiating play• EV_SYNC – synchronous operation (must be specified)

***dx_playvox()* — play voice data stored in a single VOX file**

■ Cautions

When playing or recording VOX files, the data format is specified in DX_XPB rather than through the mode parameter of **dx_playvox()**.

■ Errors

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

EDX_BADIOTT
Invalid **DX_IOTT** setting

EDX_BADWAVFILE
Invalid WAVE file

EDX_BUSY
Channel is busy

EDX_SH_BADCMD
Unsupported command or WAVE file format

EDX_SYSTEM
Error from operating system

EDX_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);
    }
}
```

play voice data stored in a single VOX file — dx_playvox()

```
/* Start 6KHz ADPCM playback */
if (dx_playvox(chdev, "HELLO.VOX", &tp, NULL, EV_SYNC) == -1) {
    printf("Error playing file - %s\n", ATDV_ERRMSGP(chdev));
    exit(4);
}
}
```

■ See Also

- [dx_play\(\)](#)
- [dx_playf\(\)](#)
- [dx_playiottdata\(\)](#)
- [dx_playwav\(\)](#)

dx_playwav() — *play voice data stored in a single WAVE file*

dx_playwav()

Name: int dx_playwav(chdev, filenamep, tptp, mode)

Inputs:

int chdev	• valid channel device handle
char *filenamep	• pointer to name of file to play
DV_TPT *tptp	• pointer to Termination Parameter Table structure
unsigned short mode	• play mode

Returns: 0 if successful
-1 if failure

Includes: srllib.h
dxxxlib.h

Category: I/O Convenience

Mode: synchronous

Platform: HMP Software, Springware boards

■ 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.

Parameter	Description
chdev	specifies the valid channel device handle obtained when the channel was opened using dx_open()
tptp	points to the Termination Parameter Table structure, DV_TPT, which specifies termination conditions for playing. For more information on this function, see DV_TPT , on page 481.
filenamep	points to the name of the file to play
mode	specifies the play mode. The following two values can be used individually or ORed together: <ul style="list-style-type: none">• PM_TONE – transmit a 200 msec tone before initiating play• EV_SYNC – synchronous operation (must be specified)

■ Cautions

This function fails when an unsupported WAVE file format is attempted to be played. For information on supported data formats, see the description for **DX_XPB**, on page 514 and the *Voice API Programming Guide*.

■ Errors

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

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

■ Example

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

main()
{
    int chdev;          /* channel descriptor */
    DV_TPT tpt;        /* termination parameter table */
    .
    .
    .

    /* Open channel */
    if ((chdev = dx_open("dxxxB1C1",0)) == -1) {
        printf("Cannot open channel\n");
        /* Perform system error processing */
        exit(1);
    }

    /* Set to terminate play on 1 digit */
    tpt.tp_type = IO_EOT;
    tpt.tp_termno = DX_MAXDTMF;
    tpt.tp_length = 1;
    tpt.tp_flags = TF_MAXDTMF;

    /* Wait forever for phone to ring and go offhook */
    if (dx_wtring(chdev,1,DX_OFFHOOK,-1) == -1) {
        printf("Error waiting for ring - %s\n", ATDV_LASTERR(chdev));
        exit(3);
    }

    /* Start playback */
    if (dx_playwav(chdev,"HELLO.WAV",&tpt,EV_SYNC) == -1) {
        printf("Error playing file - %s\n", ATDV_ERRMSGP(chdev));
        exit(4);
    }
}
```

dx_playwav() — *play voice data stored in a single WAVE file*

}

■ **See Also**

- [dx_playiottdata\(\)](#)
- [dx_playvox\(\)](#)

dx_PutStreamData()

Name: int dx_PutStreamData(hBuffer, pNewData, BuffSize, flag)

Inputs: int hBuffer • stream buffer handle
char* pNewData • pointer to user buffer of data to place in the stream buffer
int BuffSize • number of bytes in the user buffer
int flag • flag indicating last block of data

Returns: 0 if successful
-1 if failure

Includes: srllib.h
dxxlib.h

Category: streaming to board

Mode: synchronous

Platform: HMP Software

■ Description

The **dx_PutStreamData()** function puts data into the specified circular stream buffer. If there is not enough room in the buffer (an overrun condition), an error of -1 is returned and none of the data will be placed in the stream buffer. Writing 0 bytes of data to the buffer is not considered an error. The flag field is used to indicate that this is the last block of data. Set this flag to **STREAM_CONT** (0) for all buffers except the last one, which should be set to **STREAM_EOD** (1). This function can be called at any time between the opening and closing of the stream buffer.

Parameter	Description
hBuffer	specifies the circular stream buffer handle obtained from dx_OpenStreamBuffer()
pNewData	a pointer to the user buffer containing data to be placed in the circular stream buffer
BuffSize	specifies the number of bytes in the user buffer
flag	a flag indicating whether this is the last block of data in the user buffer. Valid values are: <ul style="list-style-type: none">• STREAM_CONT – for all buffers except the last one• STREAM_EOD – for the last buffer

■ Cautions

None.

■ Errors

If there is not enough room in the buffer (an overrun condition), this function returns an error of -1.

***dx_PutStreamData()* — place data into a circular stream buffer**

Unlike other Dialogic[®] 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**

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

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

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

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

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

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

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

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

■ **See Also**

- [**dx_OpenStreamBuffer\(\)**](#)

dx_querytone()

Name: int dx_querytone(brdhdl, toneid, tonedata, mode)

Inputs:

int brdhdl	• a valid 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
dxxplib.h

Category: Call Progress Analysis

Mode: asynchronous or synchronous

Platform: HMP Software

■ Description

The **dx_querytone()** function returns tone information for a call progress tone currently available on the board device. On successful completion of the function, the **TONE_DATA** structure contains the relevant tone information.

Before creating a new tone definition with **dx_createtone()**, first use **dx_querytone()** to get tone information for the tone ID, then use **dx_deletetone()** to delete that same tone ID. Only tones listed in the **toneid** parameter description are supported for this function. For more information on modifying call progress analysis tone definitions, see the *Dialogic® Voice API Programming Guide*.

When running in asynchronous mode, this 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 in synchronous mode and returns 0 to indicate completion.

dx_querytone() — get tone information for a specific call progress tone

Parameter	Description
brdhdl	<p>specifies a valid board device handle (not a virtual board device) of the format brdBn obtained by a call to dx_open().</p> <p>To get the board name, use the SRLGetPhysicalBoardName() function. This function and other device mapper functions return information about the structure of the system. For more information, see the <i>Dialogic® Standard Runtime Library API Library Reference</i>.</p>
toneid	<p>specifies the tone ID of the call progress tone. Valid values are:</p> <ul style="list-style-type: none">• TID_BUSY1• TID_BUSY2• TID_DIAL_INTL• TID_DIAL_LCL• TID_DISCONNECT• TID_FAX1• TID_FAX2• TID_RNGBK1• TID_RNGBK2• TID_SIT_NC• TID_SIT_IC• TID_SIT_VC• TID_SIT_RO <p><i>Note:</i> The following tone IDs are not supported by this function: TID_SIT_ANY, TID_SIT_NO_CIRCUIT_INTERLATA, TID_SIT_REORDER_TONE_INTERLATA, and TID_SIT_INEFFECTIVE_OTHER.</p>
tonedata	<p>specifies a pointer to the TONE_DATA data structure that contains the tone information for the call progress tone identified by toneid</p>
mode	<p>specifies the mode in which the function will run. Valid values are:</p> <ul style="list-style-type: none">• EV_ASYNC – asynchronous mode• EV_SYNC – synchronous mode (default)

■ **Cautions**

- Only the default call progress tones as listed in the **toneid** parameter description are supported for this function. The following tone IDs are not supported by this function: TID_SIT_ANY, TID_SIT_NO_CIRCUIT_INTERLATA, TID_SIT_REORDER_TONE_INTERLATA, and TID_SIT_INEFFECTIVE_OTHER.
- To modify a default tone definition, use the three functions **dx_querytone()**, **dx_deletetone()**, and **dx_createtone()** in this order, for one tone at a time.
- When **dx_querytone()** is issued on a board device in asynchronous mode, and the function is immediately followed by another similar call prior to completion of the previous call on the same device, the subsequent call will fail with device busy.

■ **Errors**

If the function returns -1, use the Dialogic® 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_SYSTEM
error from operating system
- EDX_TONEID
bad tone template ID

■ **Example**

```
#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: HMP Software, Springware boards

■ 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.

Parameter	Description
chdev	specifies the valid channel device handle obtained when the channel was opened using dx_open()
iottp	points to the I/O Transfer Table Structure, DX_IOTT , which specifies the order 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 502, for more information on this data structure.

Parameter	Description
tptp	<p>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 481.</p> <p>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.</p>
mode	<p>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).</p> <p>Choose one only:</p> <ul style="list-style-type: none"> • EV_ASYNC – run asynchronously • EV_SYNC – run synchronously (default) <p>Choose one or more:</p> <ul style="list-style-type: none"> • 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 200 msec tone before initiating record • 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_USERTONE – (Springware boards Linux only) Play a user-defined tone before initiating record. If RM_USERTONE is not set but RM_TONE is set, the built-in tone will be played prior to initiating a record.

- Notes:**
1. If both MD_ADPCM and MD_PCM are set, MD_PCM will take precedence. If both MD_GAIN and MD_NOGAIN are set, MD_NOGAIN will take precedence. If both RM_TONE and NULL are set, RM_TONE takes precedence. If both RM_SR6 and RM_SR8 are set, RM_SR6 will take precedence.
 2. Specifying RM_SR6 or RM_SR8 in mode changes the setting of the parameter DXCH_RECRDRATE. DXCH_RECRDRATE can also be set and queried using [dx_setparm\(\)](#) and [dx_getparm\(\)](#). The default setting for DXCH_RECRDRATE is 6 kHz.
 3. The rate specified in the last record function will apply to the next record function, unless the rate was changed in the parameter DXCH_RECRDRATE using [dx_setparm\(\)](#).
 4. When using the RM_TONE bit for tone-initiated record, each time slot must be “listening” to the transmit time slot of the recording channel because the alert tone can only be transmitted on the recording channel transmit time slot.

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

***dx_rec()* — record voice data from a single channel**

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

Feature	ADPCM (24 kbps)	ADPCM (32 kbps)	PCM (48 kbps)	PCM (64 kbps)
AGC No Tone	RM_SR6 {MD_ADPCM} {MD_GAIN}	RM_SR8 {MD_ADPCM} {MD_GAIN}	RM_SR6 RM_ALAW* MD_PCM {MD_GAIN}	RM_SR8 RM_ALAW* MD_PCM {MD_GAIN}
No AGC No Tone	MD_NOGAIN RM_SR6 {MD_ADPCM}	MD_NOGAIN RM_SR8 {MD_ADPCM}	MD_NOGAIN RM_SR6 MD_PCM	MD_NOGAIN RM_SR8 MD_PCM
AGC Tone	RM_TONE RM_SR6 {MD_ADPCM} {MD_GAIN}	RM_TONE RM_SR8 {MD_ADPCM} {MD_GAIN}	RM_TONE RM_ALAW* RM_SR6 MD_PCM {MD_GAIN}	RM_TONE RM_ALAW* RM_SR8 MD_PCM {MD_GAIN}
No AGC Tone	MD_NOGAIN RM_TONE RM_SR6 {MD_ADPCM}	MD_NOGAIN RM_TONE RM_SR8 {MD_ADPCM}	MD_NOGAIN MD_PCM RM_SR6 RM_TONE RM_ALAW*	MD_NOGAIN MD_PCM RM_SR8 RM_TONE RM_ALAW*
{} = Default modes. * = Select if A-law encoding is required				

■ 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 Dialogic® 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

- 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).
- On HMP Software, voice channels must be listening to a TDM bus time slot in order for voice recording functions, such as **dx_rec()**, to work. The actual recording operation will start only after the voice channel is listening to the proper external time slot. In other words, you must issue a **dx_listen()** function call on the device handle before calling a voice recording function for that device handle, and the **dx_listen()** must be called from the same process as the voice recording function. If not, the voice recording function will return TDX_ERROR with EDX_SH_MISSING as the termination reason.
- 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.
- It is recommended that you start recording before receiving any incoming data on the channel so that initial data is not missed in the recording.

■ Errors

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

EDX_BADDEV
Invalid Device Descriptor

EDX_BADIOTT
Invalid [DX_IOTT](#) entry

EDX_BADPARAM
Invalid parameter

EDX_BADTPT
Invalid [DX_TPT](#) entry

EDX_BUSY
Busy executing I/O function

EDX_SYSTEM
Error from operating system

dx_rec() — record voice data from a single channel

■ Example 1

This example illustrates how to using ***dx_rec()*** in synchronous mode.

```
#include <fcntl.h>
#include <srllib.h>
#include <dxxxxlib.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("dxxxxB1C1",NULL)) == -1) {
        /* process error */
    }

    /*
     * Set up the DV_TPT structures for MAXDTMF
     */
    dx_clrtpt(&tpt,1);
    tpt.tp_type = IO_EOT;          /* last entry in the table */
    tpt.tp_termno = DX_MAXDTMF;   /* Maximum digits */
    tpt.tp_length = 1;           /* terminate on the first digit */
    tpt.tp_flags = TF_MAXDTMF;   /* Use the default flags */

    /*
     * Set up the DX_IOTT. The application records the voice data to memory
     * allocated by the user.
     */

    iott[0].io_type = IO_MEM|IO_CONT; /* Record to memory */
    iott[0].io_bufp = basebufp;      /* Set up pointer to buffer */
    iott[0].io_offset = 0;           /* Start at beginning of buffer */
    iott[0].io_length = MAXLEN;     /* Record 10,000 bytes of voice data */

    iott[1].io_type = IO_DEV|IO_EOT; /* Record to file, last DX_IOTT entry */
    iott[1].io_bufp = 0;             /* Set up pointer to buffer */
    iott[1].io_offset = 0;           /* Start at beginning of buffer */
    iott[1].io_length = MAXLEN;     /* Record 10,000 bytes of voice data */

    /* For Windows applications */
    if((iott[1].io_fhandle = dx_fileopen("file.vox",
        O_RDWR|O_CREAT|O_TRUNC|O_BINARY,0666)) == -1) {
        /* process error */
    }

    /* For Linux applications */
    if((iott[1].io_fhandle = open("file.vox", O_RDWR|O_CREAT|O_TRUNC,
        0666)) == -1) {
        /* process error */
    }

    /* clear previously entered digits */
    if (dx_clrdigbuf(chdev) == -1) {
        /* process error */
    }
    if (dx_rec(chdev,&iott[0],&tpt,RM_TONE|EV_SYNC) == -1) {
        /* process error */
    }
}
```

```

    }
    /* Analyze the data recorded */
    .
    .
}

```

■ Example 2

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

```

#include <stdio.h>
#include <fcntl.h>
#include <srllib.h>
#include <dxxxxlib.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_enbhdr(), set up handler function to handle record
        * completion events on this channel.
        */
        if (sr_enbhdr(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 */
    }
}

```

dx_rec() — record voice data from a single channel

```
/*
 * Set up the DX_IOTT. The application records the voice data to memory
 * allocated by the user.
 */
iott[i].io_type = IO_MEM|IO_EOT; /* Record to memory, last DX_IOTT
 * entry */
iott[i].io_bufp = basebufp[i]; /* Set up pointer to buffer */
iott[i].io_offset = 0; /* Start at beginning of buffer */
iott[i].io_length = MAXLEN; /* Record 10,000 bytes voice data */

/* clear previously entered digits */
if (dx_clrdigbuf(chdev) == -1) {
    /* process error */
}

/* Start asynchronous dx_rec() on the channel */
if (dx_rec(chdev[i], &iott[i], &tpt, RM_TONE|EV_ASYNC) == -1) {
    /* process error */
}
}

/* Use sr_waitevt to wait for the completion of dx_rec().
 * On receiving the completion event, TDX_RECORD, control is transferred
 * to a handler function previously established using sr_enbhdr().
 */
.
.
}

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 *Dialogic® Standard Runtime Library API Library Reference*
- [ATDX_TERMMSK\(\)](#)

record voice data from a single channel — dx_rec()

- [DV_TPT](#) data structure (to specify a termination condition)
- [dx_setuio\(\)](#)

dx_recf() — record voice data to a single file

dx_recf()

Name: int dx_recf(chdev, fnamep, tptp, mode)

Inputs:

int chdev	• valid channel device handle
char *fnamep	• pointer to name of 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: HMP Software, Springware boards

■ 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 name of the file where 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 481.
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 320 for record mode combinations).

■ Cautions

None.

■ **Errors**

If the function returns -1, use the Dialogic® 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_BADPARAM
Invalid parameter
- EDX_BADTPT
Invalid **DX_TPT** entry
- EDX_BUSY
Busy executing I/O function
- EDX_SYSTEM
Error from operating system

■ **Source Code**

```

/*****
 *      NAME: int dx_recf(devd,filep,tptp,mode)
 * DESCRIPTION: Record data to a file
 *      INPUTS: devd - channel descriptor
 *              tptp - TPT pointer
 *              filep - ASCIIIZ string for name of file to read into
 *              mode - tone initiation flag
 *      OUTPUTS: Data stored in file, status in CSB pointed to by csbp
 *      RETURNS: 0 or -1 on error
 *      CALLS: open() dx_rec() close()
 *      CAUTIONS: none.
 *****/

*/
int dx_recf(devd,filep,tptp,mode)
int devd;
char *filep;
DV_TPT *tptp;
USHORT mode;
{
    int rval;
    DX_IOTT iott;
    /*
     * If Async then return Error
     * Reason: IOTT's must be in scope for the duration of the record
     */
    if ( mode & EV_ASYNC ) {
        return( -1 );
    }

    /* Open the File */
    if ((iott.io_fhandle = open(filep,(O_WRONLY|O_CREAT|O_TRUNC),0666)) == -
        1) {
        return -1;
    }
}

```

dx_recf() — record voice data to a single file

```
/* Use dx_rec() to do the record */
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>

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_clrtp(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_TERMMASK(chdev); /* investigate termination reason */
    if (termtype & TM_MAXDTMF) {
        /* process DTMF termination */
    }
    . . .
}
```

■ See Also

- [dx_rec\(\)](#)

record voice data to a single file — dx_recf()

- [dx_reciottdata\(\)](#)
- [dx_recm\(\)](#)
- [dx_recmf\(\)](#)
- [dx_recvox\(\)](#)
- [dx_setparm\(\)](#)
- [dx_getparm\(\)](#)
- [ATDX_TERMMSK\(\)](#)
- [DV_TPT](#) data structure (to specify a termination condition)

dx_reciottdata()

Name: int dx_reciottdata(chdev, iottp, tptp, xpbp, mode)

Inputs:

int chdev	• valid channel device handle
DX_IOTT *iottp	• pointer to I/O Transfer Table structure
DV_TPT *tptp	• pointer to Termination Parameter Table structure
DX_XPB *xpbp	• pointer to I/O Transfer Parameter block
unsigned short mode	• play mode

Returns: 0 if success
-1 if failure

Includes: srllib.h
dxxxlib.h

Category: I/O

Mode: asynchronous or synchronous

Platform: HMP Software, Springware boards

■ 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.

Parameter	Description
chdev	specifies the valid channel device handle obtained when the channel was opened using dx_open()
iottp	points to the I/O Transfer Table Structure, DX_IOTT , which specifies the order 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 502, for more information on this data structure.
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 481.

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 DX_XPB , on page 514.
mode	<p>specifies the recording mode. One or more of the values listed below may be selected in the bit mask using bitwise OR.</p> <p>Choose one only:</p> <ul style="list-style-type: none"> • EV_ASYNC – asynchronous mode • EV_SYNC – synchronous mode <p>Choose one or more:</p> <ul style="list-style-type: none"> • MD_NOGAIN – record without automatic gain control (AGC). AGC is on by default. • RM_TONE – transmit a 200 msec tone before initiating record. • RM_VADNOTIFY – generates an event, TDX_VAD, on detection of voice energy by the voice activity detector (VAD) during the recording operation. For details on recording with the voice activity detector (VAD), see the <i>Voice API Programming Guide</i>. Note that TDX_VAD does not indicate function termination; it is an unsolicited event. Do not confuse this event with the TEC_VAD event which is used in the continuous speech processing (CSP) library. • RM_ISCR – adds initial silence compression to the voice activity detector (VAD) capability. Note that the RM_ISCR mode can only be used in conjunction with RM_VADNOTIFY. For details on recording with the voice activity detector (VAD), see the <i>Voice API Programming Guide</i>. • RM_NOTIFY – (Windows® only) generate record notification beep tone. • RM_USERTONE – (Springware boards Linux only) plays a user-defined tone before initiating record. <p>On Linux, once the GTG tone template has been set in firmware, the application may use the customized tone preceding a record by specifying both the RM_TONE and RM_USERTONE bits. If the RM_USERTONE bit is not set but the RM_TONE bit is set in the record mode field, the built-in tone will be played prior to initiating a record. For details, see dx_settone().</p>

■ **Cautions**

- On HMP Software, voice channels must be listening to a TDM bus time slot in order for voice recording functions, such as [dx_reciottdata\(\)](#), to work. The actual recording operation will start only after the voice channel is listening to the proper external time slot. In other words, you must issue a [dx_listen\(\)](#) function call on the device handle before calling a voice recording function for that device handle, and the [dx_listen\(\)](#) must be called from the same process as the voice recording function. If not, the voice recording function will return TDX_ERROR with EDX_SH_MISSING as the termination reason.
- 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.

***dx_reciottdata()* — record voice data to multiple destinations**

- 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.
- It is recommended that you start recording before receiving any incoming data on the channel so that initial data is not missed in the recording.

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

EDX_BADIOTT
Invalid **DX_IOTT** setting

EDX_BADWAVFILE
Invalid WAVE file

EDX_BUSY
Channel is busy

EDX_SYSTEM
Error from operating system

EDX_XPBPARAM
Invalid **DX_XPB** setting

EDX_SH_BADCMD
Unsupported command or WAVE file format

■ **Example**

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

main()
{
```

record voice data to multiple destinations — dx_reciottdata()

```
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("dxxxBlC1",0)) == -1) {
    printf("Cannot open channel\n");
    /* Perform system error processing */
    exit(1);
}

/* Set to terminate play on 1 digit */
tpt.tp_type = IO_EOT;
tpt.tp_termno = DX_MAXDTMF;
tpt.tp_length = 1;
tpt.tp_flags = TF_MAXDTMF;

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

/* For Linux applications: open file */
if ((fd = open("MESSAGE.VOX",O_RDWR)) == -1) {
    printf("File open error\n");
    exit(2);
}

/* Set up DX_IOTT */
iott.io_fhandle = fd;
iott.io_bufp = 0;
iott.io_offset = 0;
iott.io_length = -1;
iott.io_type = IO_DEV | IO_EOT;
/*
 * Specify VOX file format for PCM at 8KHz.
 */
xpb.wFileFormat = FILE_FORMAT_VOX;
xpb.wDataFormat = DATA_FORMAT_PCM;
xpb.nSamplesPerSec = DRT_8KHZ;
xpb.wBitsPerSample = 8;
/* Wait forever for phone to ring and go offhook */
if (dx_wtring(chdev,1,DX_OFFHOOK,-1) == -1) {
    printf("Error waiting for ring - %s\n", ATDV_LASTERR(chdev));
    exit(3);
}
/* Play intro message */
if (dx_playvox(chdev,"HELLO.VOX",&tpt,&xpb,EV_SYNC) == -1) {
    printf("Error playing file - %s\n", ATDV_ERRMSGP(chdev));
    exit(4);
}
/* Start recording */
if (dx_reciottdata(chdev,&iott,&tpt,&xpb,PM_TONE|EV_SYNC) == -1) {
    printf("Error recording file - %s\n", ATDV_ERRMSGP(chdev));
    exit(4);
}
}
```

■ See Also

- [dx_rec\(\)](#)

dx_reciottdata() — record voice data to multiple destinations

- [dx_recf\(\)](#)
- [dx_recm\(\)](#)
- [dx_recmf\(\)](#)
- [dx_recvox\(\)](#)
- [dx_recwav\(\)](#)
- [dx_setuio\(\)](#)

`dx_recm()`

Name: `int dx_recm(chdev, iottp, tptp, mode, tsinfo)`

Inputs:

<code>int chdev</code>	• valid channel device handle
<code>DX_IOTT *iottp</code>	• pointer to I/O Transfer Table structure
<code>DV_TPT *tptp</code>	• pointer to Termination Parameter Table structure
<code>unsigned short mode</code>	• recording mode bit mask for this record session
<code>SC_TSINFO *tsinfo</code>	• pointer to TDM bus Time Slot Information structure

Returns: 0 if successful
-1 if failure

Includes: `srllib.h`
`dxxxlib.h`

Category: I/O

Mode: asynchronous or asynchronous

Platform: Springware boards Linux

■ Description

The `dx_recm()` function records voice data from two channels to a combination of data files, memory, or custom devices.

This function is used for the Transaction record feature, which enables the recording of a two-party conversation by allowing two TDM bus time slots from a single channel to be recorded.

Note: On HMP Software, use the `dx_mreciottdata()` function for transaction record.

Parameter	Description
<code>chdev</code>	specifies the valid channel device handle obtained when the channel was opened using <code>dx_open()</code>
<code>iottp</code>	points to the I/O Transfer Table Structure, <code>DX_IOTT</code> , 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 <code>DX_IOTT</code> , on page 502, for more information on this data structure.
<code>tptp</code>	points to the Termination Parameter Table Structure, <code>DV_TPT</code> , which specifies termination conditions for this function. For a list of termination conditions and for more information on this structure, see <code>DV_TPT</code> , on page 481.

dx_recn() — record voice data from two channels

Parameter	Description
mode	defines the recording mode for the record session. One or more of the values listed in the description of the mode parameter for dx_rec() may be selected in the bitmask using bitwise OR (see Table 11, “Record Mode Selections” , on page 320 for record mode combinations).
tsinfo	points to the SC_TSINFO structure that contains the TDM bus time slot information. To have two-channel recording, you need to provide information on both time slots in the SC_TSINFO. For more information on this structure, see SC_TSINFO , on page 523.

After **dx_recn()** 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. In addition, recording will stop if the function fails; for example, if maximum byte count is exceeded or the end of the file is reached.

When **dx_recn()** 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.

By default, this function runs synchronously, and returns 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 returns 0 to indicate it has initiated successfully, and generates a **TDX_RECORD** termination event to indicate completion.

Termination of asynchronous recording is indicated by the same **TDX_RECORD** event used in **dx_rec()**. Use the Standard Runtime Library (SRL) Event Management functions to handle the termination event.

■ Cautions

- When playing pre-recorded data, ensure it is played using the same encoding algorithm and sampling rate used when the data was recorded.
- When using MSI/SC products for transaction recording, ensure that a full duplex connection is established. You must call **ms_listen()** even though the MSI station is used for transmitting.
- Since the digital signal processor (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 sampling rate changes will apply to both time slots. Thus, 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 function 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. See [dx_listen\(\)](#) for an explanation of the errors.

- The API will connect the channel to the time slot specified in the `sc_tsarrayp[0]` field of the `SC_TSINFO` structure. The record channel will continue to listen to both time slots after the function has completed, until `dx_listen()` or `dx_unlisten()` is subsequently issued to re-route the record channel. Both `sc_tsarrayp[0]` and `sc_tsarrayp[1]` must be within the range 0 to 1023. No checking is made to verify that `sc_tsarrayp[0]` or `sc_tsarrayp[1]` has been connected to a valid channel. For more information on this structure, see `SC_TSINFO`, on page 523.
- Upon termination of the `dx_recm()` or `dx_recmf()` function, the recording channel continues to listen to the time slot pointed to by `sc_tsarray[0]`.
- The recording channel can only detect a loop current drop on a physical analog interface that is associated with that channel. If you have a configuration where the recording channel is not listening to its corresponding analog interface, you will have to design the application to detect the loop current drop event and issue a `dx_stopch()` to the recording device. The recording channel hook state should be off-hook while the recording is in progress.
- Any device connected via the TDM bus to the recording device of the transaction record function `dx_recm()` will be unrouted before the transaction record starts and will not be routed back to the device when the transaction is completed.
- 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. 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. One of the following error codes may be returned:

`EDX_BADIOTT`
Invalid `DX_IOTT` entry

`EDX_BADPARAM`
Invalid parameter

`EDX_BADTPT`
Invalid `DX_TPT` entry

`EDX_BUSY`
Busy executing I/O function

`EDX_SYSTEM`
Error from operating system

■ Example 1

This example illustrates using `dx_recm()` in synchronous mode.

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

dx_recm() — record voice data from two channels

```
#define MAXLEN 10000

main()
{
    DV_TPT tpt;
    DX_IOTT iott[2];
    int chdev1;
    char basebufp[MAXLEN];
    SC_TSINFO tsinfo;
    long scts1, scts2, arrayp[32];

    /* Open the channel */
    if ((chdev1 = dx_open("dxxxB1C1", NULL)) == -1){
        printf("Could not open dxxxB1C1\n");
        exit (1);
    }

    /* get two external timeslots */
    arrayp[0] = scts1;
    arrayp[1] = scts2;

    tsinfo.sc_numts = 2;
    tsinfo.sc_tsarrayp = &arrayp[0];

    /* Setup DV_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;

    /* Setup DX_IOTT */
    iott[0].io_type = IO_MEM | IO_CONT ;
    iott[0].io_bufp = basebufp ;
    iott[0].io_offset= 0;
    iott[0].io_length= MAXLEN;

    iott[1].io_type = IO_DEV | IO_EOT ;
    iott[1].io_bufp = 0;
    iott[1].io_offset = 0;
    iott[1].io_length = MAXLEN ;

    if ((iott[1].io_fhandle = open("file.vox", O_RDWR | O_CREAT | O_TRUNC,
        0666)) == -1){
        printf("File open error\n");
        exit (1);
    }
    if (dx_clrdigbuf(chdev1) == -1) {
        printf("Error Message = %s\n", ATDV_ERRMSGP(chdev1));
        exit (1);
    }

    if (dx_recm(chdev1, &iott[0], &tpt, RM_TONE | EV_SYNC, &tsinfo) == -1) {
        printf("Error Message = %s\n", ATDV_ERRMSGP(chdev1));
        exit (1);
    }

    if (dx_close(chdev1) == -1) {
        printf("Error Message = %s\n", ATDV_ERRMSGP(chdev1));
        exit (1);
    }
}
```

■ Example 2

This example illustrates using `dx_recvm()` in asynchronous mode.

```
#include <stdio.h>
#include <fcntl.h>
#include <srllib.h>
#include <dxxplib.h>

#define MAXLEN 10000

main()
{
    DV_TPT tpt;
    DX_IOTT iott[2];
    int chdev1;
    char basebufp[MAXLEN];
    SC_TSINFO tsinfo;
    long scts1, scts2, arrayp[32];
    int srlmode;

    /* Open the channel */
    if ((chdev1 = dx_open("dxxxB1C1", NULL)) == -1){
        printf("Could not open dxxxB1C1\n");
        exit (1);
    }

    srlmode = SR_POLLMODE;

    if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1){
        printf("Cannot set SRL to Polled mode ! \n");
        exit(1);
    }

    if (sr_enbhdr(chdev1, TDX_RECORD, record_handler) == -1){
        printf("Enable handler failed from CH-1\n");
        exit (1);
    }

    /* get two external timeslots */
    arrayp[0] = scts1;
    arrayp[1] = scts2;
    tsinfo.sc_numts = 2;
    tsinfo.sc_tsarrayp = &arrayp[0];

    /* Set up DV_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 */
    iott[1].io_type = IO_DEV | IO_EOT ;
    iott[1].io_bufp = basebufp;
    iott[1].io_offset = 0;
    iott[1].io_length = -1 ;

    if ((iott[1].io_fhandle = open("file.vox", O_RDWR | O_CREAT | O_TRUNC,
        0666)) == -1){
        printf("File open error\n");
        exit (1);
    }
}
```

dx_recv() — record voice data from two channels

```
if (dx_clrdigbuf(chdev1) == -1) {
    printf("Error Message = %s\n", ATDV_ERRMSGP(chdev1));
    exit (1);
}

if (dx_recv(chdev1, &iott[1], &tpt, RM_TONE | EV_ASYNC, &tsinfo) == -1) {
    printf("Error Message = %s\n", ATDV_ERRMSGP(chdev1));
    exit (1);
}

    printf ("Waiting for Event ..... \n");

if(sr_waitevt(-1) == -1){
    printf("sr_waitevt, %s\n", ATDV_ERRMSGP(SRL_DEVICE));
    exit(1);
}
/*Disable the handler*/
if (sr_dishdlr(chdev1, TDX_RECORD, record_handler) == -1){
    printf("Disable handler fialed from CH-1\n");
    exit (1);
}

if (dx_close(chdev1) == -1) {
    printf("Error Message = %s\n", ATDV_ERRMSGP(chdev1));
    exit (1);
}
}

int record_handler(){

    long term;
    term = ATDX_TERMSK(sr_getevtdev());

    if (term & TM_MAXDTMF) {
        printf("Record terminated on receiving DTMF digit\n");
    }else if (term & TM_NORMTERM) {
        printf ("Normal termination of dx_rec\n");
    }else {
        printf("Unknown termination reason : %x\n", term);
    }
}
}
```

■ See Also

- [dx_recf\(\)](#)
- [dx_reciottdata\(\)](#)
- [dx_recmf\(\)](#)
- [dx_recvox\(\)](#)
- [dx_setparm\(\)](#)
- [dx_getparm\(\)](#)

`dx_recmf()`

Name: `int dx_recmf(chdev, fnamep, tptp, mode, tsinfop)`

Inputs:

<code>int chdev</code>	• valid channel device handle
<code>char *fnamep</code>	• pointer to file where voice data will be recorded
<code>DV_TPT *tptp</code>	• pointer to Termination Parameter Table structure
<code>unsigned short mode</code>	• recording mode bit mask for this record session
<code>SC_TSINFO *tsinfop</code>	• pointer to TDM bus Time Slot Information structure

Returns: 0 if successful
-1 if failure

Includes: `srllib.h`
`dxxxlib.h`

Category: I/O

Mode: asynchronous or synchronous

Platform: Springware boards Linux

■ Description

The `dx_recmf()` function records voice data from two channels to a single file.

This function is used for the transaction record feature, which enables the recording of a two-party conversation by allowing two TDM bus time slots from a single channel to be recorded.

Note: On HMP Software, use the `dx_mreciottdata()` function for transaction record.

Calling `dx_recmf()` is the same as calling `dx_recmf()` and specifying a single file entry in the `DX_IOTT` structure. Using `dx_recmf()` is more convenient for recording to single 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_recmf()` function opens and closes the file specified by `fnamep`.

Parameter	Description
<code>chdev</code>	specifies the valid channel device handle obtained when the channel was opened using <code>dx_open()</code>
<code>fnamep</code>	points to the file to which voice data will be recorded
<code>tptp</code>	points to the Termination Parameter Table Structure, <code>DV_TPT</code> , which specifies termination conditions for recording. For a list of termination conditions and for more information on this structure, see <code>DV_TPT</code> , on page 481.

dx_recmf() — *record voice data from a single channel*

Parameter	Description
mode	defines the recording mode for the record session. One or more of the values listed in the description of the mode parameter for dx_rec() may be selected in the bitmask using bitwise OR (see Table 11, “Record Mode Selections”, on page 320 for record mode combinations).
tsinfo	points to the SC_TSINFO data structure that contains the TDM bus time slot information. To have two-channel recording, you need to provide information on both time slots in the SC_TSINFO. For more information on this structure, see SC_TSINFO , on page 523.

After **dx_recmf()** 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. In addition, recording will stop if the function fails; for example, if maximum byte count is exceeded or the end of the file is reached.

When **dx_recmf()** 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.

By default, this function runs synchronously, and returns 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 returns 0 to indicate it has initiated successfully, and generates a **TDX_RECORD** termination event to indicate completion.

Termination of asynchronous recording is indicated by the same **TDX_RECORD** event used in **dx_rec()**. Use the Standard Runtime Library (SRL) Event Management functions to handle the termination event.

■ **Cautions**

See the Cautions section in the **dx_recmf()** function description for information.

■ **Errors**

In asynchronous mode, the function returns immediately and a **TDX_RECORD** event is queued upon completion. 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. One of the following error codes may be returned:

- EDX_BADIOTT**
Invalid **DX_IOTT** entry
- EDX_BADPARAM**
Invalid parameter

EDX_BADTPT
Invalid DX_TPT entry

EDX_BUSY
Busy executing I/O function

EDX_SYSTEM
Error from operating system

■ Example

```
#include <fcntl.h>
#include <srllib.h>
#include <dxxxlib.h>
#define MAXLEN 10000

main()
{
    int chdev;
    DV_TPT tpt[2];
    long termtype;
    SC_TSINFO tsinfo;
    long scts1,scts2, ts_array[32];

    /* Open the channel */
    if ((chdev = dx_open("dxxxB1C1", NULL)) == -1) {
        printf("Could not open dxxxB1C1\n");
        exit (1);
    }

    /* get two external timeslots */
    arrayp[0] = scts1;
    arrayp[1] = scts2;
    tsinfo.sc_numts = 2;
    tsinfo.sc_tsarrayp = &arrayp[0];

    /* Set up DV_TPT structure */
    dx_clrtp(tpt,2);
    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_MAXSIL;
    tpt[1].tp_length = 20;
    tpt[1].tp_flags = TF_MAXSIL | TF_SETINIT;
    tpt[1].tp_data = 40;

    if (dx_recmf(chdev,"file.vox", tpt, RM_TONE, &tsinfo) == -1) {
        printf("Error Message = %s\n", ATDV_ERRMSGP(chdev));
        exit (1);
    }

    termtype = ATDX_TERMSK(chdev);

    if (dx_unlisten(chdev) == -1) {
        printf("Error Message = %s\n", ATDV_ERRMSGP(chdev));
        exit (1);
    }
}
```

dx_recmf() — record voice data from a single channel

```
if (dx_close(chdev) == -1) {  
    printf("Error Message = %s\n", ATDV_ERRMSGP(chdev));  
    exit (1);  
}  
}
```

■ See Also

- [dx_recf\(\)](#)
- [dx_reciottdata\(\)](#)
- [dx_recm\(\)](#)
- [dx_recvox\(\)](#)
- [dx_setparm\(\)](#)
- [dx_getparm\(\)](#)

`dx_recvox()`

Name: `int dx_recvox(chdev, filenamep, tptp, xpbp, mode)`

Inputs:

<code>int chdev</code>	• valid channel device handle
<code>char *filenamep</code>	• pointer to name of file to record to
<code>DV_TPT *tptp</code>	• pointer to Termination Parameter Table structure
<code>DX_XPB *xpbp</code>	• pointer to I/O Transfer Parameter Block structure
<code>unsigned short mode</code>	• record mode

Returns: 0 if successful
-1 if failure

Includes: `srllib.h`
`dxxlib.h`

Category: I/O Convenience

Mode: synchronous

Platform: HMP Software, Springware boards

■ Description

The `dx_recvox()` function records voice data from a channel to a single VOX file. This is a convenience function.

Parameter	Description
chdev	specifies the valid channel device handle obtained when the channel was opened using <code>dx_open()</code>
filenamep	points to the name of the VOX file to record to
tptp	points to the Termination Parameter Table Structure, <code>DV_TPT</code> , which specifies termination conditions for recording. For more information on this structure, see <code>DV_TPT</code> , on page 481.
xpbp	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 <code>DX_XPB</code> , on page 514. <i>Note:</i> If xpbp is set to <code>NULL</code> , this function interprets the data as 6 kHz linear ADPCM.
mode	specifies the record mode. The following values may be used individually or ORed together: <ul style="list-style-type: none"> • <code>EV_SYNC</code> – synchronous operation (must be specified) • <code>RM_TONE</code> – transmits a 200 msec tone before initiating record

***dx_recvox()* — record voice data to a single VOX file**

■ Cautions

- On HMP Software, voice channels must be listening to a TDM bus time slot in order for voice recording functions, such as **dx_recvoicdata()**, to work. The actual recording operation will start only after the voice channel is listening to the proper external time slot. In other words, you must issue a **dx_listen()** function call on the device handle before calling a voice recording function for that device handle, and the **dx_listen()** must be called from the same process as the voice recording function. If not, the voice recording function will return **TDX_ERROR** with **EDX_SH_MISSING** as the termination reason.
- When playing or recording VOX files, the data format is specified in **DX_XPB** rather than through the mode parameter of **dx_recvox()**.
- It is recommended that you start recording before receiving any incoming data on the channel so that initial data is not missed in the recording.

■ Errors

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

EDX_BADIOTT
Invalid **DX_IOTT** setting

EDX_BUSY
Channel is busy

EDX_SH_BADCMD
Unsupported command or VOX file format

EDX_SYSTEM
Error from operating system

EDX_XPBPARAM
Invalid **DX_XPB** setting

■ Example

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

main()
{
    int chdev;          /* channel descriptor */
    DV_TPT tpt;        /* termination parameter table */
    DX_XPB xpb;        /* I/O transfer parameter block */
    .
    .
    .

    /* Open channel */
    if ((chdev = dx_open("dxxxB1C1",0)) == -1) {
        printf("Cannot open channel\n");
        /* Perform system error processing */
        exit(1);
    }
}
```

record voice data to a single VOX file — dx_recvox()

```
/* 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 prompt playback */
if (dx_playvox(chdev,"HELLO.VOX",&tpt,EV_SYNC) == -1) {
    printf("Error playing file - %s\n", ATDV_ERRMSGP(chdev));
    exit(4);
}

/* clear digit buffer */
dx_clrdigbuf(chdev);

/* Start 6KHz ADPCM recording */
if (dx_recvox(chdev,"MESSAGE.VOX",&tpt,NULL,RM_TONE|EV_SYNC) == -1){
    printf("Error recording file - %s\n", ATDV_ERRMSGP(chdev));
    exit(4);
}
}
```

■ See Also

- [dx_rec\(\)](#)
- [dx_recf\(\)](#)
- [dx_reciottdata\(\)](#)
- [dx_recm\(\)](#)
- [dx_recmf\(\)](#)
- [dx_recwav\(\)](#)

dx_recwav() — record voice data to a single WAVE file

dx_recwav()

Name: int dx_recwav(chdev, filenamep, tptp, xpbp, mode)

Inputs:

int chdev	• valid channel device handle
char *filenamep	• pointer to name of file to record to
DV_TPT *tptp	• pointer to Termination Parameter Table structure
DX_XPB *xpbp	• pointer to I/O Transfer Parameter Block
unsigned short mode	• record mode

Returns: 0 if successful
-1 if failure

Includes: srllib.h
dxxxlib.h

Category: I/O Convenience

Mode: synchronous

Platform: HMP Software, Springware boards

■ Description

The **dx_recwav()** convenience function records voice data to a single WAVE file. This function in turn calls **dx_reciottdata()**.

Parameter	Description
chdev	specifies the valid channel device handle obtained when the channel was opened using dx_open()
tptp	points to the Termination Parameter Table structure, DV_TPT, which specifies termination conditions for playing. For more information on this structure, see DV_TPT , on page 481.
filenamep	points to the name of the file to record to
xpbp	points to the I/O Transfer Parameter Block, DX_XPB, which specifies the file format, data format, sampling rate, and resolution. For more information on this structure, see DX_XPB , on page 514. <i>Note:</i> If xpbp is set to NULL, the function will record in 11 kHz linear 8-bit PCM.
mode	specifies the record mode. The following values may be used individually or ORed together: <ul style="list-style-type: none">• EV_SYNC – synchronous operation (must be specified)• RM_TONE – transmits a 200 msec tone before initiating record

■ Cautions

- On HMP Software, voice channels must be listening to a TDM bus time slot in order for voice recording functions, such as `dx_reciottdata()`, to work. The actual recording operation will start only after the voice channel is listening to the proper external time slot. In other words, you must issue a `dx_listen()` function call on the device handle before calling a voice recording function for that device handle, and the `dx_listen()` must be called from the same process as the voice recording function. If not, the voice recording function will return `TDX_ERROR` with `EDX_SH_MISSING` as the termination reason.
- It is recommended that you start recording before receiving any incoming data on the channel so that initial data is not missed in the recording.

■ Errors

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

`EDX_BADIOTT`
Invalid `DX_IOTT` setting

`EDX_BADWAVFILE`
Invalid WAVE file

`EDX_BUSY`
Channel is busy

`EDX_SH_BADCMD`
Unsupported command or WAVE file format

`EDX_SYSTEM`
Error from operating system

`EDX_XPBPARM`
Invalid `DX_XPB` setting

■ Example

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

main()
{
    int chdev;          /* channel device handle */
    DV_TPT tpt;        /* termination parameter table */
    DX_XPB xpb;        /* I/O transfer parameter block */
    .
    .
    .

    /* Open channel */
    if ((chdev = dx_open("dxxxB1C1",0)) == -1) {
        printf("Cannot open channel\n");
        /* Perform system error processing */
        exit(1);
    }
}
```

dx_recwav() — record voice data to a single WAVE file

```
/* 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 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_resetch()`

Name: `dx_resetch(chdev, mode)`

Inputs: `int chdev` • valid channel device handle
`int mode` • mode of operation

Returns: 0 if success
-1 if failure

Includes: `srllib.h`
`dxxplib.h`

Category: I/O

Mode: asynchronous or synchronous

Platform: HMP Software

■ Description

The `dx_resetch()` function recovers a channel that is “stuck” (busy or hung) and in a recoverable state, and brings it to an idle and usable state. This function blocks all other functions from operating on the channel until the function completes. I

Parameter	Description
<code>chdev</code>	Specifies the valid device handle obtained when the channel was opened using <code>dx_open()</code>
<code>mode</code>	Specifies the mode of operation: <ul style="list-style-type: none">• <code>EV_ASYNC</code> – asynchronous mode. The calling thread returns immediately so it can process media functionality on other channels.• <code>EV_SYNC</code> – synchronous mode. The calling thread waits until the channel is recovered or discovers that the channel is not in a recoverable state.

In synchronous mode, 0 is returned if the function completes successfully, and -1 is returned in case of error.

In asynchronous mode, the `TDX_RESET` event is generated to indicate that the channel was recovered and is in an idle and usable state. The `TDX_RESETERR` event is generated to indicate that the channel is not recoverable. Issuing any other media calls on this channel will result in an error.

■ Cautions

- The `dx_resetch()` function is intended for use on channels that are stuck and not responding. Do **not** use it in place of `dx_stopch()`. Use `dx_resetch()` only if you do not receive an event within 30 seconds of when it’s expected. Overuse of this function creates unnecessary overhead and may affect system performance.

dx_resetch() — *reset a channel that is hung*

■ Errors

If the function returns -1, use the Dialogic® 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_FWERROR
Firmware error

EDX_NOERROR
No error

■ Example

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

main()
{
    int chdev, srlmode;
    /* Set SRL to run in polled mode. */
    srlmode = SR_POLLMODE;

    if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1) {
        /* process error */
    }

    /* Open the channel using dx_open(). Get channel device descriptor in
    * chdev.
    */

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

    /* continue processing */
    . . .
    /* Force the channel to idle state. The I/O function that the channel
    * is executing will be terminated, and control passed to the handler
    * function previously enabled, using sr_enbhdr(), for the
    * termination event corresponding to that I/O function.
    * In asynchronous mode, dx_stopch() returns immediately,
    * without waiting for the channel to go idle.
    */

    if ( dx_stopch(chdev, EV_ASYNC) == -1) {
        /* process error */
    }

    /* Wait for dx_stopch() to stop the channel and return the termination event
    * for the present media function.
    */

    /* After waiting for 30 secs if the termination event is not returned, issue a
    * dx_resetch() to reset the channel.
    */

    if (dx_resetch(chdev, EV_ASYNC) <0 )
    {
        /*process error */
    }
}
```


reset a channel that is hung — dx_resetch()

```
    }  
  
    /* Wait for TDX_RESET or TDX_RESETEERR events */  
  
}
```

■ See Also

- **ec_resetch()** in the *Dialogic® Continuous Speech Processing API Library Reference*

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: HMP Software

■ 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).

Parameter	Description
hBuffer	specifies the circular stream buffer handle

■ 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 Dialogic[®] 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

```
#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");
    }
}
```

reset internal data for a circular stream buffer — dx_ResetStreamBuffer()

```
        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_RxlottData() — receive data on a specified channel

dx_RxlottData()

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: Springware boards

■ Description

The **dx_RxIottData()** function is used to receive data on a specified channel. The **wType** parameter specifies the type of data to be received, for example ADSI data.

Parameter	Description
chdev	specifies the valid channel device handle obtained when the channel was opened using dx_open()
iottp	points to the I/O Transfer Table, DX_IOTT . The iottp parameter specifies the destination for the received data. This is the same DX_IOTT structure used in dx_playiottdata() and dx_reciottdata() . See DX_IOTT , on page 502, for more information on this data structure.
lpTerminations	points to the Termination Parameter Table Structure, DV_TPT , which specifies termination conditions for the device handle. Supported values are: <ul style="list-style-type: none">• DX_MAXTIME For more information on this structure, see DV_TPT , on page 481.
wType	specifies the type of data to be received. To receive ADSI data, set wType to DT_ADSI .

Parameter	Description
lpParams	points to information specific to the data type specified in wType . The format of the parameter block depends on wType . For ADSI data, set lpParams to point to an <code>ADSI_XFERSTRUC</code> structure. For more information on this structure, see ADSI_XFERSTRUC , on page 474.
mode	specifies how the function should execute: <ul style="list-style-type: none"> • <code>EV_ASYNC</code> – asynchronous • <code>EV_SYNC</code> – synchronous

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:

<code>TM_EOD</code>	End of FSK data detected on receive
<code>TM_ERROR</code>	I/O device error
<code>TM_MAXTIME</code>	Maximum function time exceeded
<code>TM_USRSTOP</code>	Function stopped by user

When running asynchronously, this function returns 0 to indicate it has initiated successfully, and generates a `TDX_RXDATA` termination event to indicate completion.

■ 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.
- `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

In asynchronous mode, the function returns immediately and a `TDX_RXDATA` event is queued upon completion. 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:

<code>EDX_BADIOTT</code>	Invalid <code>DX_IOTT</code> (pointer to I/O transfer table)
--------------------------	--

dx_RxIottData() — receive data on a specified channel

EDX_BADPARAM
Invalid data mode

EDX_BUSY
Channel already executing I/O function

EDX_SYSTEM
Error from operating system

■ 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 = "dxxxB1C1";
    char buffer[16];
    ADSI_XFERSTRUC adsimode;
    DV_TPT tpt;
    int chdev;
    .
    .
    .

    sprintf(buffer, "RECEIVE.ADSI");

    if ((iott.io_fhandle = dx_fileopen(buffer, O_RDWR|O_CREAT|O_TRUNC|O_BINARY, 0666)) == -1) {
        // process error
        exit(1);
    }

    if ((chdev = dx_open(devnamep, 0)) == -1) {
        fprintf(stderr, "Error opening channel %s\n", devnamep);
        dx_fileclose(iott.io_fhandle);
        exit(2);
    }
    .
    .
    .

    // 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 DV_TPT.
    // Application specific value is used to terminate dx_RxIottData()
    // if end of data is not detected over a specified duration.
```

receive data on a specified channel — `dx_RxlottData()`

```
tpt.tp_type = IO_EOT;
if (dx_clrtpt(&tpt, 1) == -1) {
    // Process error
}

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));
}
.
.
}
```

■ See Also

- [dx_TxIottData\(\)](#)
- [dx_TxRxIottData\(\)](#)

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
dxxxlib.h

Category: Call Status Transition Event

Mode: synchronous

Platform: Springware boards

■ 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.

Parameter	Description
dev	specifies the valid channel device handle obtained when the channel was opened using dx_open()
evttype	specifies the type of event to be sent. See the following page for more information on defining the type of event.
evtdatap	points to a data block associated with evttype . <i>Note:</i> The evtdatap parameter can be NULL and the evtlen parameter 0 if there is no data associated with an event type.
evtlen	specifies the length of the data block in bytes (between 0 and 256)
flags	determines which processes are going to receive this event. Valid values are: <ul style="list-style-type: none">• EVFL_SENDESELF – Only the process calling dx_sendevt() will receive the event.• EVFL_SENDOOTHERS – 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_enbhdr()**, or by calling **dx_getevt()** to catch the event if the **evttype** is set to TDX_CST.

The application can define the **evtttype** and **evtdata** to be any values as long as **evtttype** is greater than 0x1FFFFFFF 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 **evtdata** 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_BADPARAM
Invalid parameter

EDX_SYSTEM
Error from operating system

■ Example

```
#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 dxxxB1C1 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_enbhdr\(\)](#)
- [sr_waitevt\(\)](#)

dx_sendevt() — *allow inter-process event communication*

`dx_setchxfercnt()`

Name: `int dx_setchxfercnt(chdev, bufsize_identifier)`

Inputs: `int chdev` • valid channel device handle
`int bufsize_identifier` • equate for a buffer size

Returns: 0 to indicate successful completion
 -1 if failure

Includes: `srllib.h`
`dxxlib.h`

Category: Configuration

Mode: synchronous

Platform: HMP Software, Springware boards

■ 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 driver. The minimum buffer size is 1 Kbytes, and the largest is 32 Kbytes.

This function is typically used in conjunction with the user I/O feature or the streaming to board feature. (For more information on user I/O, see the `dx_setuio()` function.) This function sets up the frequency with which the application-registered UIO 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. For information on streaming to board functions, see [Section 1.5, “Streaming to Board Functions”](#), on page 18. For streaming to board programming guidelines, see the *Dialogic® Voice API Programming Guide*.

Parameter	Description
<code>chdev</code>	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
<code>bufsize_identifier</code>	specifies the bulk queue buffer size for the channel. Use one of the following values: <ul style="list-style-type: none"> • 0 – sets the buffer size to 4 Kbytes • 1 – sets the buffer size to 8 Kbytes • 2 – sets the buffer size to 16 Kbytes (default) • 3 – sets the buffer size to 32 Kbytes • 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.

***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 the board. 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.

■ 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_SYSTEM
Error from operating system

■ Example

```
#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\(\)](#)
- [dx_OpenStreamBuffer\(\)](#)
- streaming to board topic in the *Dialogic® Voice API Programming Guide*

`dx_setdevuio()`

Name: `int dx_setdevuio(chdev, devuio, retuio)`

Inputs:

<code>int chdev</code>	• valid channel device handle
<code>DX_UIO *devuio</code>	• pointer to user I/O routines structure
<code>DX_UIO **retuio</code>	• pointer to return pointer for user I/O routines structure

Returns: 0 if successful
-1 error return code

Includes: `srllib.h`
`dxxlib.h`

Category: I/O

Mode: synchronous

Platform: HMP Software, Springware boards

■ Description

The `dx_setdevuio()` function installs and retrieves user-defined I/O functions on a per channel device basis. These user I/O functions are used on all subsequent I/O operations performed on the channel even if the application installs global user I/O functions for all devices using the `dx_setuio()` function. The user I/O functions are installed by installing a pointer to a `DX_UIO` structure which contains addresses of the user-defined I/O functions.

For more information on working with user-defined I/O functions, see the Application Development Guidelines chapter in the *Dialogic® Voice API Programming Guide*.

Parameter	Description
<code>chdev</code>	the channel for which the user-defined I/O functions will be installed
<code>devuio</code>	a pointer to an application-defined global <code>DX_UIO</code> structure which contains the addresses of the user-defined I/O functions. This pointer to the <code>DX_UIO</code> structure will be stored in the voice DLL for the specified <code>chdev</code> channel device. The application must not overwrite the <code>DX_UIO</code> structure until <code>dx_setdevuio()</code> has been called again for this device with the pointer to another <code>DX_UIO</code> structure.

dx_setdevuio() — install and retrieve user-defined I/O functions

Parameter	Description
retuiop	<p>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.</p> <p>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.</p>

■ **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 513 for more information.
- On HMP Software, 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 Dialogic® SRL Standard Attribute function **ATDV_LASTERR()** to obtain the error code or you can use **ATDV_ERRMSGP()** to obtain a descriptive error message. The error codes returned by **ATDV_LASTERR()** are:

EDX_BADDEV
Invalid device descriptor

EDX_BADPARAM
Invalid parameter

■ **Example**

```
#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 */
```

install and retrieve user-defined I/O functions — `dx_setdevuio()`

```
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("dxxxBlC1",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()

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 boards

■ Description

The **dx_setdigbuf()** function sets the digit buffering mode that will be used by the voice driver. Once the digit buffer is full, the application may select whether subsequent digits will be ignored or will overwrite the oldest digits in the queue. The maximum size of the digit buffer varies with the board type and technology.

Parameter	Description
chdev	specifies the valid channel device handle obtained when the channel was opened using dx_open()
mode	specifies the type of digit buffering that will be used. Mode can be: <ul style="list-style-type: none">• DX_DIGCYCLIC – Incoming digits will overwrite the oldest digits in the buffer if the buffer is full.• DX_DIGTRUNC – Incoming digits will be ignored if the digit buffer is full (default).

■ 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_BADPARAM
Invalid parameter

EDX_SYSTEM
Error from operating system

EDX_TIMEOUT
Timeout limit is reached

■ **Example**

```
#include <srllib.h>
#include <dxxlib.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_ERRMGSP(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: HMP Software, Springware boards

■ 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 HMP Software, the following are valid values: <ul style="list-style-type: none">• DM_DTMF – enable DTMF digit detection• DM_MF – enable MF digit detection• NULL – disable digit detection On Springware boards, the following are valid values: [<ul style="list-style-type: none">• D_DTMF – enable DTMF digit detection (default setting)• D_LPD – enable loop pulse detection• D_APD – enable audio pulse digits detection• D_MF – enable MF digit detection• D_DPD – enable dial pulse digit (DPD) detection• D_DPDZ – enable zero train DPD detection To disable digit detection, set dmask to NULL.

control the types of digits detected by the voice channel — `dx_setdigtyp()`

- 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 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 478.

■ 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 Dialogic® 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_SYSTEM`
Error from operating system

■ Example

Dial pulse detection (DPD) is supported on Springware boards only.

```
/*$ 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;
```

dx_setdigtyp() — control the types of digits detected by the voice channel

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

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

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

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

/* collect 3 digits from the user */
if (dx_getdig(dev, &tpt, &dig, EV_SYNC) == -1) {
    /* error, display error message */
    printf("dx_getdig error %d, %s\n", ATDV_LASTERR(dev), ATDV_ERRMSGP(dev));
} else {

    /* display digits received and digit type */
    printf("Received \"%s\"\n", dig.dg_value);
    printf("Digit type is ");

    /*
     * digit types have 0x30 ORed with them strip it off
     * so that we can use the DG_xxx equates from the header files
     */
    switch ((dig.dg_type[0] & 0x000f)) {
        case DG_DTMF:
            printf("DTMF\n");
            break;
        case DG_DPD:
            printf("DPD\n");
            break;
        default:
            printf("Unknown, %d\n", (dig.dg_type[0] & 0x000f));
    }
}

/*
 * continue processing call
 */
```

■ 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
 dxxplib.h

Category: Call Status Transition Event

Mode: synchronous

Platform: HMP Software, Springware boards

■ 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 470.

- 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**

Parameter	Description
chdev	specifies the valid channel device handle obtained when the channel was opened using dx_open()
mask	<p>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.</p> <p>On HMP Software, one or more of the following bits can be set:</p> <ul style="list-style-type: none">• DM_SILOF – wait for non-silence• DM_SILON – wait for silence• DM_DIGITS – enable digit reporting on the event queue (each detected digit is reported as a separate event on the event queue)• DM_DIGOFF – disable digit reporting on the event queue (as enabled by DM_DIGITS). This is the only way to disable DM_DIGITS.• DM_UNDERRUN – enables firmware underrun reporting (TDX_UNDERRUN event) for streaming to board feature. This mask works like a toggle key. If set once, the next call to the function will unset this mask.• DM_VADEVTS – voice activity detector (VAD) event notification (used in conjunction with the continuous speech processing (CSP) API library only)• DM_CONVERGED – echo cancellation convergence notification (used in conjunction with the Dialogic® Continuous Speech Processing (CSP) API library only) <p>On Springware boards, one or more of the following bits can be set:</p> <ul style="list-style-type: none">• 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_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 queuing 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_setevtmask()`

```
/* Set event mask to collect digits */  
if (dx_setevtmask(chdev, DM_DIGITS) == -1) {
```

To disable DM_DIGITS (turn off the digits flag and stop queuing digits):

```
dx_setevtmask(DM_DIGOFF);  
dx_clrldigbuf(chdev); /*Clear out queue*/
```

The following outlines the synchronous or asynchronous handling of CST events:

<u>Synchronous Application</u>	<u>Asynchronous Application</u>
Call <code>dx_setevtmask()</code> to enable CST events.	Call <code>dx_setevtmask()</code> to enable CST events.
Call <code>dx_getevt()</code> to wait for CST events. Events are returned to the <code>DX_EBLK</code> structure.	Use Dialogic® Standard Runtime Library (SRL) to asynchronously wait for TDX_CST events.
	Use <code>sr_getevtdatap()</code> to retrieve <code>DX_CST</code> structure.

■ Cautions

- If you call this function on a busy device, and specify DM_DIGITS as the **mask** argument, the function will fail.
- On Linux, events are preserved between `dx_getevt()` function calls. The event that was set remains the same until another call to `dx_setevtmask()` or `dx_wtring()` changes it. See `dx_wtring()` for more information on how it changes the event mask.
- On Linux, in a TDM bus configuration, when a voice resource is not listening to a network device, it may report spurious silence-off transitions and ring events if the events are enabled. To eliminate this problem:
 - Disable the ring and silence detection on unrouted/unlistened channels using the `dx_setevtmask()` function.
 - When you need to change the resource currently connected to your network device, do a half duplex disconnect of the current resource to disconnect the transmit time slot of the current resource (since two resources cannot transmit on the same time slot, although they can both listen), and a full duplex connect on the new resource using the appropriate listen/unlisten functions or the convenience functions `nr_scroute()` and `nr_scuroute()`.

■ 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 Dialogic® 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_SYSTEM

Error from operating system

***dx_setevtmsk()* — enable detection of call status transition (CST) events**

■ 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 <dxxxxlib.h>

main()
{
    int chdev;
    DX_EBLK eblk;
    .
    .

    /* open a channel with chdev as descriptor */
    if ((chdev = dx_open("dxxxxB1C1",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.

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

#define MAXCHAN 24

int cst_handler();

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


enable detection of call status transition (CST) events — dx_setevtmsk()

```
/* Set SRL to run in polled mode. */
srlmode = SR_POLLMODE;
if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1) {
    /* process error */
}
for (i=0; i<MAXCHAN; i++) {
    /* Set chname to the channel name, e.g., dxxxB1C1, dxxxB1C2,... */
    /* Open the device using dx_open(). chdev[i] has channel device
    * descriptor.
    */
    if ((chdev[i] = dx_open(chname,NULL)) == -1) {
        /* process error */
    }

    /* Use dx_setevtmsk() to enable call status transition events
    * on this channel.
    */
    if (dx_setevtmsk(chdev[i],
        DM_LCOFF|DM_LCON|DM_RINGS|DM_SILOFF|DM_SILON|DM_WINK) == -1) {
        /* process error */
    }

    /* Using sr_enbhdr(), set up handler function to handle call status
    * transition events on this channel.
    */
    if (sr_enbhdr(chdev[i], TDX_CST, cst_handler) == -1) {
        /* process error */
    }

    /* Use sr_waitevt to wait for call status transition event.
    * On receiving the transition event, TDX_CST, control is transferred
    * to the handler function previously established using sr_enbhdr().
    */
    .
    .
}
}

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:
            .
            .
    }
}
```

dx_setevtmsk() — enable detection of call status transition (CST) events

```
/* 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_minampl1, gtd_maxampl1, gtd_minampl2, gtd_maxampl2)`

Inputs:

- `short int gtd_minampl1` • minimum amplitude of the first frequency
- `short int gtd_maxampl1` • maximum amplitude of the first frequency
- `short int gtd_minampl2` • minimum amplitude of the second frequency
- `short int gtd_maxampl2` • maximum amplitude of the second frequency

Returns: `void`

Includes: `srllib.h`
`dxllib.h`

Category: Global Tone Detection

Mode: synchronous

Platform: HMP Software, Springware boards

■ 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.

Parameter	Description
<code>gtd_minampl1</code>	specifies the minimum amplitude of tone 1, in dB
<code>gtd_maxampl1</code>	specifies the maximum amplitude of tone 1, in dB
<code>gtd_minampl2</code>	specifies the minimum amplitude of tone 2, in dB
<code>gtd_maxampl2</code>	specifies the maximum amplitude of tone 2, in dB

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.

Default Value	Description
<code>GT_MIN_DEF</code>	Default value in dB for minimum GTD amplitude that can be entered for <code>gtd_minampl*</code> parameters.
<code>GT_MAX_DEF</code>	Default value in dB for maximum GTD amplitude that can be entered for <code>gtd_maxampl*</code> parameters.

■ Cautions

- If this function is called, then the amplitudes set will take effect for all tones added afterwards. To reset the amplitudes back to the defaults, call this function with the defines `GT_MIN_DEF` and `GT_MAX_DEF` for minimum and maximum defaults.

***dx_setgtdamp()* — set up the tone detection amplitudes**

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

■ Errors

None.

■ Example

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

#define TID 1;          /* Tone ID */

.
.
.
/*
 * Set amplitude for GTD;
 *   freq1 -30dBm to 0 dBm
 *   freq2 -30dBm to 0 dBm
 */
dx_setgtdamp(-30,0,-30,0);

/*
 * Build temporary simple dual tone frequency tone of
 * 950-1050 Hz and 475-525 Hz. using trailing edge detection, and
 * -30dBm to 0dBm.
 */
if (dx_blddt(TID, 1000, 50, 500, 25, TN_LEADING) == -1) {
    /* Perform system error processing */
    exit(3);
}
.
.
.
```

■ See Also

None.

`dx_sethook()`

Name: `int dx_sethook(chdev, hookstate, mode)`

Inputs:

<code>int chdev</code>	• valid channel device handle
<code>int hookstate</code>	• hook state (on-hook or off-hook)
<code>unsigned short mode</code>	• asynchronous/synchronous

Returns: 0 if successful
-1 if failure

Includes: `srllib.h`
`dxxlib.h`

Category: Configuration

Mode: asynchronous or synchronous

Platform: Springware boards

■ 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.

Parameter	Description
chdev	specifies the valid channel device handle obtained when the channel was opened using <code>dx_open()</code>
hookstate	forces the hookstate of the specified channel to on-hook or off-hook. The following values can be specified: <ul style="list-style-type: none">• <code>DX_OFFHOOK</code> – set to off-hook state• <code>DX_ONHOOK</code> – set to on-hook state
mode	specifies whether to run <code>dx_sethook()</code> asynchronously or synchronously. Specify one of the following: <ul style="list-style-type: none">• <code>EV_ASYNC</code> – run <code>dx_sethook()</code> asynchronously• <code>EV_SYNC</code> – run <code>dx_sethook()</code> synchronously (default)

Note: 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_sethook()* — provide control of the hook switch status**

- `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 return the hook state.

■ Synchronous Operation

By default, this function runs synchronously, and will return 0 upon successful completion.

`ATDX_HOOKST()` will also return hook state.

■ Cautions

None.

■ Errors

In asynchronous mode, the function returns immediately and a `TDX_SETHOOK` event is queued upon completion. If a failure occurs during operation, then a `TDX_ERROR` event will be queued. 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_SYSTEM`

Error from operating system

■ Example 1

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

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

main()
{
    int chdev;
    /* open a channel with chdev as descriptor */
    if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
        /* process error */
    }

    /* put the channel on-hook */
    if (dx_sethook(chdev,DX_ONHOOK,EV_SYNC) == -1) {
        /* error setting hook state */
    }
    .
    .
}
```

provide control of the hook switch status — dx_sethook()

```
/* take the channel off-hook */
if (dx_sethook(chdev,DX_OFFHOOK,EV_SYNC) == -1) {
    /* error setting hook state */
}
.
.
}
```

■ Example 2

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

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

#define MAXCHAN 24

int sethook_hdlr();

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

    /* Set SRL to run in polled mode. */
    srlmode = SR_POLLMODE;
    if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1) {
        /* process error */
    }
    for (i=0; i<MAXCHAN; i++) {
        /* Set chnamep to the channel name - e.g, dxxxB1C1, dxxxB1C2,... */

        /* open a channel with chdev[i] as descriptor */
        if ((chdev[i] = dx_open(chnamep,NULL)) == -1) {
            /* process error */
        }

        /* Using sr_enbhdr(), set up handler function to handle sethook
         * events on this channel.
         */
        if (sr_enbhdr(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_enbhdr().
     */
    .
    .
}

int sethook_hdlr()
{
    DX_CST *cstp;
```

***dx_sethook()* — provide control of the hook switch status**

```
/* 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()`

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`
`dxxlib.h`

Category: Configuration

Mode: synchronous

Platform: HMP Software, Springware boards

■ 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()`.

Parameter	Description
dev	Specifies the valid channel or board device handle obtained when the channel or board was opened using <code>dx_open()</code> .
parm	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. For HMP Software, see Table 1 for board parameter defines and Table 3 for channel parameter defines. For Springware boards, see Table 2 for board parameter defines and Table 4 for channel parameter defines. <i>Note:</i> The parameters set in parm will remain valid after the device has been closed and reopened.
valuep	Points to the 4-byte variable that specifies the channel or board parameter to set. <i>Note:</i> 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.

***dx_setparm()* — set physical parameters of a channel or board device**

The *dxlib.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 HMP Software, supported board parameter defines are shown in Table 1.

Table 1. Voice Board Parameters (HMP Software)

Define	Bytes	Read/Write	Default	Description
DXBD_CHNUM	1	R	-	Channel Number. Number of channels on the board
DXBD_SYSCFG	1	R	-	System Configuration. On HMP, 1 is always returned.

For Springware boards, the supported board parameter defines are shown in Table 2.

Table 2. Voice Board Parameters (Springware boards)

Define	Bytes	Read/Write	Default	Description
DXBD_CHNUM	1	R	-	Channel Number. Number of channels on the board
DXBD_FLASHCHR	1	R/W	&	Flash character. Character that causes a hook flash when detected
DXBD_FLASHTM	2	R/W	50	Flash Time. Length of time onhook during flash (10 msec units)
DXBD_HWTYPE	1	R	-	Hardware Type - value can be: <ul style="list-style-type: none">• TYP_D40 – D/40 board• TYP_D41 – D/21, D/41, D/xxxSC board
DXBD_MAXPDOFF	2	R/W	50	Maximum Pulse Digit Off. Maximum time loop current may be off before the existing loop pulse digit is considered invalid and reception is reinitialized (10 msec units)
DXBD_MAXSLOFF	2	R/W	25	Maximum Silence Off. Maximum time for silence being off, during audio pulse detection (10 msec units)
DXBD_MFDELAY	2	R/W	6	MF Interdigit Delay. Sets the length of the silence period between tones during MF dialing (10 msec units).
DXBD_MFLKPTONE	2	R/W	10	MF Length of LKP Tone. Specifies the length of the LKP tone during MF dialing. Maximum value: 15 (10 msec units)
DXBD_MFMINON	2	R/W	0	Minimum MF On. Sets the duration to be added to the standard MF tone duration before the tone is detected. The minimum detection duration is 65 msec for KP tones and 40 msec for all other tones (10 msec units).

set physical parameters of a channel or board device — dx_setparm()

Table 2. Voice Board Parameters (Springware boards) (Continued)

Define	Bytes	Read/Write	Default	Description
DXBD_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).
DXBD_MINIPD	2	R/W	25	Minimum Loop Interpulse Detection. Minimum time between loop pulse digits during loop pulse detection (10 msec units)
DXBD_MINISL	2	R/W	25	Minimum Interdigit Silence. Minimum time for silence on between pulse digits for audio pulse detection (10 msec units)
DXBD_MINLCOFF	2	R/W	0	Minimum Loop Current Off. Minimum time before loop current drop message is sent (10 msec units)
DXBD_MINOFFHKTM	2	R/W	250	Minimum offhook time (10 msec units)
DXBD_MINPDOFF	1	R/W	2	Minimum Pulse Detection Off. Minimum break interval for valid loop pulse detection (10 msec units)
DXBD_MINPDON	1	R/W	2	Minimum Pulse Detection On. Minimum make interval for valid loop pulse detection (10 msec units)
DXBD_MINSLOFF	1	R/W	2	Minimum Silence Off. Minimum time for silence to be off for valid audio pulse detection (10 msec units)
DXBD_MINSLOON	1	R/W	1	Minimum Silence On. Minimum time for silence to be on for valid audio pulse detection (10 msec units)
DXBD_MINTIOFF	1	R/W	5	Minimum DTI Off. Minimum time required between rings-received events (10 msec units)
DXBD_MINTION	1	R/W	5	Minimum DTI On. Minimum time required for rings received event (10 msec units)
DXBD_OFFHDLY	2	R/W	50	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).
DXBD_P_BK	2	R/W	6	Pulse Dial Break. Duration of pulse dial off-hook interval (10 msec units)
DXBD_P_IDD	2	R/W	100	Pulse Interdigit Delay. Time between digits in pulse dialing(10 msec units)
DXBD_P_MK	2	R/W	4	Pulse Dial Make. Duration of pulse dial offhook interval (10 msec units)
DXBD_PAUSETM	2	R/W	200	Pause Time. Delay caused by a comma in the dialing string (10 msec units)
DXBD_R_EDGE	1	R/W	ET_ROFF	Ring Edge. Detection of ring edge, values can be: <ul style="list-style-type: none"> • ET_RON – beginning of ring • ET_ROFF – end of ring
DXBD_R_IRD	2	R/W	80	Inter-ring Delay. Maximum time to wait for the next ring (100 msec units). Used to distinguish between calls. Set to 1 for T-1 applications.
DXBD_R_OFF	2	R/W	5	Ring-off Interval. Minimum time for ring not to be present before qualifying as “not ringing” (100 msec units)
DXBD_R_ON	2	R/W	3	Ring-on Interval. Minimum time ring must be present to qualify as a ring (100 msec units)

***dx_setparm()* — set physical parameters of a channel or board device**

Table 2. Voice Board Parameters (Springware boards) (Continued)

Define	Bytes	Read/Write	Default	Description
DXBD_S_BNC	2	R/W	4	Silence and Non-silence Debounce. Length of a changed state before call status transition message is generated (10 msec units)
DXBD_SYSCFG	1	R	-	System Configuration. JP8 status for D/4x boards. <ul style="list-style-type: none"> • 0 – loop start interface (JP8 in) • 1 – DTI/xxx interface (JP8 out)
DXBD_T_IDD	2	R/W	5	DTMF Interdigit delay. Time between digits in DTMF dialing (10 msec units)
DXBD_TTDATA	1	R/W	10	DTMF length (duration) for dialing (10 msec units)

■ **Channel Parameter Defines**

For HMP Software, the supported channel parameter defines are shown in Table 3. All time units are in multiples of 10 msec unless otherwise noted.

Table 3. Voice Channel Parameters (HMP Software)

Define	Bytes	Read/Write	Default	Description
DXCH_DFLAGS	2	R/W	0	Specifies leading or trailing edge for DTMF detection. This flag affects the reporting timing of CST digit detection events. When detection of CST digits is enabled, the setting of DXCH_DFLAGS allows changing the reporting edge of digits. Valid values are: <ul style="list-style-type: none"> • 0 – leading edge (default). Digit event is immediately reported upon detection (leading edge). • 1 – trailing edge. Digit event reporting is delayed to its trailing edge, when its detection ceases. See Example 2 for sample code.
DXCH_EC_ACTIVE	2	R/W	0	Echo cancellation. Specifies whether the echo cancellation feature is enabled or disabled. Valid values are: <ul style="list-style-type: none"> • 0 – disabled • 1 – enabled
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: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • 6000 – 6 kHz sampling rate • 8000 – 8 kHz sampling rate
DXCH_SCRFEATURE	2	R/W	-	Silence Compressed Record (SCR). Valid values are: <ul style="list-style-type: none"> • DXCH_SCRDISABLED – SCR feature disabled • DXCH_SCREENABLED – SCR feature enabled
DXCH_XFERBUFSIZE	4	R	16 kbytes	Transfer buffer size. Returns the bulk queue buffer size as set by the <i>dx_setchxfercnt()</i> function.

set physical parameters of a channel or board device — dx_setparm()

For Springware boards, the supported channel parameter defines are shown in Table 4. All time units are in multiples of 10 msec unless otherwise noted.

Table 4. Voice Channel Parameters (Springware boards)

Define	Bytes	Read/Write	Default	Description
DXCH_CALLID			disabled	Enables or disables caller ID for the channel. Valid values are: <ul style="list-style-type: none"> • DX_CALLIDENABLE • DX_CALLIDDISABLE (default)
DXCH_DFLAGS	2	R/W	0	Specifies leading or trailing edge for DTMF detection. This flag affects the reporting timing of CST digit detection events. When detection of CST digits is enabled, the setting of DXCH_DFLAGS allows changing the reporting edge of digits. Valid values are: <ul style="list-style-type: none"> • 0 – leading edge (default). Digit event is immediately reported upon detection (leading edge). • 1 – trailing edge. Digit event reporting is delayed to its trailing edge, when its detection ceases. See Example 2 for sample code.
DXCH_DTINITSET	2	R/W	0	Specifies which DTMF digits to initiate play on. Values of different DTMF digits may be ORed together to form the bit mask. Possible values and the corresponding digits are: <ul style="list-style-type: none"> • DM_1 – DTMF digit 1 • DM_2 – DTMF digit 2 • DM_3 – DTMF digit 3 • DM_4 – DTMF digit 4 • DM_5 – DTMF digit 5 • DM_6 – DTMF digit 6 • DM_7 – DTMF digit 7 • DM_8 – DTMF digit 8 • DM_9 – DTMF digit 9 • DM_0 – DTMF digit 0 • DM_S – * • DM_P – # • DM_A – a • DM_B – b • DM_C – c • DM_D – d
DXCH_DTMFDEB	2	R/W	0	DTMF debounce time (record delay). Sets the minimum time for DTMF to be present to be considered valid. Used to remove false DTMF signals during recording. Increase the value for less sensitivity to DTMF.
DXCH_DTMFTLK	2	R/W	5	Sets the minimum time for DTMF to be present during playback to be considered valid. Increasing the value provides more immunity to talk-off/playoff. Set to -1 to disable.

dx_setparm() — set physical parameters of a channel or board device

Table 4. Voice Channel Parameters (Springware boards) (Continued)

Define	Bytes	Read/Write	Default	Description
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: <ul style="list-style-type: none"> • 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_NUMRXBUFFERS	2	R/W	2	Supported on Windows® only. 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	2	R/W	2	Supported on Windows® only. 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: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • 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. This parameter will work even if the application has been restarted after an exit.
DXCH_RXDATABUFSIZE	4	R/W	32 kbytes	Supported on Windows® only. 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_RxlottData() and dx_TxRxlottData() 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.
DXCH_T_IDD	2	R/W	5	Specifies DTMF Interdigit delay (time between digits in DTMF dialing)
DXCH_TTDATA	1	R/W	10	Specifies DTMF length (duration) for dialing.
DXCH_TXDATABUFSIZE	4	R/W	32 kbytes	Supported on Windows® only. Sets the size of the play buffers only that are used to transfer data 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.

Table 4. Voice Channel Parameters (Springware boards) (Continued)

Define	Bytes	Read/Write	Default	Description
DXCH_VARNUMBUFFERS	4	R/W	0	Supported on Windows® only. 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: <ul style="list-style-type: none"> • 1 (True) – more than 2 buffers • 0 (False) – 2 buffers
DXCH_XFERBUFSIZE	4	R/W	16 kbytes	Sets the size of both the play and record buffers used to transfer data between the application on the host and the driver. This parameter can be used with the dx_getparm() function. The largest available buffer size is 32 Kbytes (must be in multiples of 128); however, only certain discrete buffer size values are supported. See the dx_setchxfercnt() function reference for more information on supported values.

■ **Cautions**

- A constant cannot be used in place of **valuep**. The value of the parameter to be set must be placed in a variable and the address of the variable cast as void * must be passed to the function.
- When setting channel parameters, the channel must be open and in the idle state.
- When setting board parameters, all channels on that board must be idle.

■ **Errors**

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

- EDX_BADPARAM
Invalid parameter
- EDX_SYSTEM
Error from operating system

■ **Example 1**

This example shows how to set inter-ring delay on Springware boards.

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

main()
{
    int bddev, parmval;
    /* Open the board using dx_open(). Get board device descriptor in
     * bddev.
     */
    if ((bddev = dx_open("dxxxB1",NULL)) == -1) {
        /* process error */
    }
}
```

dx_setparm() — *set physical parameters of a channel or board device*

```
/* 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 */
. . .
}
```

■ **Example 2**

This example shows how to configure voice resources for the reporting of call status transition (CST) digit detection on the trailing edge.

Use **dx_setparm()** with **DXCH_DFLAGS** set to 1. Then use **dx_setevtmsk()** with **DM_DIGITS** to enable digit CST events on the channel. Edge selection choice takes effect in the voice channel immediately. Edge selection is active until another selection is made. To switch the edge selection, you must first turn off the current selection using **dx_setevtmsk()** with **DM_DIGOFF**.

The following are notes on the use of edge selection in digit reporting:

- This feature is limited to the reporting digits as part of CST events. Edge selection will not affect the regular digit buffering reporting done through the **dx_getdig()** function, nor will it affect digit termination timing when using any **DV_TPT** digit termination condition on any other relevant voice functions.
- The default digit type is DTMF. MF type of digits is not supported by this feature.
- The leading or trailing edge digit setting does not affect digit detection but rather affects the reporting of the digit. In other words, DTMF signal detection continues unaltered regardless of the digit edge reporting selection.
- The CST event is still reported as **DE_DIGITS** (and not as **DE_DIGOFF**) in the **DX_CST.cst_events** irrespective of the edge selection.

```
...
/* Set chname to a voice channel name, e.g., dxxxB1C1, dxxxB1C2,... */
if ((chdev = dx_open(chname,NULL)) == -1) {
    /* process error */
}
...

/* 1 - Trailing Edge, 0 - Leading Edge */
int parmval = 1;
if (dx_setparm(chdev, DXCH_DFLAGS, (void *)&parmval) == -1)
{
    /* process error */
}

/* Use dx_setevtmsk() to enable digit call status transition events on this channel.*/
if (dx_setevtmsk(chdev, DM_DIGITS) == -1)
{
    /* process error */
}
}
```

■ **See Also**

- **dx_getparm()**

dx_setsvcond()

Name: int dx_setsvcond(chdev, numblk, svcbp)

Inputs: int chdev • valid channel device handle
unsigned short numblk • number of DX_SVCB blocks
DX_SVCB * svcbp • pointer to array of DX_SVCB structures

Returns: 0 if success
-1 if failure

Includes: srllib.h
dxxlib.h

Category: Speed and Volume

Mode: synchronous

Platform: HMP Software, Springware boards

■ Description

The **dx_setsvcond()** function sets adjustments and adjustment conditions for all subsequent plays on the specified channel (until changed or cancelled).

Note: On HMP Software, before you can use the speed control feature, you must enable this feature in the [decoder] section of the CONFIG file. For more information, see the Configuration Guide applicable to your release or product.

An adjustment is a modification to play speed, play volume due to an adjustment condition such as start of play, or the occurrence of an incoming digit during play. This function uses the specified channel's Speed or Volume Modification Table. For more information about these tables, see the *Dialogic® 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)
- adjustment conditions (incoming digit, beginning of play)
- level/edge sensitivity for incoming digits

See **DX_SVCB**, on page 507, for a full description of the data structure. Up to 20 **DX_SVCB** blocks can be specified in the form of an array.

dx_setsvcond() — *set conditions that adjust speed or volume of play*

- 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.

Parameter	Description
chdev	specifies the valid channel device handle obtained when the channel was opened using dx_open()
numblk	specifies the number of DX_SVCB blocks in the array. Set to a value between 1 and 20.
svcbp	points to an array of DX_SVCB structures

■ **Cautions**

- Speed control is not supported for all voice coders. For more information on supported coders, see the speed control topic in the *Dialogic® Voice API Programming Guide*.
- On HMP Software, digits that are used for play adjustment may also be used as a terminating condition. If a digit is defined as both, then both actions are applied upon detection of that digit.
- On 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 HMP Software, when adjustment is associated with a DTMF digit, speed can be increased or decreased in increments of 1 (10%) only.
- On HMP Software, 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 Dialogic® 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_BADPROD

Function not supported on this board

EDX_SVADJBLKS

Invalid number of speed/volume adjustment blocks

set conditions that adjust speed or volume of play — dx_setsvcond()

EDX_SYSTEM

Error from operating system

■ Example

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

/*
 * Global Variables
 */
DX_SVCB svcb[ 10 ] = {
    /* BitMask AjustmentSize AsciiDigit DigitType */
    { SV_SPEEDTBL | SV_RELCURPOS, 1, '1', 0 }, /* 1 */
    { SV_SPEEDTBL | SV_ABSPOS, -4, '2', 0 }, /* 2 */
    { SV_VOLUMETBL | SV_ABSPOS, 1, '3', 0 }, /* 3 */
    { SV_SPEEDTBL | SV_ABSPOS, 1, '4', 0 }, /* 4 */
    { SV_SPEEDTBL | SV_ABSPOS, 1, '5', 0 }, /* 5 */
    { SV_VOLUMETBL | SV_ABSPOS, 1, '6', 0 }, /* 6 */
    { SV_SPEEDTBL | SV_RELCURPOS, -1, '7', 0 }, /* 7 */
    { SV_SPEEDTBL | SV_ABSPOS, 6, '8', 0 }, /* 8 */
    { SV_VOLUMETBL | SV_RELCURPOS, -1, '9', 0 }, /* 9 */
    { SV_SPEEDTBL | SV_ABSPOS, 10, '0', 0 }, /* 10 */
};

main()
{
    int dxxxdev;

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

    /*
     * Set Speed and Volume Adjustment Conditions
     */
    if ( dx_setsvcond( dxxxdev, 10, svcb ) == -1 ) {
        printf( "Unable to Set Speed and Volume" );
        printf( " Adjustment Conditions\n" );
        printf( "Lasterror = %d Err Msg = %s\n",
            ATDV_LASTERR( dxxxdev ), ATDV_ERRMSGP( dxxxdev ) );
        dx_close( dxxxdev );
        exit( 1 );
    }

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

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

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

dx_setsvcond() — *set conditions that adjust speed or volume of play*

■ **See Also**

- [dx_clrsvcond\(\)](#)
- [DX_SVCB](#) structure
- [dx_setsvmt\(\)](#)
- [dx_getcursv\(\)](#)
- [dx_getsvmt\(\)](#)
- [dx_adjsv\(\)](#)
- speed and volume modification tables in *Dialogic® Voice API Programming Guide*

`dx_setsvmt()`

Name: `int dx_setsvmt(chdev, tabletype, svmt, flag)`

Inputs:

<code>int chdev</code>	• valid channel device handle
<code>unsigned short tabletype</code>	• type of table to update (speed or volume)
<code>DX_SVMT * svmt</code>	• pointer to speed or volume modification table to modify
<code>unsigned short flag</code>	• optional modification flag

Returns: 0 if success
-1 if failure

Includes: `srllib.h`
`dxxlib.h`

Category: Speed and Volume

Mode: synchronous

Platform: HMP Software, Springware boards

■ 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.

Note: On HMP Software, before you can use the speed control feature, you must enable this feature in the [decoder] section of the CONFIG file. For more information, see the Configuration Guide applicable to your release or product.

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 *Dialogic® Voice API Programming Guide*.

For more information on speed and volume modification tables, refer to `DX_SVMT`, on page 511, and see also the *Dialogic® Voice API Programming Guide*.

Parameter	Description
<code>chdev</code>	specifies the valid channel device handle obtained when the channel was opened using <code>dx_open()</code>
<code>tabletype</code>	specifies whether to update the speed modification table or the volume modification table: <ul style="list-style-type: none"> • <code>SV_SPEEDTBL</code> – update the speed modification table values • <code>SV_VOLUMETBL</code> – update the volume modification table values

dx_setsvmt() — *change default values of the speed or volume modification table*

Parameter	Description
svmtp	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: <ul style="list-style-type: none">• SV_SETDEFAULT – reset the table to its default values. See the <i>Dialogic® Voice API Programming Guide</i> for a list of default values. In this case, the DX_SVMT pointed to by svmtp 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 HMP Software, 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 Dialogic® 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_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

change default values of the speed or volume modification table — dx_setsvmt()

■ Example

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

/*
 * Global Variables
 */

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

    /*
     * Open the Voice Channel Device and Enable a Handler
     */
    if ( ( dxxxxdev = dx_open( "dxxxxB1C1", 0 ) ) == -1 ) {
        perror( "dxxxxB1C1" );
        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( dxxxxdev, SV_VOLUMETBL, &svmt, 0 ) == -1 ) {
        printf( "Unable to Set the Volume Modification Table\n" );
        printf( "Lasterror = %d Err Msg = %s\n",
            ATDV_LASTERR( dxxxxdev ), ATDV_ERRMSGP( dxxxxdev ) );
        dx_close( dxxxxdev );
        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( dxdev ) != 0 ) {
    perror( "close" );
}

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

■ See Also

- [dx_adjsv\(\)](#)
- [dx_getcursv\(\)](#)
- [dx_getsvmt\(\)](#)
- speed and volume modification tables in *Dialogic[®] Voice API Programming Guide*
- [DX_SVMT](#) data structure

`dx_settone()`

Name: `int dx_settone(chdev, toneid, tngenp)`

Inputs:

- `int chdev` • valid channel device handle
- `int toneid` • tone identifier
- `TN_GEN *tngenp` • pointer to the Tone Generation Template structure

Returns: 0 if success
-1 if failure

Includes: `dxxxlib.h`
`srllib.h`

Category: Global Tone Generation (GTG)

Mode: Synchronous

Platform: Springware boards Linux

■ Description

The `dx_settone()` function adds a global tone generation (GTG) tone template defined by `TN_GEN` to the firmware. This tone template can be later used by the application for tone-initiated record. In previous versions, an application had to use the built-in tone of fixed frequency and amplitude to provide notification of start-of-record. The duration of the tone may be changed; however, the units of duration are 200 msec, thus limiting the shortest beep to 200 msec.

The customization of record pre-beep lets the user select the frequencies, amplitudes, and duration of the beep being played prior to record. The `dx_bldtngen()` function is used to build the tone definition in `tngenp`. The `dx_settone()` function is then used to download this GTG tone template to the firmware for the channel device `chdev`. The `toneid` parameter for record pre-beep must be set to `TID_RECBEEP`.

Once the GTG tone template has been set in firmware, the application may use the customized tone preceding a record by specifying the `RM_TONE` and `RM_USERTONE` bits in the `mode` parameter of `dx_rec()` (or other record function). If `RM_USERTONE` is not set but `RM_TONE` is set, then the built-in tone will be played prior to initiating a record. This approach maintains existing functionality.

Parameter	Description
<code>chdev</code>	specifies the valid channel device handle obtained when the channel was opened using <code>dx_open()</code>
<code>toneid</code>	specifies the user-defined tone identifier For record pre-beep, set this parameter to <code>TID_RECBEEP</code> .
<code>tngenp</code>	points to the <code>TN_GEN</code> template structure, which defines the frequency, amplitude, and duration of a single- or dual-frequency tone. See <code>TN_GEN</code> , on page 524, for a full description of this structure.

dx_settone() — adds a GTG tone template

■ Cautions

- This function will fail if an invalid device handle is specified.
- Only call this function during initialization. Do not call this function after a [dx_playtone\(\)](#) has been initiated.

■ Errors

If this function returns -1 to indicate failure, call the Dialogic® 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

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

main()
{
    int      ChDev;          /* Channel Device handle */
    TN_GEN   tngen;         /* Tone Generation Template */
    DV_TPT   tpt;          /* Termination Parameter Table for record */

    /* Open board 1 channel 1 device */
    if ((ChDev = dx_open("dxxxB1C1", 0)) == -1) {
        printf("Cannot open channel dxxxB1C1.  errno = %d", errno);
        exit(1);
    }

    /*
     * Build a Single Tone Generation Template.
     * Frequency = 1000Hz, Amplitude = -10db and
     * Duration of 2 * 200msec = 400msec
     */
    dx_bldtngen(&tngen, 1000, 0, -10, 0, 2);

    /* Set the Tone Generation Template in firmware for record pre-beep */
    if (dx_settone(ChDev, TID_RECBEEP, &tngen) == -1) {
        printf("Error message = %s", ATDV_ERRMSGP(dev));
        exit(1);
    }

    /* Now issue a record using this tone */
    dx_clrtpt(&tpt, 1);
    tpt.tp_type = IO_EOT;
    tpt.tp_termno = DX_MAXDTMF;
    tpt.tp_length = 1;
    tpt.tp_flags = TF_MAXDTMF;

    if (dx_recf(ChDev, "usertone.vox", &tpt, RM_TONE|RM_USERTONE) == -1) {
        printf("Error message = %s", ATDV_ERRMSGP(dev));
        exit(1);
    }
}
```

■ See Also

- [dx_bldtngen\(\)](#)
- [dx_rec\(\)](#)

`dx_settonelen()`

Name: `int dx_settonelen(tonelength)`

Inputs: `int tonelength` • tone duration

Returns: 0 if successful

Includes: `srllib.h`
`dxxplib.h`

Category: Configuration

Mode: synchronous

Platform: Springware boards Windows

■ Description

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.

Parameter	Description
<code>tonelength</code>	specifies the duration of the tone in 200 ms units. Default: 1 (200 ms). Range: 1 - 65535.

■ 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.

***dx_settonelen()* — change the duration of the built-in beep tone**

■ Example

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

int chdev;                /* channel descriptor */
DV_TPT tpt;              /* termination parameter table */
DX_XPB xpb;              /* I/O transfer parameter block */
.
.
.

/* Increase beep tone len to 800ms */
dx_settonelen (4);

/* 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(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, RM_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_playvox\(\)](#)
- [dx_rec\(\)](#)
- [dx_reciottdata\(\)](#)
- [dx_recvox\(\)](#)

dx_setuio()

Name: int dx_setuio(uioblk)

Inputs: uioblk • DX_UIO data structure

Returns: 0 if success
 -1 if failure

Includes: srllib.h
 dxxxlib.h

Category: I/O

Mode: synchronous

Platform: HMP Software, Springware boards

■ 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 513 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 502 for more information.

For more information on working with user-defined I/O functions, see the Application Development Guidelines chapter in the *Dialogic® Voice API Programming Guide*.

Parameter	Description
uioblk	specifies the DX_UIO structure, a user-defined I/O structure

■ 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 HMP Software, 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.

■ **Example**

```
#include <stdio.h>
#include <srllib.h>
#include <dxxlib.h> /* voice library header file */

int cd; /* channel descriptor */
DX_UIO myio; /* user definable I/O structure */

/*
 * User defined I/O functions
 */
int my_read9(fd,ptr,cnt)
int fd;
char * ptr;
unsigned cnt;
{
    printf("My read\n");
    return(read(fd,ptr,cnt));
}

/*
 * my write function
 */
int my_write(fd,ptr,cnt)
int fd;
char * ptr;
unsigned cnt;
{
    printf("My write \n");
    return(write(fd,ptr,cnt));
}

/*
 * my seek function
 */
long my_seek(fd,offset,whence)
int fd;
long offset;
int whence;
{
    printf("My seek\n");
    return(lseek(fd,offset,whence));
}

void main(argc,argv)
int argc;
char *argv[];
{
    .
    . /* Other initialization */
    .
    DX_UIO uioblk;

    /* Initialize the UIO structure */
    uioblk.u_read=my_read;
    uioblk.u_write=my_write;
    uioblk.u_seek=my_seek;
}
```

dx_setuio() — *install user-defined I/O functions*

```
/* Install my I/O routines */
dx_setuio(uioblk);
vodat_fd = dx_fileopen("JUNK.VOX",O_RDWR|O_BINARY);

/*This block uses standard I/O functions */
iott->io_type = IO_DEV|IO_CONT
iott->io_fhandle = vodat_fd;
iott->io_offset = 0;
iott->io_length = 20000;

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

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

/*This block uses my I/O functions */
iott->io_type = IO_DEV|IO_UIO|IO_EOT
iott->io_fhandle = vodat_fd;
iott->io_offset = 10003;
iott->io_length = 20000;

devhandle = dx_open("dxxxB1C1", 0);
dx_sethook(devhandle, DX_ONHOOK,EV_SYNC)
dx_wtring(devhandle,1,DX_OFFHOOK,EV_SYNC);
dx_clrdigbuf;
if(dx_rec(devhandle,iott,(DX_TPT*)NULL,RM_TONE|EV_SYNC) == -1) {
    perror("");
    exit(1);
}

dx_clrdigbuf(devhandle);
if(dx_play(devhandle,iott,(DX_TPT*)EV_SYNC) == -1 {
    perror("");
    exit(1);
}
dx_close(devhandle);
}
```

■ See Also

- [dx_play\(\)](#)
- [dx_playiottdata\(\)](#)
- [dx_rec\(\)](#)
- [dx_reciottdata\(\)](#)

`dx_SetWaterMark()`

Name: `int dx_SetWaterMark(hBuffer, parm_id, value)`

Inputs:

<code>int hBuffer</code>	• circular stream buffer handle
<code>int parm_id</code>	• <code>LOW_MARK</code> or <code>HIGH_MARK</code>
<code>int value</code>	• value of water mark in bytes

Returns: 0 if successful
-1 if failure

Includes: `srllib.h`
`dxxlib.h`

Category: streaming to board

Mode: synchronous

Platform: HMP Software

■ Description

The `dx_SetWaterMark()` function sets the low and high water marks for the specified stream buffer. If you don't use this function, default values are in place for the low and high water marks based on the stream buffer size. See parameter description table for more information.

When setting the low and high water mark values for the stream buffer, do so in conjunction with the buffer size in `dx_OpenStreamBuffer()`. For hints and tips on setting water mark values, see the streaming to board topic in the *Dialogic® 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.

Parameter	Description
<code>hBuffer</code>	specifies the circular stream buffer handle
<code>parm_id</code>	specifies the type of water mark. Valid values are: <ul style="list-style-type: none"> • <code>LOW_MARK</code> – low water mark, which by default is set to 10% of the stream buffer size • <code>HIGH_MARK</code> – high water mark, which by default is set to 90% of the stream buffer size
<code>value</code>	specifies the value of the water mark in bytes

■ 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

```
#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
 dxxplib.h

Category: I/O

Mode: asynchronous or synchronous

Platform: HMP Software, Springware boards

■ Description

The **dx_stopch()** function forces termination of currently active I/O functions on a channel. It forces a channel in the busy state to become idle. If the channel specified in **chdev** already is idle, **dx_stopch()** has no effect and will return a success.

Running this function asynchronously will initiate **dx_stopch()** without affecting processes on other channels.

Running this function synchronously within a process does not block other processing. Other processes continue to be serviced.

When you issue **dx_stopch()** to terminate an I/O function, the termination reason returned by **ATDX_TERMMSK()** is TM_USRSTOP. However, if **dx_stopch()** terminates a **dx_dial()** function with call progress analysis, use **ATDX_CPTERM()** to determine the reason for call progress analysis termination, which is CR_STOPD.

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: <ul style="list-style-type: none">• 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_STOPGETEVT – If this bit is set and dx_stopch() is issued during dx_getevt(), TDX_CST event is generated with reason of DE_STOPGETEVT.• EV_STOPWTRING – (Windows[®] only) If this bit is set and dx_stopch() is issued during dx_wtring(), EDX_WTRINGSTOP error is generated.• IGNORESTATE – (Windows[®] only) Ignores the busy/idle state of the channel. Performs a stop on the channel regardless of whether the channel is busy or idle. If this flag is used, the function will not check for a busy state on the channel and will issue a stop even if the channel is busy.

■ 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_dialtpt()** without call progress analysis enabled

The functions will continue to run normally, and **dx_stopch()** will return a success. For **dx_dial()** or **dx_dialtpt()**, 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 that **dx_stopch()** was called.
- When calling **dx_stopch()** from a signal handler, **mode** must be set to **EV_ASYNC**.
- On Linux, when issued on a channel that is already idle, **dx_stopch()** will return an event, **TDX_NOSTOP**, to specify that no STOP was needed or issued. To use this functionality, “OR” the mode flag with the **EV_NOSTOP** flag. This does not affect the existing functionality of **dx_stopch()**. If a function is in progress when **dx_stopch()** is called with the **EV_NOSTOP** flag, that function will be stopped as usual and **EV_NOSTOP** will be ignored.
- On Linux, an application can use **dx_stopch()** from within a signal handler to stop the **dx_getevt()** function. To do so, “OR” the mode flag with the **EV_STOPGETEVT** flag. The **dx_getevt()** function will successfully return with the event **DE_STOPGETEVT**.
- On Windows, 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_STOPGETEVT** and **EV_STOPWTRING** flags, respectively, to stop these functions. In these cases, **dx_getevt()** will successfully return with the event **DE_STOPGETEVT** while

force termination of currently active I/O functions — dx_stopch()

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_BADPARAM
Invalid parameter

EDX_SYSTEM
Error from operating system

■ Example

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

main()
{
    int chdev, srlmode;

    /* Set SRL to run in polled mode. */
    srlmode = SR_POLLMODE;
    if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1) {
        /* process error */
    }

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

    /* continue processing */
    .
    .

    /* Force the channel idle. The I/O function that the channel is
     * executing will be terminated, and control passed to the handler
     * function previously enabled, using sr_enbhdr(), 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_dialtpt\(\)](#)
- [dx_getdig\(\)](#)
- [dx_getdigEx\(\)](#)

dx_stopch() — *force termination of currently active I/O functions*

- ***dx_play()***
- ***dx_playf()***
- ***dx_playiottdata()***
- ***dx_playtone()***
- ***dx_playvox()***
- ***dx_rec()***
- ***dx_recf()***
- ***dx_reciottdata()***
- ***dx_recm()***
- ***dx_recmf()***
- ***dx_recvox()***
- ***ATDX_TERMMSK()***
- ***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: srl.lib.h
dxx.lib.h

Category: Configuration

Mode: synchronous

Platform: Springware boards Windows

■ 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 progress 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**

```
/*$ dx_TSFStatus( ) example $*/

#include <stdio.h>
#include <dxxlib.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 Dialogic Configuration Manager\n");
            break;

        default:
            break;
    }
}
```

■ **See Also**

- [**dx_initcallp\(\)**](#)

`dx_TxlottData()`

Name: `int dx_TxlottData(chdev, iottp, lpTerminations, wType, lpParams, mode)`

Inputs:

<code>int chdev</code>	• valid channel device handle
<code>DX_IOTT *iottp</code>	• pointer to I/O Transfer Table
<code>DV_TPT *lpTerminations</code>	• pointer to Termination Parameter Table
<code>int wType</code>	• data type
<code>LPVOID lpParams</code>	• pointer to data type-specific information
<code>int mode</code>	• function mode

Returns: 0 if successful
-1 if error

Includes: `srllib.h`
`dxxlib.h`

Category: Analog Display Services Interface (ADSI)

Mode: asynchronous or synchronous

Platform: Springware boards

■ Description

The `dx_TxlottData()` 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.

Parameter	Description
<code>chdev</code>	specifies the valid channel device handle obtained when the channel was opened using <code>dx_open()</code> .
<code>iottp</code>	points to the I/O Transfer Table structure. The source of message(s) to be transmitted is specified by this transfer table. This is the same <code>DX_IOTT</code> structure used in <code>dx_playiottdata()</code> and <code>dx_reciottdata()</code> . See DX_IOTT , on page 502, for more information on this data structure.
<code>lpTerminations</code>	points to the Termination Parameter Table Structure, <code>DV_TPT</code> , which specifies termination conditions for the device handle. Supported values are: <ul style="list-style-type: none"> • <code>DX_MAXTIME</code> For more information on this structure, see DV_TPT , on page 481.
<code>wType</code>	specifies the type of data to be transmitted. To transmit ADSI data, set <code>wType</code> to <code>DT_ADSI</code> .

***dx_TxIottData()* — transmit data on a specified channel**

Parameter	Description
lpParams	points to information specific to the data type specified in wType . The format of the parameter block depends on wType . For ADSI data, set lpParams to point to an ADSI_XFERSTRUC structure. For more information on this structure, see ADSI_XFERSTRUC , on page 474.
mode	specifies how the function should execute: <ul style="list-style-type: none">• EV_ASYNC – asynchronous• EV_SYNC – synchronous

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_MAXTIME	Maximum function time exceeded
TM_USRSTOP	Function stopped by user

■ **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.

■ **Errors**

In asynchronous mode, the function returns immediately and a **TDX_TXDATA** event is queued upon completion. 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_BADPARAM	Invalid data mode
EDX_BUSY	Channel already executing I/O function
EDX_SYSTEM	Error from operating system

■ **Example**

```
// 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) {
        // process error
        exit(1);
    }

    if ((chdev = dx_open(devnamep, 0)) == -1) {
        fprintf(stderr, "Error opening channel %s\n", 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, -1);

    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: Springware boards

■ 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.

Parameter	Description
chdev	specifies the valid channel device handle obtained when the channel was opened using dx_open() .
lpTxIott	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 502, for more information on this data structure.

start a transmit-initiated reception of data — dx_TxRxIottData()

Parameter	Description
lpTxTerminations	points to the Termination Parameter Table structure, DV_TPT, which specifies termination conditions for the device handle. Supported values are: <ul style="list-style-type: none">• DX_MAXTIME For more information on this structure, see DV_TPT , on page 481.
lpRxIott	points to the I/O Transfer Table structure. lpRxIott specifies the destination of the messages to be received. This is the same DX_IOTT structure used in dx_playiottdata() and dx_reciottdata() .
lpRxTerminations	points to the Termination Parameter Table structure, DV_TPT, which specifies termination conditions for the device handle. Supported values are: <ul style="list-style-type: none">• DX_MAXTIME For more information on this structure, see DV_TPT , on page 481.
wType	specifies the type of data to be transmitted and received. To transmit and receive ADSI data, set wType to DT_ADSI.
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 wType . For ADSI data, set lpParams to point to an ADSI_XFERSTRUC parameter block structure. For more information on this structure, see ADSI_XFERSTRUC , on page 474.
mode	specifies how the function should execute: <ul style="list-style-type: none">• EV_ASYNC – asynchronous• EV_SYNC – synchronous

The transmit portion of the [dx_TxRxIottData\(\)](#) function will continue until one of the following occurs:

- all data specified in [DX_IOTT](#) has been transmitted
- [dx_stopch\(\)](#) is issued on the channel
- one of the conditions specified in [DV_TPT](#) is satisfied

The receive portion of the [dx_TxRxIottData\(\)](#) function will continue 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 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

***dx_TxRxIottData()* — start a transmit-initiated reception of data**

TM_ERROR	I/O device error
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 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

In asynchronous mode, the function returns immediately and either a TDX_TXDATA or TDX_RXDATA event is queued upon completion. 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_BADPARAM	Invalid data mode
EDX_BUSY	Channel already executing I/O function
EDX_SYSTEM	Error from operating system

■ Example

```
// Synchronous transmit initiated receive ADSI data

#include "srllib.h"
#include "dxxxlib.h"

main()
{
```

start a transmit-initiated reception of data — dx_TxRxIottData()

```
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_RDWR|O_CREAT|O_TRUNC|O_BINARY, 0666)) == -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;
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);
```

dx_TxRxIottData() — *start a transmit-initiated reception of data*

```
if (dx_TxRxIottData(chdev, &TxIott, NULL, &RxIott, &tpt, DT_ADSI,
    &adsimode, EV_SYNC) < 0) {
    fprintf(stderr, "ERROR: dx_TxIottData failed on Channel %s; error:
        %s\n", ATDV_NAMEEP(chdev), ATDV_ERRMSGP(chdev));
}
.
.
.
}
```

■ See Also

- [dx_TxIottData\(\)](#)
- [dx_RxIottData\(\)](#)

dx_unlisten()

Name: int dx_unlisten(chdev)
Inputs: int chdev • voice channel device handle
Returns: 0 on success
 -1 on error
Includes: srllib.h
 dxxplib.h
Category: TDM Routing
Mode: synchronous
Platform: HMP Software, Springware boards

■ Description

The **dx_unlisten()** function disconnects the voice receive channel from the TDM bus.

Note: The **dx_unlistenEx()** function is an extension of the **dx_unlisten()** function. See the **dx_unlistenEx()** function reference for more information.

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.

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

■ Cautions

This function will fail when an invalid channel device handle is specified.

■ Errors

If the function returns -1, use the Dialogic® 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
 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_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 failback failed
EDX_SYSTEM	Error from operating system

■ **Example**

```
#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_listenEx\(\)](#)
- [dx_unlistenEx\(\)](#)

dx_unlistenEx()

Name: int dx_unlistenEx(chdev, mode)

Inputs: int chdev • voice channel device handle
 unsigned short mode • mode flag

Returns: 0 on success
 -1 on error

Includes: srllib.h
 dxxlib.h

Category: TDM Routing

Mode: asynchronous or synchronous

Platform: HMP Software

■ Description

The **dx_unlistenEx()** function disconnects the voice receive channel from the TDM bus. This function is an extension of the **dx_unlisten()** function; it supports asynchronous as well as synchronous mode.

Calling **dx_listenEx()** to connect to a different TDM bus time slot automatically breaks an existing connection. Thus, when changing connections, you do not need to call **dx_unlistenEx()** first.

Parameter	Description
chdev	specifies the voice channel device handle obtained when the channel was opened using dx_open()
mode	specifies the mode of operation: <ul style="list-style-type: none">• EV_SYNC – synchronous mode (default)• EV_ASYNC – asynchronous mode

In synchronous mode, the voice receive channel is disconnected from the TDM bus upon return from the **dx_unlistenEx()** function. By default, this function runs in synchronous mode and returns a 0 to indicate that it has completed successfully. If a failure occurs, this function returns -1.

In asynchronous mode, a TDX_UNLISTEN event is queued upon successful completion of the unrouting. If a failure occurs during unrouting, a TDX_UNLISTEN_FAIL event is queued. In some limited cases, such as when invalid arguments are passed to the library, the function may fail before unrouting is attempted. In such cases, the function returns -1 immediately to indicate failure and no event is queued.

■ Cautions

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

***dx_unlistenEx()* — disconnect voice receive channel from TDM bus**

- When using this function in asynchronous mode, do not issue another unlisten operation on the same channel using either **dx_unlisten()** or **dx_unlistenEx()** until the TDX_UNLISTEN event is received. If you attempt to do this, the unlisten function will return failure.
- It is recommended that you use **dx_listenEx()** and **dx_unlistenEx()** in your application, rather than **dx_listen()** and **dx_unlisten()**. In particular, do not use both pairs of functions on the same channel. Doing so may result in unpredictable behavior.

■ **Errors**

If the function returns -1, use the Dialogic® 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

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 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

■ Example 1: Synchronous Mode

This example code for **dx_unlistenEx()** illustrates the synchronous mode of operation.

```
#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_unlistenEx(chdev, EV_SYNC) == -1) {
        printf("Error message = %s", ATDV_ERRMSGP(chdev));
        exit(1);
    }
}
```

■ Example 2: Asynchronous Mode

This example code for **dx_unlistenEx()** illustrates the asynchronous mode of operation.

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

main()
{
    int srlmode;

    /* Set SRL to run in polled mode. */
    srlmode = SR_POLLMODE;
    if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1) {
        /* process error */
    }

    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_unlistenEx(chdev, EV_ASYNC) == -1) {
        printf("Error message = %s", ATDV_ERRMSGP(chdev));
        exit(1);
    }

    /* Use sr_waitevt to wait for the TDX_UNLISTEN event */
}
```

■ See Also

- [dx_listenEx\(\)](#)
- [dx_listen\(\)](#)
- [dx_unlisten\(\)](#)

dx_wtcallid()

Name: int dx_wtcallid (chdev, nrings, timeout, bufferp)

Inputs:

int chdev	• valid channel device handle
int nrings	• number of rings to wait
short timeout	• time to wait for rings (in seconds)
unsigned char *bufferp	• pointer to where to return the caller ID information

Returns: 0 success
-1 error return code

Includes: srllib.h
dxxxlib.h

Category: Caller ID

Mode: synchronous

Platform: Springware boards

■ 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.

Parameter	Description
chdev	specifies the valid channel device handle obtained when the channel was opened using dx_open()
nrings	specifies the number of rings to wait before answering On Windows, valid values: ≥ 1 (Note: Minimum 2 for CLASS and ACLIP) On Linux, valid values: ≥ 2

Parameter	Description
timeout	<p>specifies the maximum length of time to wait for a ring</p> <p>Valid values (0.1-second units):</p> <ul style="list-style-type: none"> • ≥ 0 • -1 waits forever; never times out <p>If timeout is set to 0 and a ring event does not already exist, the function returns immediately.</p>
bufferp	<p>pointer to buffer where the calling line Directory Number (DN) is to be stored</p> <p><i>Note:</i> The application must allocate a buffer large enough to accommodate the DN.</p>

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_BADPARAM
Invalid parameter

EDX_BUSY
Channel is busy

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_CLIDOOA

Caller ID is out of area
(other information may be available using [dx_gtextcallid\(\)](#))

EDX_SYSTEM

Error from operating system

EDX_TIMEOUT

Time out limit is reached

■ **Example**

```
/*$ 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_gtcallid\(\)](#)
- [dx_setevtmsk\(\)](#)
- [dx_getevt\(\)](#)

`dx_wtring()`

Name: `int dx_wtring(chdev, nrings, hstate, timeout)`

Inputs:

<code>int chdev</code>	• valid channel device handle
<code>int nrings</code>	• number of rings to wait for
<code>int hstate</code>	• hook state to set after rings are detected
<code>int timeout</code>	• timeout, in seconds

Returns: 0 if successful
-1 if failure

Includes: `srllib.h`
`dxxlib.h`

Category: Configuration

Mode: synchronous

Platform: Springware boards

■ 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()`.

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

Parameter	Description
chdev	specifies the valid channel device handle obtained when the channel was opened using dx_open() function
nrings	specifies the number of rings to wait for before setting the hook state
hstate	sets the hookstate of the channel after the number of rings specified in nrings are detected. Valid values: <ul style="list-style-type: none">• 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: <ul style="list-style-type: none">• 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_BADPARAM

Invalid parameter

EDX_SYSTEM

Error from operating system

EDX_TIMEOUT

Timeout limit is reached

■ **Example**

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

nr_scroute() — *make a full or half-duplex connection*

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
dxxlib.h
faxlib.h (optional)
sctools.h

Category: TDM Routing

Mode: synchronous

Platform: HMP Software, Springware boards

■ **Description**

The **nr_scroute()** convenience function makes a full or half-duplex connection between two devices connected to the time division multiplexing (TDM) bus.

This convenience function is not a part of any library and is provided in a separate C source file called *sctools.c* in the *sctools* subdirectory.

The **nr_sc** prefix to the function signifies network (analog and digital) devices and resource (voice, and fax) devices accessible via the TDM bus.

Note: Fax functionality may be conditionally compiled in or out of the function using the FAXSC defines in the makefile provided with the function. For example, to compile in fax functionality, link with the fax library. Error message printing may also be conditionally compiled in or out by using the PRINTON define in the makefile.

Parameter	Description
devh1	specifies the valid channel device handle obtained when the channel was opened for the first device (the transmitting device for half duplex)
devtype1	specifies the type of device for devh1 : <ul style="list-style-type: none">• SC_VOX – voice channel device• SC_LSI – analog network (loop start interface) channel device• SC_FAX – fax channel device
devh2	specifies the valid channel device handle obtained when the channel was opened for the second device (the listening device for half duplex)
devtype2	specifies the type of device for devh1 . See devtype1 for a list of defines.
mode	specifies full or half-duplex connection. This parameter contains one of the following defines from <i>sctools.h</i> to specify full or half duplex: <ul style="list-style-type: none">• SC_FULLDUP – full-duplex connection (default)• SC_HALFDUP – half-duplex connection When SC_HALFDUP is specified, the function returns with the second device listening to the TDM bus time slot connected to the first device.

■ Cautions

- The **devtype1** and **devtype2** parameters must match the types of the device handles in **devh1** and **devh2**.
- If you have not defined FAXSC when compiling the *sctools.c* file, you cannot use this function to route fax channels.
- If you have not defined PRINTON in the makefile, errors will not be displayed.
- It is recommended that you do not use the **nr_scroute()** convenience function in high performance or high density applications because this convenience function performs one or more `xx_getxmitslot` invocations that consume CPU cycles unnecessarily.

■ Errors

None.

■ Example

See source code. The C source code for this function is provided in the *sctools.c* file located in the *sctools* subdirectory.

■ See Also

- [nr_scunroute\(\)](#)

***nr_scunroute()* — break a full or half-duplex connection**

nr_scunroute()

Name: int nr_scunroute(devh1, devtype1, devh2, devtype2, mode)

Inputs:

int devh1	• valid channel device handle
unsigned short devtype1	• type of device for devh1
int devh2	• valid channel device handle
unsigned short devtype2	• type of device for devh2
unsigned char mode	• half or full duplex connection

Returns: 0 on success
-1 on error

Includes: stdio.h
varargs.h
srllib.h
dxxlib.h
faxlib.h (optional)
sctools.h

Category: TDM Routing

Mode: synchronous

Platform: HMP Software, Springware boards

■ Description

The **nr_scunroute()** convenience function breaks a full or half-duplex connection between two devices connected to the time division multiplexing (TDM) bus.

This convenience function is not a part of any library and is provided in a separate C source file called *sctools.c* in the *sctools* subdirectory.

The **nr_sc** prefix to the function signifies network (analog and digital) devices and resource (voice, and fax) devices accessible via the TDM bus.

Note: Fax functionality may be conditionally compiled in or out of the function using the FAXSC defines in the makefile provided with the function. For example, to compile in fax functionality, link with the fax library. Error message printing may also be conditionally compiled in or out by using the PRINTON define in the makefile.

Parameter	Description
devh1	specifies the valid channel device handle obtained when the channel was opened for the first device (the transmitting device for half duplex)
devtype1	specifies the type of device for devh1 : <ul style="list-style-type: none">• SC_VOX – voice channel device• SC_LSI – analog (loop start interface) channel device• SC_FAX – fax channel device
devh2	specifies the valid channel device handle obtained when the channel was opened for the second device (the listening device for half duplex)
devtype2	specifies the type of device for devh1 . See devtype1 for a list of defines.
mode	specifies full or half-duplex connection. This parameter contains one of the following defines from <i>sctools.h</i> to specify full or half duplex: <ul style="list-style-type: none">• SC_FULLDUP – full-duplex connection (default)• SC_HALFDUP – half-duplex connection When SC_HALFDUP is specified, the function returns with the second device listening to the TDM bus time slot connected to the first device.

■ Cautions

- The **devtype1** and **devtype2** parameters must match the types of the device handles in **devh1** and **devh2**.
- If you have not defined FAXSC when compiling the *sctools.c* file, you cannot use this function to route fax channels.
- If you have not defined PRINTON in the makefile, errors will not be displayed.
- It is recommended that you do not use the **nr_scunroute()** convenience function in high performance or high density applications because this convenience function performs one or more `xx_getxmitslot` invocations that consume CPU cycles unnecessarily.

■ Errors

None.

■ Example

See source code. The C source code for this function is provided in the *sctools.c* file located in the *sctools* subdirectory.

■ See Also

- [nr_scroute\(\)](#)

This chapter provides information on events that may be returned by the Dialogic® Voice API software. The following topics are discussed:

- [Overview of Events](#) 467
- [Termination Events](#) 467
- [Unsolicited Events](#) 469
- [Call Status Transition \(CST\) Events](#) 470

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 *dxxlib.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 *Dialogic® Voice API Programming Guide*. For details on event management and event handling, see the *Dialogic® 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 Dialogic® Standard Runtime Library (SRL) functions such as **sr_waitevt()** and **sr_enbhdr()** 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 Dialogic® Voice API library:

TDX_CALLP

Termination event. Returned by **dx_dial()** or **dx_dialtpt()** 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 470 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()** or **dx_dialtpt()** to indicate that dialing without call progress analysis completed. Use **ATDX_TERMMASK()** 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()** and **dx_getdigEx()** to indicate completion of asynchronous digit collection from a channel digit buffer.

TDX_LISTEN

Termination event. Returned by **dx_listenEx()** to indicate completion of routing.

TDX_LISTEN_FAIL

Termination event. Returned by **dx_listenEx()** to indicate failure of routing.

TDX_NOSTOP

Termination event. Returned by **dx_stopch()**. On Linux, when issued on a channel that is already idle, **dx_stopch()** with **EV_NOSTOP** flag will return this event to indicate that no **STOP** was needed or issued.

TDX_PLAY

Termination event. Returned by play functions such as **dx_play()** to indicate completion of play.

TDX_PLAYTONE

Termination event. Returned by **dx_playtone()** and **dx_playtoneEx()** to indicate completion of play tone.

TDX_QUERYTONE

Termination event. Returned by **dx_querytone()** to indicate completion of query tone.

Events

TDX_QUERYTONE_FAIL

Termination event. Returned by [dx_querytone\(\)](#) to indicate failure of query tone.

TDX_RECORD

Termination event. Returned by record functions such as [dx_rec\(\)](#) to indicate completion of record.

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_UNLISTEN

Termination event. Returned by [dx_unlistenEx\(\)](#) to indicate completion of unrouting.

TDX_UNLISTEN_FAIL

Termination event. Returned by [dx_unlistenEx\(\)](#) to indicate failure of unrouting.

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 470.

The following unsolicited events may be returned by the Dialogic® Voice API library:

TDX_HIGHWATER

Unsolicited event. Generated when a high water mark is reached during a streaming to board operation.

TDX_LOWWATER

Unsolicited event. Generated when a low water mark is reached during a streaming to board operation.

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_VAD

Unsolicited event. Generated when the voice activity detector (VAD) detects voice energy during a [dx_reciottdata\(\)](#) recording operation. This event will only be generated when [dx_reciottdata\(\)](#) is set to `RM_VADNOTIFY`.

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 500. To retrieve CST events in an asynchronous environment, use the Dialogic® Standard Runtime Library (SRL) Event Management functions such as `sr_getevtdatap()`. Events are returned to the `DX_CST` structure.

Call Status Transition Events on Dialogic® HMP Software

On HMP Software, 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 (`est_data`) using `sr_getevtdatap()`.

DE_SILOFF

Call status transition event. Indicates non-silence detected on the channel.

DE_SILON

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.

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

Call Status Transition Events on Dialogic® 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_LCOFF

Call status transition event. Indicates loop current off.

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_SILON

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_STOPWTRING

Call status transition event. Indicates that the `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.

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 the Dialogic® Voice API library functions.

ADSI_XFERSTRUC

```
typedef struct ADSI_XFERSTRUC
{
    UINT    cbSize;
    DWORD   dwTxDataMode;
    DWORD   dxRxDataMode;
} 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 *dxlib.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. On return from the [dx_getctinfo\(\)](#) function, CT_DEVINFO contains the relevant device and device configuration information.

The valid values for each field of the CT_DEVINFO structure are defined in *ctinfo.h*, which is referenced by *dxxlib.h*.

■ Field Descriptions (HMP Software)

The fields of the CT_DEVINFO data structure are described as follows for HMP Software:

ct_prodid

Contains a valid product identification number for the device.

ct_devfamily

Specifies the device family. Possible values are:

- CT_DFDM3 – DM3 device
- CT_DFHMPDM3 – HMP device (Host Media Processing)

ct_devmode

Specifies the device mode. Possible values are:

- CT_DMRESOURCE – voice device
- CT_DMNETWORK – network device

ct_nettype

Specifies the type of network interface for the device. Possible values are:

- CT_NTIPT – IP connectivity
- CT_NTT1 – T1 digital network interface
- CT_NTE1 – E1 digital network interface

ct_busmode

Specifies the bus architecture used to communicate with other devices in the system. Possible values are:

- CT_BMSCBUS – TDM bus architecture
- CT_BMH100 – H.100 bus

- CT_BMH110 – H.110 bus

ct_busencoding

Describes the PCM encoding used on the bus. Possible values are:

- CT_BEULAW – mu-law encoding
- CT_BEALAW – A-law encoding
- CT_BELLAW – linear encoding
- CT_BEBYPASS – encoding is being bypassed

ct_ext_devinfo.ct_RFU

Not used in HMP.

ct_ext_devinfo.ct_net_devinfo.ct_prototype

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

■ **Field Descriptions (Springware boards)**

The fields of the CT_DEVINFO data structure are described as follows for Springware boards:

ct_prodid

Contains a valid product identification number for the device.

ct_devfamily

Specifies the device family. Valid value is:

- CT_DFD41E – analog or voice channel

ct_devmode

Specifies the device mode field. Valid value is:

- CT_DMNETWORK – analog channel available to process calls from the telephone network

ct_nettype

Specifies the type of network interface for the device. Valid value is:

- CT_NTANALOG – analog interface

ct_busmode

Specifies the bus architecture used to communicate with other devices in the system. Possible values are:

- CT_BMSCBUS – TDM bus architecture
- CT_H100 – H.100 bus

ct_busencoding

Describes the PCM encoding used on the bus. Possible values are:

- CT_BEULAW – Mu-law encoding
- CT_BEALAW – A-law encoding

ct_ext_devinfo.ct_rfu

Reserved for future use.

CT_DEVINFO — channel/time slot device information

■ **Example**

For an example of how to use the CT_DEVINFO structure, see the Example section for [dx_getctinfo\(\)](#).

DV_DIGIT

```
typedef struct DV_DIGIT {
    char dg_value[DG_MAXDIGS + 1]; /* ASCII values of digits */
    char dg_type[DG_MAXDIGS + 1]; /* Type of digits */
} DV_DIGIT;
```

■ Description

The DV_DIGIT data structure stores an array of digits. When `dx_getdig()` is called, the digits are collected from the firmware and transferred to the user's digit buffer. The digits are stored as an array inside the DV_DIGIT structure.

The DG_MAXDIGS define in `dxxlib.h` indicates the maximum number of digits that can be returned by a single call to `dx_getdig()`. The maximum size of the digit buffer varies with the board type and technology.

■ Field Descriptions

The fields of the DV_DIGIT data structure are described as follows:

`dg_value`

Specifies a null-terminated string of the ASCII values of the digits collected.

`dg_type`

Specifies an array (terminated by DG_END) of the digit types that correspond to each of the digits contained in the `dg_value` string.

Use the following defines to identify the digit type:

- DG_DTMF_ASCII – DTMF
- DG_MF_ASCII – MF
- DG_USER1 – GTD user-defined
- DG_USER2 – GTD user-defined
- DG_USER3 – GTD user-defined
- DG_USER4 – GTD user-defined
- DG_USER5 – GTD user-defined
- DG_END – Terminator for dg_type array

On Springware boards in Linux, use the following defines to identify the digit type:

- DG_DTMF – DTMF
- DG_LPD – loop pulse digit
- DG_DPD – DPD (dial pulse)
- DG_MF – 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

On Springware boards in Windows, use the following defines to identify the digit type:

- DG_DTMF_ASCII – DTMF
- DG_DPD_ASCII – DPD (dial pulse)
- DG_MF_ASCII – MF

DV_DIGIT — user digit buffer

- 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\(\)](#).

DV_DIGITEX

```
typedef struct dv_digitEx {
    short  numdigits;      // size in bytes of array
    char  *dg_valuep;     // ASCII value of digits
    char  *dg_typep;      // type of digits
} DV_DIGITEX;
```

■ Description

Supported on Linux only. The DV_DIGITEX data structure stores an array of digits. When a [dx_getdigEx\(\)](#) function is executed, the digits are collected from the firmware and transferred to the user's digit buffer. The digits are stored as an array in the DV_DIGITEX structure. After the function has completed, information on the digits detected is available to the application.

■ Field Descriptions

The fields of the DV_DIGITEX data structure are described as follows:

numdigits

Contains the size in bytes of the user-allocated array pointed to by dg_valuep and dg_typep.

dg_value

Points to a user-allocated array, which on return from [dx_getdigEx\(\)](#) is filled with a NULL-terminated string of the ASCII values of the digits collected.

dg_typep

Points to another user-allocated array. On return from [dx_getdigEx\(\)](#), this user-allocated array is filled with the digit types corresponding to each of the digits contained in the array (as pointed to by dg_valuep) and is terminated by DG_END. For the defines for the digit types, see [DV_DIGIT](#), on page 478.

■ Example

For an example of how to use this data structure, see the Example section for [dx_getdigEx\(\)](#).

DV_TPT

```
typedef struct DV_TPT {
    unsigned short tp_type;           /* Flags describing this entry */
    unsigned short tp_termno;        /* Termination Parameter number */
    unsigned short tp_length;        /* Length of terminator */
    unsigned short tp_flags;         /* Parameter attribute flag */
    unsigned short tp_data;          /* Optional additional data */
    unsigned short rfu;              /* Reserved */
    DV_TPT *tp_nextp;                /* Pointer to next termination
    * parameter if IO_LINK set */
}DV_TPT;
```

■ Description

The DV_TPT data structure specifies a termination condition for an I/O function. To specify multiple termination conditions for a function, use multiple DV_TPT structures configured as a linked list, an array, or a combined linked list and array, with each DV_TPT specifying a termination condition. The first termination condition that is met will terminate the I/O function.

For a list of functions in the I/O category, see [Chapter 1, “Function Summary by Category”](#). For more information on termination conditions, see the I/O terminations topic in the *Dialogic® Voice API Programming Guide*.

The DV_TPT structure is defined in the Standard Runtime Library (*srllib.h*).

- Notes:**
1. Not all termination conditions are supported by all I/O functions. Exceptions are noted in the description of the termination condition.
 2. Use the [dx_clrtp\(\)](#) 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 **HMP Software**, the supported termination conditions are:

- DX_DIGMASK – digit termination for a bit mask of digits received
- DX_DIGTYPE – digit termination for user-defined tone. The ASCII value set in the tp_length field must match a real DTMF tone (0-9, a-d, *, #).
- DX_IDDTIME – maximum delay between digits. This termination condition is only supported by the [dx_getdig\(\)](#) function.

- **DX_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. This termination condition is only supported by **dx_RxIottData()**, **dx_TxIottData()**, and **dx_TxRxIottData()**.
- **DX_MAXDTMF** – maximum number of digits received
- **DX_MAXSIL** – maximum length of silence. The range is 10 msec to 250 sec (25000 in 10 msec units).
- **DX_MAXTIME** – maximum function time. This termination condition is not supported by tone generation functions such as **dx_playtone()** and **dx_playtoneEx()**.
- **DX_TONE** – tone on or tone off termination for global tone detection (GTD)

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_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.
- number of digits – Applies when using DX_MAXDTMF, which specifies termination after a certain number of digits is received.
- digit type description – Applies when using DX_DIGTYPE, which specifies termination on a user-specified digit. Specify the digit type in the high byte and the ASCII digit value

DV_TPT — termination parameter table

in the low byte. See the global tone detection topic in the *Dialogic® 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:

Digit	Digit Define
0	DM_0
1	DM_1
2	DM_2
3	DM_3
4	DM_4
5	DM_5
6	DM_6
7	DM_7
8	DM_8
9	DM_9
*	DM_S
#	DM_P
a	DM_A
b	DM_B
c	DM_C
d	DM_D

- 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_CLRBEQ – History of this termination condition is cleared when the function begins. This bit overrides the TF_LEVEL bit. If both are set, the history will be cleared and no past history of this terminator will be taken into account.
- TF_CLREND – History of this termination condition is cleared when the function terminates. This bit has special meaning for DX_IDDTIME (interdigit delay). If set, the terminator will be started after the first digit is received; otherwise, the terminator will be started as soon as the function is started. This bit has no effect on HMP Software and will be ignored.
- TF_EDGE – Termination condition is edge-sensitive. Edge-sensitive means that the function will not terminate unless the condition occurs after the function starts. Refer to the table later in this section to see which termination conditions can be edge-sensitive and which can be level-sensitive. This bit has no effect on HMP Software 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 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 *Dialogic® Continuous Speech Processing API Library Reference*.
- **TF_LEVEL** – Termination condition is level-sensitive. Level-sensitive means that if the condition is satisfied when the function starts, termination will occur immediately. Termination conditions that can be level-sensitive have a history associated with them which records the state of the terminator before the function started. Refer to the table later in this section to see which termination conditions can be edge-sensitive and which can be level-sensitive. This bit has no effect on HMP Software 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:

Default	Define	Underlying Flags
TF_DIGMASK	(TF_LEVEL)	
TF_DIGTYPE	(TF_LEVEL)	
TF_IDDTIME	(TF_EDGE)	
TF_LCOFF	(TF_LEVEL TF_USE TF_CLREND)	
TF_MAXDTMF	(TF_LEVEL TF_USE)	
TF_MAXNOSIL	(TF_EDGE TF_USE)	
TF_MAXSIL	(TF_EDGE TF_USE)	
TF_MAXTIME	(TF_EDGE)	
TF_PMON	(TF_EDGE)	
TF_TONE	(TF_LEVEL TF_USE TF_CLREND)	

DV_TPT — termination parameter 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.
 4. 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.
The bitmap for the tp_flags field is as follows:

Bit	7	6	5	4	3	2	1	0
Name	rfu	rfu	units	ini	use	beg	end	level

For HMP Software, the following table shows the default sensitivity of a termination condition.

Termination Condition	Level-sensitive	Edge-sensitive
DX_DIGMASK	✓	
DX_DIGTYPE	✓	
DX_IDDTIME		✓
DX_MAXDTMF	✓	
DX_MAXSIL		✓
DX_MAXTIME		✓
DX_TONE	✓	

For **Springware boards**, the following table shows whether a termination condition can be level-sensitive or edge-sensitive.

Termination Condition	Level-sensitive	Edge-sensitive
DX_DIGMASK	✓	✓
DX_DIGTYPE	✓	✓
DX_IDDTIME		✓
DX_LCOFF	✓	✓
DX_MAXDTMF	✓	✓
DX_MAXNOSIL	✓	✓
DX_MAXSIL	✓	✓
DX_MAXTIME		✓
DX_PMON/DX_PMOFF		✓
DX_TONE	✓	✓

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.
- If tp_termno contains DX_PMON, tp_data can specify the maximum time of silence on.
- If tp_termno contains DX_TONE, tp_data can specify one of the following values:
DX_TONEOFF (for termination after a tone-off event)
DX_TONEON (for termination after a tone-on event)

tp_nextp

Points to the next DV_TPT structure in a linked list if the tp_type field is set to IO_LINK.

Table 5 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 5. DV_TPT Field Settings Summary

tp_termno	tp_type	tp_length	tp_flags: not set	tp_flags: set	tp_data	tp_nextp
DX_MAXDTMF	IO_LINK IO_EOT IO_CONT	max number of digits	bit 0: TF_EDGE bit 1: no clr* bit 2: no clr* bit 3: clr hist	TF_LEVEL* TF_CLREND TF_CLRBEG TF_USE*	N/A	pointer to next DV_TPT if linked list
DX_MAXSIL	IO_LINK IO_EOT IO_CONT	max length silence	bit 0: bit 1: no clr* bit 2: no clr* bit 3: clr hist bit 4: no-setinit bit 5: 100 msec*	TF_EDGE* TF_LEVEL TF_CLREND TF_CLRBEG TF_USE* TF_SETINIT TF_10MS	length of init silence	pointer to next DV_TPT in linked list
DX_MAXNOSIL	IO_LINK IO_EOT IO_CONT	max length non-silence	bit 0: TF_EDGE* bit 1: no clr* bit 1: no clr* bit 2: no clr* bit 3: clr hist bit 4: N/A bit 5: 100 msec*	TF_LEVEL TF_CLREND TF_CLRBEG TF_USE* N/A TF_10MS	N/A	pointer to next DV_TPT if linked list
DX_LCOFF	IO_LINK IO_EOT IO_CONT	max length loop current drop	bit 0: TF_EDGE bit 1: no clr bit 2: no clr* bit 3: clr hist bit 4: N/A bit 5: 100 msec*	TF_LEVEL* TF_CLREND* TF_CLRBEG TF_USE* N/A TF_10MS	N/A	pointer to next DV_TPT if linked list

DV_TPT — termination parameter table

Table 5. DV_TPT Field Settings Summary (Continued)

tp_termno	tp_type	tp_length	tp_flags: not set	tp_flags: set	tp_data	tp_nextp
DX_IDDTIME	IO_LINK IO_EOT IO_CONT	max length interdigit delay	bit 0: TF_EDGE* bit 1: start@call* bit 2: N/A bit 3: N/A bit 4: N/A bit 5: 100 msec*	N/A start@1st N/A N/A N/A TF_10MS	N/A	pointer to next DV_TPT if linked list
DX_MAXTIME	IO_LINK IO_EOT IO_CONT	max length function time	bit 0: TF_EDGE* bit 1: N/A bit 2: N/A bit 3: N/A bit 4: N/A bit 5: 100 msec*	N/A N/A N/A N/A N/A TF_10MS	N/A	pointer to next DV_TPT if linked list
DX_DIGMASK	IO_LINK IO_EOT IO_CONT	bit 0: d (set) bit 1: 1 bit 2: 2 bit 3: 3 bit 4: 4 bit 5: 5 bit 6: 6 bit 7: 7 bit 8: 8 bit 9: 9 bit 10: 0 bit 11: - bit 12: # bit 13: a bit 14: b bit 15: c	bit 0: TF_EDGE	TF_LEVEL*	N/A	pointer to next DV_TPT if linked list
DX_PMOFF	IO_LINK IO_EOT IO_CONT	number of pattern repetitions	minimum time silence off		max time silence off	pointer to next DV_TPT if linked list
DX_PMON	IO_LINK IO_EOT IO_CONT	bit 0: TF_EDGE*/ TF_LEVEL bit 1: N/A bit 2: N/A bit 3: N/A bit 4: N/A bit 5: 100 msec/ TF_10MS	maximum time silence on		max time silence on	pointer to next DV_TPT if linked list

Table 5. DV_TPT Field Settings Summary (Continued)

tp_termno	tp_type	tp_length	tp_flags: not set	tp_flags: set	tp_data	tp_nextp
DX_TONE	IO_LINK IO_EOT IO_CONT	Tone ID	bit 0: TF_EDGE bit 1: no clr bit 2: no clr* bit 3: clr hist	TF_LEVEL* TF_CRLREND* TF_CLRBEQ TF_USE*	DX_TONEON DX_TONEOFF	pointer to next DV_TPT if linked list
DX_DIGTYPE	IO_LINK IO_EOT IO_CONT	low byte: ASCII val. *hi byte: digit type	bit 0: TF_EDGE	TF_LEVEL	N/A	pointer to next DV_TPT if linked list

■ Example

See `dx_playiottdata()` and `dx_reciottdata()` for an example of how to use the DV_TPT structure.

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) */
```

DV_TPT — termination parameter table

```
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_CAP

```

* DX_CAP
* call progress analysis parameters
*/

typedef struct DX_CAP {
    unsigned short ca_nbrdna; /* # of rings before no answer. */
    unsigned short ca_stdely; /* Delay after dialing before analysis. */
    unsigned short ca_cnosig; /* Duration of no signal time out delay. */
    unsigned short ca_lcdly; /* Delay after dial before lc drop connect */
    unsigned short ca_lcdly1; /* Delay after lc drop con. Before msg. */
    unsigned short ca_hedge; /* Edge of answer to send connect message. */
    unsigned short ca_cnosil; /* Initial continuous noise timeout delay. */
    unsigned short ca_lo1tola; /* % acceptable pos. dev of short low sig. */
    unsigned short ca_lo1tolb; /* % acceptable neg. dev of short low sig. */
    unsigned short ca_lo2tola; /* % acceptable pos. dev of long low sig. */
    unsigned short ca_lo2tolb; /* % acceptable neg. dev of long low sig. */
    unsigned short ca_hiltola; /* % acceptable pos. dev of high signal. */
    unsigned short ca_hiltolb; /* % acceptable neg. dev of high signal. */
    unsigned short ca_lo1bmax; /* Maximum interval for shrt low for busy. */
    unsigned short ca_lo2bmax; /* Maximum interval for long low for busy. */
    unsigned short ca_hilbmax; /* Maximum interval for 1st high for busy */
    unsigned short ca_nsbuys; /* Num. of highs after nbrdna busy check. */
    unsigned short ca_logltch; /* Silence deglitch duration. */
    unsigned short ca_higlth; /* Non-silence deglitch duration. */
    unsigned short ca_lo1rmax; /* Max. short low dur. of double ring. */
    unsigned short ca_lo2rmin; /* Min. long low dur. of double ring. */
    unsigned short ca_intfltr; /* 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_nrbreg; /* Number of rings before analysis begins. */
    unsigned short ca_hilceil; /* Maximum 2nd high dur. for a retrain. */
    unsigned short ca_lolceil; /* Maximum 1st low dur. for a retrain. */
    unsigned short ca_lowerfrq; /* Lower allowable frequency in Hz. */
    unsigned short ca_upperfrq; /* Upper allowable frequency in Hz. */
    unsigned short ca_timefrq; /* Total duration of good signal required. */
    unsigned short ca_rejctfrq; /* Allowable % of bad signal. */
    unsigned short ca_maxansr; /* Maximum duration of answer. */
    unsigned short ca_ansrdgl; /* Silence deglitching value for answer. */
    unsigned short ca_mxtimefrq; /* max time for 1st freq to remain in bounds */
    unsigned short ca_lower2frq; /* lower bound for second frequency */
    unsigned short ca_upper2frq; /* upper bound for second frequency */
    unsigned short ca_time2frq; /* min time for 2nd freq to remain in bounds */
    unsigned short ca_mxtime2frq; /* max time for 2nd freq to remain in bounds */
    unsigned short ca_lower3frq; /* lower bound for third frequency */
    unsigned short ca_upper3frq; /* upper bound for third frequency */
    unsigned short ca_time3frq; /* min time for 3rd freq to remain in bounds */
    unsigned short ca_mxtime3frq; /* max time for 3rd freq to remain in bounds */
    unsigned short ca_dtn_pres; /* Length of a valid dial tone (def=1sec) */
    unsigned short ca_dtn_npres; /* Max time to wait for dial tone (def=3sec) */
    unsigned short ca_dtn_deboff; /* The dialtone off debouncer (def=100msec) */
    unsigned short ca_pamd_failtime; /* 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 (10 sec) */
} DX_CAP;

```

DX_CAP — call progress analysis parameters

■ **Description**

The DX_CAP data structure contains call progress analysis parameters.

The DX_CAP structure modifies parameters that control frequency detection, cadence detection, loop current, positive voice detection (PVD), and positive answering machine detection (PAMD). The DX_CAP structure is used by **dx_dial()**.

For more information about call progress analysis as well as how and when to use the DX_CAP structure, see the *Dialogic[®] Voice API Programming Guide*.

Note: 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.

■ **Field Descriptions (HMP Software)**

On HMP Software, the following fields of the DX_CAP data structure are supported:

Note: By setting a DX_CAP field to 0, the default value for that field will be used.

ca_cnosi

Continuous No Signal. The maximum time of silence (no signal) allowed immediately after cadence detection begins. If exceeded, a “no ringback” is returned.

Length: 2 Default: 4000 Units: 10 msec

ca_intflg

Intercept Mode Flag. Enables or disables SIT frequency detection, positive voice detection (PVD), and/or positive answering machine detection (PAMD), and selects the mode of operation for SIT frequency detection.

- DX_OPTDIS – Disable SIT frequency detection, PAMD, and PVD.
This setting provides call progress without SIT frequency detection.
- DX_OPTNOCON – Enable SIT frequency detection and return an “intercept” immediately after detecting a valid frequency.
This setting provides call progress with SIT frequency detection.
- DX_PVDENABLE – Enable PVD and fax tone detection.
This setting provides PVD call analysis only (no call progress).
- DX_PVDOPTNOCON – Enable PVD, DX_OPTNOCON, and fax tone detection.
This setting provides call progress with SIT frequency detection and PVD call analysis.
- DX_PAMDENABLE – Enable PAMD, PVD, and fax tone detection.
This setting provides PAMD and PVD call analysis only (no call progress).
- DX_PAMDOPTEN – Enable PAMD, PVD, DX_OPTNOCON, and fax tone detection.
This setting provides full call progress and call analysis.

Length: 1 Default: DX_OPTNOCON

ca_noanswer

No Answer. Length of time to wait after first ringback before deciding that the call is not answered.

Default: 3000 Units: 10 msec

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

■ **Field Descriptions (Springware Boards)**

On Springware boards, the following fields of the DX_CAP data structure are supported:

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

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)

DX_CAP — call progress analysis parameters

- 2 – Falling Edge (after the end of the salutation)

Length: 1 Default: 2

ca_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

ca_lo1tola

Low 1 Tolerance Above. Percent acceptable positive deviation of short low signal (Basic CPA only)

Length: 1 Default: 13 Units:%

ca_lo1tolb

Low 1 Tolerance Below. Percent acceptable negative deviation of short low signal (Basic CPA only)

Length: 1 Default: 13 Units:%

ca_lo2tola

Low 2 Tolerance Above. Percent acceptable positive deviation of long low signal (Basic CPA only)

Length: 1 Default: 13 Units:%

ca_lo2tolb

Low 2 Tolerance Below. Percent acceptable negative deviation of long low signal (Basic CPA only)

Length: 1 Default: 13 Units:%

ca_hi1tola

High 1 Tolerance Above. Percent acceptable positive deviation of high signal (Basic CPA only)

Length: 1 Default: 13 Units:%

ca_hi1tolb

High 1 Tolerance Below. Percent acceptable negative deviation of high signal (Basic CPA only)

Length: 1 Default: 13 Units:%

ca_lo1bmax

Low 1 Busy Maximum. Maximum interval for short low for busy (Basic CPA only)

Length: 2 Default: 90 Units: 10 msec

ca_lo2bmax

Low 2 Busy Maximum. Maximum interval for long low for busy (Basic CPA only)

Length: 2 Default: 90 Units: 10 msec

ca_hi1bmax

High 1 Busy Maximum. Maximum interval for first high for busy (Basic CPA only)

Length: 2 Default: 90 Units: 10 msec

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_hightch

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 allowmax or blowmax (Basic CPA only)

Length: 2 Default: 90 Units: 10 msec

ca_allowmax

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

DX_CAP — call progress analysis parameters

- ca_nbrbeg**
Number Before Beginning. Number of non-silence periods before analysis begins (Basic CPA only)
Length: 1 Default: 1 Units: rings
- ca_hi1ceil**
High 1 Ceiling. Maximum 2nd high duration for a retrain (Basic CPA only)
Length: 2 Default: 78 Units: 10 msec
- ca_lo1ceil**
Low 1 Ceiling. Maximum 1st low duration for a retrain (Basic CPA only)
Length: 2 Default: 58 Units: 10 msec
- ca_lowerfrq**
Lower Frequency. Lower bound for 1st tone in an SIT (CPA)
Length: 2 Default: 900 Units: Hz
- ca_upperfrq**
Upper Frequency. Upper bound for 1st tone in an SIT (CPA)
Length: 2 Default: 1000 Units: Hz
- ca_timefrq**
Time Frequency. Minimum time for 1st tone in an SIT to remain in bounds. The minimum amount of time required for the audio signal to remain within the frequency detection range specified by upperfrq and lowerfrq for it to be considered valid. (CPA)
Length: 1 Default: 5 Units: 10 msec
- ca_rejctfrq**
Not used
- ca_maxansr**
Maximum Answer. The maximum allowable length of ansrsize. When ansrsize exceeds maxansr, a “connect” is returned to the application. (CPA)
Length: 2 Default: 1000 Units: 10 msec
- ca_ansrdgl**
Answer Deglitcher. The maximum silence period allowed between words in a salutation. This parameter should be enabled only when you are interested in measuring the length of the salutation. (Basic CPA only)
• -1 – Disable this condition
Length: 2 Default: -1 Units: 10 msec
- ca_mxtimefrq**
Maximum Time Frequency. Maximum allowable time for 1st tone in an SIT to be present
Default: 0 Units: 10 msec
- ca_lower2frq**
Lower Bound for 2nd Frequency. Lower bound for 2nd tone in an SIT
Default: 0 Units: Hz
- ca_upper2frq**
Upper Bound for 2nd Frequency. Upper bound for 2nd tone in an SIT
Default: 0 Units: Hz

- ca_time2frq**
Time for 2nd Frequency. Minimum time for 2nd tone in an SIT to remain in bounds
Default: 0 Units: 10 msec
- ca_mvertime2frq**
Maximum Time for 2nd Frequency. Maximum allowable time for 2nd tone in an SIT to be present
Default: 0 Units: 10 msec
- ca_lower3frq**
Lower Bound for 3rd Frequency. Lower bound for 3rd tone in an SIT
Default: 0 Units: Hz
- ca_upper3frq**
Upper Bound for 3rd Frequency. Upper bound for 3rd tone in an SIT
Default: 0 Units: Hz
- ca_time3frq**
Time for 3rd Frequency. Minimum time for 3rd tone in an SIT to remain in bounds
Default: 0 Units: 10 msec
- ca_mvertime3frq**
Maximum Time for 3rd Frequency. Maximum allowable time for 3rd tone in an SIT to be present
Default: 0 Units: 10 msec
- ca_dtn_pres**
Dial Tone Present. Length of time that a dial tone must be continuously present (PerfectCall CPA only)
Default: 100 Units: 10 msec
- ca_dtn_npres**
Dial Tone Not Present. Maximum length of time to wait before declaring dial tone failure (PerfectCall CPA only)
Default: 300 Units: 10 msec
- ca_dtn_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
- ca_pamd_failtime**
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
- ca_pamd_minring**
Minimum PAMD Ring. Minimum allowable ring duration for positive answering machine detection (PerfectCall CPA only)
Default: 190 Units: 10 msec
- ca_pamd_spdval**
PAMD Speed Value. Quick or full evaluation for PAMD detection

DX_CAP — call progress analysis parameters

- PAMD_FULLL – 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_FULLL but is more accurate than PAMD_FULLL (although slightly slower) in detecting an answering machine. Use PAMD_ACCU when accuracy is more important than speed.

Default: PAMD_FULLL

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

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

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: 1000 Units: 10 msec

■ Example

For an example of DX_CAP, see the Example section for [dx_dial\(\)](#).

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_SETHOOKevent occurs. Use Dialogic® 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_SILOFF – non-silence detected
- DE_SILON – silence detected
- DE_STOPGETEVT – **dx_getevt()** stopped
- DE_TONEOFF – tone off event
- DE_TONEON – tone on event

On Springware boards, 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_STOPGETEVT – **dx_getevt()** stopped
- DE_STOPWTRING – **dx_wtring()** stopped
- 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:

DX_CST — call status transition (CST) information

- DE_DIGITS – ASCII digit (low byte) and the digit type (high byte)
- DE_SILOFF – time since previous silence started in 10 msec units
- DE_SILON – time since previous silence stopped in 10 msec units
- DE_STOPGETEVT – monitoring of channels for call status transition events has been stopped
- DE_TONEOFF – user-specified tone ID
- DE_TONEON – user-specified tone ID

On Springware boards, 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 since previous 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_STOPGETEVT – monitoring of channels for call status transition events has been stopped
- DE_STOPWTRING – waiting for a specified number of rings has been stopped
- 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_setevtmask()`.

■ Field Descriptions

The fields of the DX_EBLK data structure are described as follows:

ev_event

Contains the event type.

Use the following defines to identify the event type:

- DE_DIGITS – digit received
- DE_SILOFF – non-silence detected
- DE_SILON – silence detected
- DE_TONEOFF – tone off event
- DE_TONEON – tone on event

On Springware boards, 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_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

DX_EBLK — call status transition event block

- DE_TONEOFF – user-specified tone ID for the tone-off event
- DE_TONEON – user-specified tone ID for the tone-on event

On Springware boards, 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_setevtmsk\(\)](#).

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 WAVE data offset I/O. Set the io_type field to an OR combination of the following defines.

Specify the data transfer type as follows:

- IO_DEV – file data
- IO_MEM – memory data
- IO_STREAM – data for streaming to board
- IO_UIO – nonstandard storage media data using the [dx_setuio\(\)](#) function; must be ORed with IO_DEV

On Springware boards, specifies 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:

DX_IOTT — input/output transfer table

- **IO_CONT** – the next **DX_IOTT** structure is contiguous (default)
- **IO_LINK** – the next **DX_IOTT** structure is part of a linked list
- **IO_EOT** – this is the last **DX_IOTT** structure in the chain

If no value is specified, **IO_CONT** is assumed.

Other Types:

- **IO_USEOFFSET** – enables use of the **io_offset** and **io_length** fields for WAVE data

To enable offset I/O for WAVE data, set the **DX_IOTT io_type** field to **IO_USEOFFSET** ORed with the **IO_DEV** define (to indicate file data rather than memory buffer).

Note: Wave file formats cannot be recorded to memory buffers or played from memory buffers.

io_fhandle

In Linux, specifies a unique file descriptor if **IO_DEV** is set in **io_type**. If **IO_DEV** is not set in **io_type**, **io_fhandle** should be set to 0.

In Windows®, specifies a unique file descriptor provided by the **dx_fileopen()** function if **IO_DEV** is set in **io_type**. If **IO_DEV** is not set in **io_type**, **io_fhandle** should be set to 0.

io_bufp

Specifies a base memory address if **IO_MEM** is set in **io_type**.

io_offset

Specifies one of the following:

- if **IO_DEV** is specified in **io_type**, an offset from the beginning of a file
- for WAVE file offset I/O (**IO_DEV** is ORed with **IO_USEOFFSET** in **io_type**), a file offset value that is calculated from the beginning of the WAVE audio data rather than the beginning of the file (that is, the first 80 bytes that make up the file header are not counted).
- if **IO_MEM** is specified in **io_type**, an offset from the base buffer address specified in **io_bufp**

io_length

Specifies the number of bytes allocated for recording or the byte length of the playback file. Specify -1 to play until end of data. During **dx_play()**, a value of -1 causes playback to continue until an EOF is received or one of the terminating conditions is satisfied. During **dx_rec()**, a value of -1 in **io_length** causes recording to continue until one of the terminating conditions is satisfied.

io_nextp

Points to the next **DX_IOTT** structure in the linked list if **IO_LINK** is set in **io_type**.

io_prevp

Points to the previous **DX_IOTT** structure. This field is automatically filled in when **dx_rec()** or **dx_play()** is called. The **io_prevp** field of the first **DX_IOTT** structure is set to **NULL**.

■ **Example**

The following example uses different sources for playback, an array or linked list of **DX_IOTT** structures.

```
#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 *dxxlib.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

```
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 *Dialogic® 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.

On Linux, at the start of play, adjustments will be made according to adjustment condition digits contained in the digit buffer. If `SV_LEVEL` is not specified, the digit adjustment condition is edge-sensitive, and will wait for a new occurrence of the digit before play adjusting.

On Windows[®], at the start of play, existing digits in the digit buffer will be checked to see if they are level-sensitive play adjustment digits. If the first digit in the buffer is a level-sensitive play adjustment digit, it will cause a play adjustment and be removed from the buffer. Subsequent digits in the buffer will be treated the same way until the first occurrence of any digit that is not an `SV_LEVEL` play adjustment digit. If `SV_LEVEL` is not specified, the digit adjustment condition is edge-sensitive. Existing edge-sensitive play adjustment digits in the digit buffer will not cause a play adjustment; but after the playback starts, edge-sensitive digits will cause a play adjustment.

- **SV_BEGINPLAY – Automatic:** Sets the play adjustment to occur automatically at the beginning of the next playback. This sets a 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)

DX_SVCB — speed and volume adjustment condition block

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 Dialogic® 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:

```
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:

```
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.

```
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\(\)](#).

speed and volume adjustment condition block — DX_SVCB

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 *Dialogic® 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.

Note: Wave file formats cannot be recorded to memory buffers or played from memory buffers.

■ **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.

■ 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_WAV – Microsoft WAVE file format

wDataFormat

Specifies the data format.

Use one of the following data formats:

- DATA_FORMAT_DIALOGIC_ADPCM – 4-bit OKI ADPCM (Dialogic registered format)
- DATA_FORMAT_MULAW or DATA_FORMAT_G711_MULAW – 8-bit mu-law G.711 PCM
- DATA_FORMAT_ALAW or DATA_FORMAT_G711_ALAW – 8-bit A-law G.711 PCM
- DATA_FORMAT_PCM – 8-bit or 16-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)

DX_XPB — input/output transfer parameter block

- DATA_FORMAT_GSM610_TIPHON – GSM 6.10 VOX full-rate coder (TIPHON format)

On Springware boards, the following are valid 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 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

The following examples explain how to fill the DX_XPB structure for various voice coders.

Table 6. G.711 Voice Coder Support Fields

DX_XPB Field	DX_XPB Field Value	Note
wFileFormat	FILE_FORMAT_WAV or FILE_FORMAT_VOX	
wDataFormat	DATA_FORMAT_G711_ALAW or DATA_FORMAT_ALAW DATA_FORMAT_G711_MULAW or DATA_FORMAT_MULAW	
nSamplesPerSec	DRT_6KHZ or DRT_8KHZ	
wBitsPerSample	8	48 or 64 kbps

Table 7. Linear PCM Voice Coder Support Fields

DX_XPB Field	DX_XPB Field Value	Note
wFileFormat	FILE_FORMAT_WAV or FILE_FORMAT_VOX	
wDataFormat	DATA_FORMAT_PCM	

Table 7. Linear PCM Voice Coder Support Fields (Continued)

nSamplesPerSec	DRT_8KHZ DRT_11KHZ	
wBitsPerSample	8 or 16	88, 128 kbps

Table 8. OKI ADPCM Voice Coder Support Fields

DX_XPB Field	DX_XPB Field Value	Note
wFileFormat	FILE_FORMAT_WAV or FILE_FORMAT_VOX	
wDataFormat	DATA_FORMAT_DIALOGIC_ADPCM	
nSamplesPerSec	DRT_6KHZ or DRT_8KHZ	
wBitsPerSample	4	24 or 32 kbps

Table 9. G.726 Voice Coder Support Fields

DX_XPB Field	DX_XPB Field Value	Note
wFileFormat	FILE_FORMAT_WAV or FILE_FORMAT_VOX	
wDataFormat	DATA_FORMAT_G726	
nSamplesPerSec	DRT_8KHZ	
wBitsPerSample	2, 4	16, 32 kbps

Table 10. GSM Voice Coder Support Fields

DX_XPB Field	DX_XPB Field Value	Note
wFileFormat	FILE_FORMAT_WAV FILE_FORMAT_VOX	WAVE format supported only with DATA_FORMAT_GSM610_MICROSOFT
wDataFormat	DATA_FORMAT_GSM610_MICROSOFT DATA_FORMAT_GSM610_TIPHON	
nSamplesPerSec	DRT_8KHZ	
wBitsPerSample	0	13 kbps

■ **Examples (Springware boards)**

Table 11 and Table 12 provide examples of how to fill the DX_XPB structure for various voice coders on Springware boards.

Table 11. G.726 Voice Coder Support Fields (Springware boards)

DX_XPB Field	DX_XPB Field Value	Note
wFileFormat	FILE_FORMAT_VOX	
wDataFormat	DATA_FORMAT_G726	
nSamplesPerSec	DRT_8KHZ	
wBitsPerSample	4	32 kbps

Table 12. GSM Voice Coder Support Fields (Springware boards)

DX_XPB Field	DX_XPB Field Value	Note
wFileFormat	FILE_FORMAT_WAV	
wDataFormat	DATA_FORMAT_GSM610_MICROSOFT DATA_FORMAT_GSM610_TIPHON	
nSamplesPerSec	DRT_8KHZ	
wBitsPerSample	0	This field can be any numeric value; it is ignored. However, the recommended setting is 0. 13 kbps

FEATURE_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 *dxlib.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(HMP Software)

The fields of the FEATURE_TABLE data structure are described as follows:

ft_play

Contains a bitmask of the play features supported on the specified device.

- FT_ADPCM – supports ADPCM encoding
- FT_ALAW – supports A-law encoding
- FT_DRT6KHZ – supports 6 kHz sampling rate
- FT_DRT8KHZ – supports 8 kHz sampling rate
- FT_DRT11KHZ – supports 11 kHz sampling rate
- FT_ITU_G_726 – supports ITU-T G.726 encoding
- FT_LINEAR – supports linear PCM encoding
- FT_PCM – supports PCM encoding
- FT_RAW64KBIT – supports raw 64 Kbps
- FT_RESRVD1 – reserved
- FT_RESRVD2 – reserved
- FT_ULAW – supports mu-law encoding

ft_record

Contains a bitmask of the record features supported on the specified device.

- FT_ADPCM – supports ADPCM encoding
- FT_ALAW – supports A-law encoding
- FT_DRT6KHZ – supports 6 kHz sampling rate
- FT_DRT8KHZ – supports 8 kHz sampling rate
- FT_DRT11KHZ – supports 11 kHz sampling rate

FEATURE_TABLE — feature information

- FT_ITU_G_726 – supports ITU-T G.726 encoding
- FT_LINEAR – supports linear PCM encoding
- FT_PCM – supports PCM encoding
- FT_RAW64KBIT – supports raw 64 Kbps
- FT_RESRVD1 – reserved
- FT_RESRVD2 – reserved
- FT_ULAW – supports mu-law encoding

ft_tone

Contains a bitmask of the tone features supported on the specified device.

- FT_GTDENABLED – supports global tone detection (GTD)
- FT_GTGENABLED – supports global tone generation (GTG)
- FT_CADENCE_TONE – supports cadenced tone generation

ft_e2p_brd_cfg

Contains a bitmask of the board configuration features supported on the specified device.

- FT_CONFERENCE – supports conferencing
- FT_CSP – supports continuous speech processing

ft_fax

Contains a bitmask of the board type and fax features supported on the specified device.

- FT_FAX – specifies that the device has a fax daughterboard
- FT_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_T38UDP – supports T.38 fax

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

Not used on HMP.

ft_misc

Not used on HMP.

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_RXASCII – supports ASCII data fax reception
- FT_RXFILEMR – supports MR encoded file format
- FT_RXFILEMMR – supports MMR encoded file format
- FT_RXLINEMR – supports MR encoded file format over the phone line
- FT_RXLINEMMR – supports MMR encoded file format over the phone line
- FT_RXECM – capable of fax line reception with error correction mode

ft_play_ext

Not used on Dialogic® HMP Software.

ft_record_ext

Not used on Dialogic® HMP Software.

ft_device

Reserved for future use.

ft_rfu

Reserved for future use.

■ Field Descriptions (Springware Boards)

The fields of the FEATURE_TABLE data structure are described as follows for Springware boards:

ft_play

Contains a bitmask of the play features supported on the specified device.

- 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_FSK_OH – •supports on-hook ADSI 2-way frequency shift encoding (FSK)FT_G729A – •supports G.729a encodingFT_ITU_G_726 – supports ITU-T G.726 encoding
- FT_LINEAR – supports linear PCM encoding
- FT_MSGSM – •supports Microsoft GSM encodingFT_PCM – supports PCM encoding
- FT_RAW64KBIT – supports raw 64 Kbps
- FT_RESRVD1 – reserved
- FT_RESRVD2 – reserved
- FT_ULAW – supports mu-law encoding

ft_record

Contains a bitmask of the record features supported on the specified device.

- FT_ADPCM – supports ADPCM encoding
- FT_ALAW – supports A-law encoding
- FT_DRT6KHZ – supports 6 kHz sampling rate
- FT_DRT8KHZ – supports 8 kHz sampling rate
- FT_DRT11KHZ – supports 11 kHz sampling rate
- FT_FFT – •supports Fast Fourier Transform (FFT) algorithm on recordsFT_FSK_OH – supports on-hook ADSI 2-way frequency shift encodingFT_G729A – •supports G.729a encodingFT_ITU_G_726 – supports ITU-T G.726 encoding

FEATURE_TABLE — feature information

- FT_LINEAR – supports linear PCM encoding
- FT_MSGSM – supports Microsoft GSM encoding
- FT_PCM – supports PCM encoding
- FT_RAW64KBIT – supports raw 64 Kbps
- FT_RESRVD1 – reserved
- FT_RESRVD2 – reserved
- FT_ULAW – supports mu-law encoding

ft_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

resourceft_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).

ft_front_end

Contains a bitmask of the front-end features supported on the specified device.

- 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_CSPEXTRATSLOT – 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_RECFCFLOWCONTROL – 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_RXASCII – supports ASCII data fax reception
- FT_RXFILEMR – supports MR encoded file format
- FT_RXFILEMMR – supports MMR encoded file format
- FT_RXLINEMR – supports MR encoded file format over the phone line
- FT_RXLINEMMR – supports MMR encoded file format over the phone line
- FT_RXECM – capable of fax line reception with error correction mode

ft_play_ext

Not used on Springware boards.ft_record_ext

Not used on Springware boards.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_tsarray;
} 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), `dx_` (voice) and `fx_` (fax).

The TDM bus includes the CT Bus and SCbus. The CT Bus has 4096 bi-directional time slots, while the SCbus has 1024 bi-directional time slots. On Dialogic[®] Host Media Processing (HMP) Software, no physical TDM bus exists but its functionality is implemented in the software; the number of time slots available is 4096.

This structure is defined in *dxxlib.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. In Linux, set to 2 for two-channel transaction recording (using `dx_recmf()` or `dx_recmf()` functions).

`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. Valid values are from 0 up to 4095.

■ 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\(\)](#).

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 *dxxlib.h*.

■ Field Descriptions

The fields of the TN_GENCAD data structure are described as follows:

cycles

The cycles field specifies the number of times the cycle will be played.

Valid values are 1 to 40 cycles.

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_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 *dxlib.h*. For information on call progress analysis and default tone definitions, see the *Dialogic® Voice API Programming Guide*.

The TONE_DATA structure contains a nested array of TONE_SEG substructures. A maximum of six TONE_SEG substructures can be specified.

Note: Be sure to set all unused fields in the structure to 0 before using this structure in a function call. This action prevents possible corruption of data in the allocated memory space.

■ Field Descriptions

The fields of the TONE_DATA structure are described as follows:

TONE_SEG.structver

Reserved for future use, to specify the version of the structure. Set to 0.

TONE_SEG.tn_dflag

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

TONE_SEG.tn1_min

Specifies the minimum frequency in Hz for tone 1.

TONE_SEG.tn1_max

Specifies the maximum frequency in Hz for tone 1.

TONE_SEG.tn2_min

Specifies the minimum frequency in Hz for tone 2.

- TONE_SEG.tn2_max
Specifies the maximum frequency in Hz for tone 2.
- TONE_SEG.tn_twinmin
Specifies the minimum frequency in Hz of the single tone proxy for the dual tone.
- TONE_SEG.tn_twinmax
Specifies the maximum frequency in Hz of the single tone proxy for the dual tone.
- TONE_SEG.tnon_min
Specifies the debounce minimum ON time in 10 msec units.
- TONE_SEG.tnon_max
Specifies the debounce maximum ON time in 10 msec units.
- TONE_SEG.tnoff_min
Specifies the debounce minimum OFF time in 10 msec units.
- TONE_SEG.tnoff_max
Specifies the debounce maximum OFF time in 10 msec units.
- TONE_DATA.structver
Reserved for future use, to specify the version of the structure. Set to 0.
- 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 Dialogic® Voice API 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 *Dialogic® Standard Runtime Library API Library Reference*.

The following error codes can be returned by the `ATDV_ERRMSGP()` function:

- 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

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
EDX_NOTENOUGHBRDMEM	Error when downloading a cached prompt from multiple sources: total length of data to be downloaded exceeds the available on-board memory
EDX_NOTIMP	Function is not implemented
EDX_SH_BADCMD	Command is not supported in current bus configuration

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 time slot channel type (voice, analog, etc.)
EDX_SH_CMDBLOCK	Blocking command is in progress
EDX_SH_LCLDSCNCT	Channel is already disconnected from TDM bus
EDX_SH_LCLTSCNCT	Channel is already connected to TDM bus
EDX_SH_LIBBSY	Switch Handler library is busy
EDX_SH_LIBNOTINIT	Switch Handler library is uninitialized
EDX_SH_MISSING	Switch Handler is not present
EDX_SH_NOCLK	Switch Handler clock fallback failed
EDX_SPDVOL	Must specify either SV_SPEEDTBL or SV_VOLUMETBL
EDX_SVADJBLKS	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. In Windows [®] , use <code>dx_fileerrno()</code> to obtain error value. In Linux, check the global variable <code>errno</code> for more information.
EDX_TIMEOUT	I/O function timed out
EDX_TONEID	Invalid tone template ID
EDX_TNMSGSTATUS	Invalid message status setting

EDX_UNSUPPORTED
Function is not supported

EDX_WTRINGSTOP
Wait-for-Rings stopped by user

EDX_XBPARAM
Bad XPB structure

Supplementary Reference Information

6

This chapter provides reference information on the following topics:

- DTMF and MF Tone Specifications 533
- DTMF and MF Detection Errors 534

6.1 DTMF and MF Tone Specifications

Table 13 provides information on DTMF specifications. Table 14 provides information on MF tone specifications.

Table 13. DTMF Tone Specifications

Code	Tone Pair Frequencies (Hz)	Default Length (msec)
1	697, 1209	100
2	697, 1336	100
3	697, 1477	100
4	770, 1209	100
5	770, 1336	100
6	770, 1477	100
7	852, 1209	100
8	852, 1336	100
9	852, 1477	100
0	941, 1336	100
*	941, 1209	100
#	941, 1477	100
a	697, 1633	100
b	770, 1633	100
c	852, 1633	100
d	941, 1633	100

Table 14. MF Tone Specifications (CCITT R1 Tone Plan)

Code	Tone Pair Frequencies (Hz)	Default Length (msec)	Name
1	700, 900	60	1
2	700, 1100	60	2
3	900, 1100	60	3
4	700, 1300	60	4
5	900, 1300	60	5
6	1100, 1300	60	6
7	700, 1500	60	7
8	900, 1500	60	8
9	1100, 1500	60	9
0	1300, 1500	60	0
*	1100, 1700	60	KP
#	1500, 1700	60	ST
a	900, 1700	60	ST1
b	1300, 1700	60	ST2
c	700, 1700	60	ST3

* The standard length of a KP tone is 100 msec

6.2 DTMF and MF Detection Errors

Some MF digits use approximately the same frequencies as DTMF digits (see Table 13 and Table 14). 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 [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 15 and Table 16 show the digits that are detected when each type of detection is enabled. Table 15 shows which digits are detected when MF digits are sent. Table 16 shows which digits are detected when DTMF digits are sent.

Supplementary Reference Information

Table 15. Detecting MF Digits

MF Digit Sent	String Received		
	Only MF Detection Enabled	Only DTMF Detection Enabled	MF and DTMF Detection Enabled
1	1		1
2	2		2
3	3		3
4	4	2 [†]	4,2 [†]
5	5		5
6	6		6
7	7	3 [†]	7,3 [†]
8	8		8
9	9		9
0	0		0
*	*		*
#	#		#
a	a		a
b	b		b
c	c		c

[†] = detection error

Table 16. Detecting DTMF Digits

DTMF Digit Sent	String Received		
	Only DTMF Detection Enabled	Only MF Detection Enabled	DTMF and MF Detection Enabled
1	1		1
2	2	4 [†]	4,2 [†]
3	3	7 [†]	7,3 [†]
4	4		4
5	5	4 [†]	4,5 [†]
6	6	7 [†]	7,6 [†]
7	7		7
8	8	5 [†]	5,8 [†]
9	9	8 [†]	8,9 [†]
0	0	5 [†]	5,0 [†]
*	*		*

[†] = detection error

Table 16. Detecting DTMF Digits (Continued)

DTMF Digit Sent	String Received		
	Only DTMF Detection Enabled	Only MF Detection Enabled	DTMF and MF Detection Enabled
#	#	8 [†]	8,# [†]
a	a	c [†]	c,a [†]
b	b	c [†]	c,b [†]
c	c	a [†]	a,c [†]
d	d	a [†]	a,d [†]
† = detection error			

Glossary

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

caller ID: calling party identification information.

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

channel: 1. When used in reference to a Dialogic® 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 Dialogic® 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. On Windows® only, a utility with a graphical user interface (GUI) that enables you to add new boards to your system, start and stop system service, and work with board configuration data.

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

deglitching: See [debouncing](#).

device: A computer peripheral or component controlled through a software device driver. A Dialogic® 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: A Dialogic® voice data path that processes one incoming or outgoing call at a time (equivalent to the terminal equipment terminating a phone line).

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

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

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

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

DM3: Refers to Dialogic® mediastream processing architecture, which is open, layered, and flexible, encompassing hardware as well as software components. A whole set of products from Dialogic are built on the Dialogic® 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.

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

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

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

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

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

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

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

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

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

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

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

GSM (Global System for Mobile Communications): A digital cellular phone technology based on time division multiple access (TDMA) used in Europe, Japan, Australia and elsewhere around the world.

I/O: Input-Output

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

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

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

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

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

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

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

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

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

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

RFU: reserved for future use

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

route: Assign a resource to a time slot.

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

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

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

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

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

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

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

SRL: See **Standard Runtime Library**.

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

standard runtime library (SRL): A Dialogic® software resource containing event management and standard attribute functions and data structures used by Dialogic® 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 Dialogic that is required to develop applications.

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

TDMA (Time Division Multiple Access): A method of digital wireless communication using time division multiplexing.

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

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

termination event: An event that is generated when an asynchronous function terminates. See also [asynchronous function](#).

time division multiplexing (TDM): See [TDM \(Time Division Multiplexing\)](#).

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

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

underrun: data is not being delivered to the board quickly enough which can result in loss of data and gaps in the audio

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

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

Index

A

- ACLIP
 - message types 252
- adjusting speed and volume
 - explicitly 125
 - using conditions 420
 - using digits 420
- adjustment conditions
 - digits 421
 - maximum number 421
 - setting 420
- ADPCM 284, 319
- ADSI 18
 - functions 18
 - two-way 447
- ADSI_XFERSTRUC data structure 474
- ag_getctinfo() 28
- ag_getxmitslot() 30
- ag_listen() 34
- ag_unlisten() 38
- AGC 319
- A-law 284, 514
- alowmax 494
- analog devices
 - connecting to time slot 34
 - disconnecting from TDM bus 38
 - get time slot number 30
 - getting information about 28
- Analog Display Services Interface (ADSI) 18
- ansrdgl 495
- answering machine detection 40
- array 503
- asynchronous operation
 - dialing 178
 - digit collection 216
 - playing 285
 - playing tone 298
 - recording 320, 336
 - setting hook state 408
 - stopping I/O functions 438
- ATDX_functions 25
- ATDX_ANSRSIZ() 40
- ATDX_BDNAMEP() 42
- ATDX_BDTYPE() 44
- ATDX_BUFDIGS() 46, 122
- ATDX_CHNAMES() 48
- ATDX_CHNUM() 50
- ATDX_CONNTYPE() 52
- ATDX_CPERM() 55
- ATDX_CPTERM() 55, 58
- ATDX_CRTNID() 61
- ATDX_DEVTYPE() 67
- ATDX_DTNFAIL() 69
- ATDX_FRQDUR() 72
- ATDX_FRQDUR2() 74
- ATDX_FRQDUR3() 76
- ATDX_FRQHZ() 78
- ATDX_FRQHZ2() 80
- ATDX_FRQHZ3() 82
- ATDX_FRQOUT() 84
- ATDX_FWVER() 86
- ATDX_HOOKST() 89, 409
- ATDX_LINEST() 91
- ATDX_LONGLOW() 93
- ATDX_PHYADDR() 95
- ATDX_SHORTLOW() 97
- ATDX_SIZEHI() 99
- ATDX_STATE() 101
- ATDX_TERMMSK() 103, 107
- ATDX_TONEID() 107
- ATDX_TRCOUNT() 110
- audio pulse digits 397
- automatic gain control 319

B

- base memory address 503
- beep tone, pre-record 430
- bits per sample 515
- blowmax 494
- board
 - device 67, 278
 - device name 42
 - parameters 413, 415, 416
 - setting 42
 - physical address 95

- board device
 - handle 48
- breaking
 - connection to a time slot 452, 454
- buffer
 - firmware digit 161
- buffer size
 - bulk queue 363
- buffer, ADSI data 474
- busy channel 351
 - forcing to idle state 438

C

- ca_dtn_deboff 496
- ca_dtn_npres 496
- ca_dtn_pres 496
- ca_lowerfrq 84
- ca_maxintering 497
- ca_noanswer 491, 497
- ca_pamd_failtime 492, 496
- ca_pamd_minring 496
- ca_pamd_qtemp 497
- ca_pamd_spdval 492, 496
- ca_upperfrq 84
- cached prompts
 - playing 294
- cadence 40
 - repetition for user-defined tones 133
- cadenced tone
 - playing 302

- call progress analysis 40, 58, 97, 99
 - activating 260
 - answering machine detection 40
 - cadence 40
 - data structure 491
 - enabling 178
 - enhanced
 - activating 260
 - errors 55
 - frequency detection
 - SIT tones(tone 1) 72, 78
 - SIT tones(tone 2) 74, 80
 - SIT tones(tone 3) 76, 82
 - functions 23
 - dx_chgdur() 144
 - dx_chgfreq() 148
 - dx_chgrepcnt() 152
 - parameter structure 159
 - results
 - busy 58
 - called line answered by 58
 - connect 58
 - error 59
 - initial non-silence 99
 - longer silence 93
 - no answer 58
 - no ringback 58
 - operator intercept 58
 - shorter silence 97
 - stopped 59
 - timeout 58
 - stopping 181, 439
 - termination 58
 - tone definitions 144, 148, 152
 - using dx_dial() 177
- call progress analysis functions
 - dx_initcallp() 260
- call progress signals, standard 303
- call progress tone 168, 172, 316
- call status transition
 - DX_CST data structure 498
 - event block structure 500
 - event handling 402
 - synchronously monitoring events 224
- call status transition event functions 19
 - dx_getevt() 224
 - dx_setevtmask() 400
- call status transition structure 498
- caller ID
 - common message types 251

- caller ID functions 24
 - dx_gtcallid() 244
 - dx_gtextcallid() 250
 - dx_wtcallid() 457
- channel
 - bulk queue buffer sizing function 363
 - current state 101
 - device 67, 278
 - digit buffer 215, 221
 - monitoring activity 91
 - names 48
 - number 50
 - number of processes 225
 - parameters 415, 416
 - status
 - dial 101
 - DTMF signal 91
 - get digit 101
 - idle 101
 - no loop current 91
 - no ringback 91
 - onhook 91
 - play 101
 - playing tone 102
 - record 101
 - ringback present 91
 - silence 91
 - stopped 101
- channel device information structure 475
- channel parameters 415, 417
- CLASS
 - message types 252
- clearing structures 159, 165
- CLIP
 - message types 252
- close() 155
- close() function, Windows 155
- closing devices 155
- cnosig 491, 492
- cnosil 493
- coders 514
- common message types 251
- CON_CAD 52
- CON_LPC 52
- CON_PAMD 52
- CON_PVD 52
- configuration functions 15
 - dx_clrdigbuf() 161
 - dx_getparm() 231
 - dx_GetRscStatus() 234
 - dx_setchxfercnt() 363
 - dx_setdigtyp() 397
 - dx_sethook() 408
 - dx_setparm() 412
 - dx_settonelen() 430
 - dx_TSFStatus() 442
 - dx_wtring() 460
- connect
 - event 40
 - type 52
- convenience functions
 - dx_playf() 291
 - dx_playvox() 307
 - dx_recf() 326
 - dx_recvox() 345
 - dx_recwav() 348
 - I/O 17
 - R2/MF 22
 - speed and volume 22
 - TDM Routing 20
- CR_BUSY 58
- CR_CEPT 58, 78, 80, 82
- CR_CNCT 52, 58
- CR_ERROR 55
- CR_FAXTONE 58
- CR_LGTUERR 55
- CR_MEMERR 55
- CR_MXFRQERR 55
- CR_NOANS 58
- CR_NODIALTONE 58
- CR_NORB 58
- CR_OVRLPERR 56
- CR_STOPD 59
- CR_TMOUTOFF 56
- CR_TMOUTON 56
- CR_UNEXPTN 56
- CR_UPFRQERR 56
- CS_CALL 101
- CS_DIAL 101
- CS_GTDIG 101
- CS_HOOK 101
- CS_IDLE 101
- CS_PLAY 101
- CS_RECD 101
- CS_STOPD 101
- CS_TONE 102

cst_data 498
cst_event 498
CT_DEVINFO data structure 28, 210, 475
current parameter settings 231
cycles 525

D

D_APD 397
D_DPD 397
D_DPDZ 397
D_DTMF 397
D_LPD 397
D_MF 397
data formats 514
data structure
 user digit buffer 478
data structures
 cadenced tone generation template 525
 call progress analysis parameters 491
 call status transition 498
 clearing 24
 event block 500
 feature information 518
 I/O
 user-definable 513
 I/O transfer table 502
 input/output transfer parameter block 514
 speed and volume adjustment conditions 507
 speed modification table 511
 TDM bus time slot information 523
 termination parameter table 481
 tone generation template 524
DE_DIGITS event 470, 471, 498, 500
DE_LCOFF event 471, 498, 500
DE_LCON event 471, 498, 500
DE_LCREV event 471, 498, 500
DE_RINGS event 471, 498, 500
DE_RNGOFF event 471, 498
DE_SILOFF event 470, 471, 498, 500
DE_SILON event 470, 471, 498, 500
DE_STOPGETEVT event 470, 471
DE_STOPRINGS event 471
DE_STOPWTRING event 471
DE_TONEOFF event 470, 471, 498, 500
DE_TONEON event 470, 471, 498, 500
DE_WINK event 498, 500
device
 opening 278
device handle 14, 44, 278
 freeing 155
device information structure 475
device management functions 14
 dx_close() 155
 dx_open() 278
device names
 displaying 48
device type 67
devices
 closing 155
 multiple processes 155
 returning features 518
 type 44
DG_DTMF 478
DG_END 478
DG_LPD 478
DG_MAXDIGS 216, 478
DG_MF 478
dg_type 478
DG_USER1 478, 479
dg_value 478
DI_D41BD 44
DI_D41CH 44
dial pulse digit (DPD) 397
dial tone
 failure 69
dialing
 ASCIIZ string 177, 185
 asynchronous 178, 186
 DTMF 179, 180
 enabling Call Progress Analysis 186
 enabling call progress analysis 178
 flash 179, 180
 flash character 180
 MF 179, 180
 pause 179, 180
 pause character 180
 pulse 179, 180
 specifying dial string 177, 179, 185
 stopping 181
 synchronous 178, 186
 synchronous termination 178
 termination events
 TDX_CALLP 178, 468
 TDX_DIAL 178, 468
 with Call Progress Analysis 186
 with call progress analysis 178
digit buffer 215, 217, 221
 flushing 161
digit buffer, extended 480

- digit buffer, user 478
- digit collection 215, 221
 - asynchronous 216
 - DTMF digits 216, 221
 - MF digits 216, 221
 - synchronous 216
 - termination 216
 - user-defined digits 216, 221
- digit detection 215
 - audio pulse 397
 - dial pulse 397
 - disabling 188
 - DPD, zero-train 397
 - DTMF 397
 - DTMF vs. MF tones 398
 - errors 534
 - mask 397
 - MF 397
 - multiple types 398
 - setting digit types 397
- digits
 - adjustment conditions 421
 - collecting 46
 - defines for user-defined tones 117
 - detecting 46
 - speed and volume 122
- disabling detection
 - user-defined tones 188
- disconnecting
 - voice receive channel 452, 454
- DM_DIGITS 401
- DM_DIGOFF 401
- DM_LCON 401
- DM_LCREV 401
- DM_RINGS 401, 460
- DM_RNGOFF 401
- DM_SILOF 401
- DM_SILON 401
- DM_UNDERRUN 401
- DPD
 - support 398
- DSP fax 234, 519, 521
- DT_DXBD 67
- DT_DXCH 67
- DTMF 535
 - detection errors 534
 - tone specifications 533
- DTMF digits 397
 - collection 216, 221
 - overlap with MF digits 217
- DV_DIGIT data structure 215, 478
 - specifying 216
- DV_DIGITEX 221
- DV_DIGITEX data structure 480
- DV_TPT data structure 481
 - clearing 165
 - contiguous 165
 - last entry in 165
 - linked 165
- dx_addspddig() 112
- dx_addtone() 116
- dx_addvoldig() 121
- dx_adjsv() 125
- dx_blddt() 129
- dx_blddtcad() 132
- dx_bldst() 135
- dx_bldstcad() 138
- dx_bldtngen() 141
- DX_CAP data structure 491
 - clearing 159
- dx_chgdur() 144
- dx_chgfreq() 148
- dx_chgrepcnt() 152
- dx_close() 155
- dx_CloseStream() 157
- dx_clrcap() 159
- dx_clrdigbuf() 46, 161, 216
- dx_clrsvcond() 163, 420
- dx_clrtpt() 165
- dx_createtone() 167
- DX_CST data structure 498
 - hook state terminations 408
- dx_deltone() 174
- dx_dial() 40, 103, 104, 159, 181, 438
- dx_dialtpt() 185
- dx_distone() 116, 188
- DX_EBLK data structure 224, 500
- dx_enbtone() 116, 191
- dx_fileclose() 194
- dx_fileerrno() 196
- dx_fileopen() 199
- dx_fileread() 201
- dx_fileseek() 204
- dx_filewrite() 207
- dx_getctinfo() 210
- dx_getcursv() 212
- dx_getdig() 46, 162, 215, 478

dx_getdigEx() 221, 480
 dx_getevt() 224, 402, 460, 500
 dx_getfeaturelist() 227
 FEATURE_TABLE data structure 518
 dx_getparm() 231, 284, 319, 413
 dx_GetRscStatus() 234
 dx_GetStreamInfo() 236
 dx_getsvmt() 238
 dx_getxmitslot() 241
 dx_gtcallid() 244
 dx_gtextcallid() 250
 dx_initcallp() 260
 DX_IOTT data structure 502
 dx_listen() 263
 dx_listenEx() 266
 dx_mreciottdata() 270
 DX_OFFHOOK event 89, 461, 471, 498, 500
 DX_ONHOOK event 89, 461, 471, 498, 500
 dx_open() 278
 dx_OpenStreamBuffer() 281
 dx_play() 162, 283, 291, 503
 dx_playf() 291
 dx_playiottdata() 294
 dx_playtone() 298
 dx_playtoneEx() 302
 dx_playvox() 307
 dx_playwav() 310
 dx_PutStreamData() 313
 dx_query() 315
 dx_querytone() 171
 dx_rec() 162, 318, 503
 dx_recf() 326
 dx_reciottdata() 330
 dx_recm() 335
 dx_recmf() 341
 dx_recvox() 345
 dx_recwav() 348
 dx_resetch() 351
 dx_ResetStreamBuffer() 354
 dx_RxIottData() 356
 dx_sendevt() 360
 dx_setchxfercnt() 363
 dx_setdevuio() 365
 dx_setdigbuf() 395
 dx_setdigtyp() 215
 dx_setevtmsk() 224, 400, 460

dx_setgtdamp() 406
 dx_sethook() 104, 408, 460
 dx_setparm() 284, 319, 412
 dx_setsvcond() 420
 dx_setsvmt() 424
 dx_settone() 428
 dx_settonelen() 430
 dx_setuio() 363, 433
 dx_SetWaterMark() 436
 dx_stopch() 181, 318, 357, 438
 DX_STREAMSTAT data structure 505
 DX_SVCB data structure 420, 507
 DX_SVMT data structure 424, 511
 dx_TSFStatus() 442
 dx_TxIottData() 444
 dx_TxRxIottData() 447
 DX_UIO data structure 513
 used by dx_setdevuio() 365
 dx_unlisten() 452
 dx_unlistenEx() 454
 dx_wtcallid() 457
 dx_wtring() 400, 460
 DX_XPB data structure 514
 examples (Springware) 517
 DXCH_PLAYDRATE 284
 DXCH_RECRDRATE 319
 dxxlib.h 413

E

echo cancellation 415
 enabling detection
 user-defined tones 191
 enhanced call progress analysis 23
 errors
 call progress analysis 55
 listing (voice library) 529
 ev_data 500
 ev_event 500
 event
 mask 401
 event block structure 224
 events 19
 call status transition (CST) 470
 categories 467
 connect 40
 disabling 155
 inter-process communication 360
 termination, list 467

- extended attribute functions
 - ATDX_ANSRSIZ() 40
 - ATDX_BDNAMEP() 42
 - ATDX_BDTYPE() 44
 - ATDX_BUFDIGS() 46
 - ATDX_CHNAMES() 48
 - ATDX_CHNUM() 50
 - ATDX_CONNTYPE() 52
 - ATDX_CPERROR() 55
 - ATDX_CPTERM() 58
 - ATDX_CRTNID() 61
 - ATDX_DEVTYPE() 67
 - ATDX_DTNFAIL() 69
 - ATDX_FRQDUR() 72
 - ATDX_FRQDUR2() 74
 - ATDX_FRQDUR3() 76
 - ATDX_FRQHZ() 78
 - ATDX_FRQHZ2() 80
 - ATDX_FRQHZ3() 82
 - ATDX_FRQOUT() 84
 - ATDX_FWVER() 86
 - ATDX_HOOKST() 89
 - ATDX_LINEST() 91
 - ATDX_LONGLOW() 93
 - ATDX_PHYADDR() 95
 - ATDX_SHORTLOW() 97
 - ATDX_SIZEHI() 99
 - ATDX_STATE() 101
 - ATDX_TERMMSK() 103
 - ATDX_TONEID() 107
 - ATDX_TRCOUNT() 110
- extended attribute functions category 25
- extended digit buffer 480

F

- fax resource 234
- feature information data structure 519, 521
- FEATURE_TABLE data structure 518
- file format 514
- file manipulation functions 24
 - dx_fileclose() 194
 - dx_fileerrno() 196
 - dx_fileopen() 199
 - dx_fileread() 201
 - dx_fileseek() 204
 - dx_filewrite() 207
- firmware
 - buffer 46
 - emulated D/4x version number 86
 - returning version number 86
- firmware digit buffer 161
- fixed length string 232

- flash character 180
- flushing digit buffer 161
- FSK
 - two-way 447
- full-duplex connection 34
- functions
 - ADSI 18
 - ATDX_ 25
 - call progress analysis 23
 - call status transition Event 19
 - caller ID 24
 - configuration 15
 - device management 14
 - extended attribute 25
 - global tone detection 21
 - global tone generation 22
 - I/O 16
 - I/O convenience 17
 - speed and volume 22
 - speed and volume convenience 22
 - structure clearance 24
 - TDM routing 20
 - Windows
 - close() 155

G

- G.711 PCM voice coder 514
- G.726 voice coder 514
- global dial pulse detection
 - support 398
- global tone detection
 - adding a tone 116
 - deleting tones 174
 - disabling 188
 - dual frequency cadence tones 132
 - dual frequency tones 129
 - enabling 191
 - enabling detection 116
 - functions 21
 - dx_addtone() 116
 - dx_blddt() 129
 - dx_blddtcad() 132
 - dx_bldst() 135
 - dx_bldstcad() 138
 - dx_deltone() 174
 - dx_distone() 188
 - dx_enbtone() 191
 - dx_setgtdamp() 406
 - removing tones 174
 - single frequency cadence tones 138
 - single frequency tones 135

- Global Tone Generation
 - adding a tone template 428
- global tone generation
 - functions 22
 - dx_bldtngen() 141
 - dx_playtone() 298
 - dx_playtoneEx() 302
 - playing a cadenced tone 302
 - playing a tone 298
 - template 524
- GSM voice coder 514
- GTD Frequency Amplitude
 - setting 406

H

- half-duplex connection 34
- hedge 492
- hi1bmax 493
- hi1ceil 495
- hi1tola 493
- hi1tolb 493
- higtch 494
- hisiz 494
- hook state 89, 155
 - setting 408
 - termination events 408
- hookstate 408
- hung channel 351

I

- I/O
 - function 103
 - transfer parameter block structure 514
 - transfer table 502
 - user-defined structure for 513
- I/O convenience functions 17

- I/O functions 16
 - dx_dial() 177
 - dx_dialtp() 185
 - dx_getdig() 215
 - dx_getdigEx() 221
 - dx_mreciottdata() 270
 - dx_play() 283
 - dx_playiottdata() 294
 - dx_rec() 318
 - dx_reciottdata() 330
 - dx_recm() 335
 - dx_recmf() 341
 - dx_resetch() 351
 - dx_RxIottData() 356
 - dx_setdigbuf() 395
 - dx_stopch() 438
 - dx_TxIottData() 444
 - dx_TxRxIottData() 447
- inter-process event communication 360
- intflg 491, 494
- io_bufp 503
- IO_CONT 165, 503
- IO_DEV 502
- IO_EOT 165, 502, 503
- io_fhandle 503
- io_length 503
- IO_LINK 165, 503
- IO_MEM 502
- io_nextp 503
- io_offset 503
- io_prevp 503
- IO_STREAM 502
- io_type 502
- IO_UIO 502
- IO_USEOFFSET 503

J

- JCLIP
 - message types 253

L

- lcdly 492
- lcdly1 492
- leading edge notification
 - user-defined tones 129
- learn mode functions 167, 171, 315
- line status 101
- lo1bmax 493

- lo1ceil 495
- lo1rmax 494
- lo1tola 493
- lo1tolb 493
- lo2bmax 493
- lo2rmin 494
- lo2tola 493
- lo2tolb 493
- loglch 494
- loop current
 - drop 52
- loop pulse detection 397
- lower2frq 495
- lower3frq 496
- lowerfrq 495

M

- maxansr 495
- MD_ADPCM 284, 319
- MD_GAIN 319
- MD_NOGAIN 319
- MD_PCM 284, 319
- message type ID 250
- message types
 - ACLIP (multiple data message) 252
 - CLASS (multiple data message) 252
 - CLIP 252
 - common to CLASS, ACLIP, and CLIP 251
 - JCLIP (multiple data message) 253
- MF
 - detection 535
 - detection errors 534
 - digit detection 397
 - digits
 - collection 216, 221
 - support 180, 398
 - tone specifications 533
- MF digits
 - overlap with DTMF digits 217
- monitor channels 224
- monitoring events 224
- mu-law 514
- mvertime2frq 496
- mvertime3frq 496
- mvertimefrq 495

N

- names
 - board device 42
- nrbeg 495
- nbrdna 492
- non-standard I/O devices
 - dx_setdevuio() 365
 - dx_setuio() 433
- nsbusy 494
- numsegs 525

O

- offset 503
- offtime 525
- off-hook 89
- off-hook state 408
- OKI ADPCM voice coder 514
- on-hook 89
- on-hook state 408
- open() function 279
- opening devices 278
- operator intercept 72

P

- parameter settings
 - getting current 231
- parameters
 - board and channel 413, 415, 416, 417
 - call progress analysis 159
 - sizes 232
- pause character 180
- PBX call progress signals 303
- physical address 95

- play
 - asynchronous 285
 - convenience function 291
 - default algorithm 284
 - default rate 284
 - encoding algorithm 284
 - mode 285
 - pausing 508
 - resuming 508
 - specifying mode 284
 - specifying number of bytes 503
 - synchronous 285
 - termination 286
 - TDX_PLAY 285
 - termination events 285
 - tone
 - asynchronous 298
 - asynchronous termination events 298
 - synchronous operation 299
 - transmitting tone before 284
 - voice data 307
- play and record functions
 - dx_mreciottdata() 270
 - dx_play() 283
 - dx_playf() 291
 - dx_playvox() 307
 - dx_rec() 318
 - dx_recf() 326
 - dx_reciottdata() 330
 - dx_recvox() 345
 - dx_recwav() 348
- playback
 - bytes transferred 110
- playing
 - see play 285
- playing voice data 294
- PM_ADSI 284
- PM_BYTE 232
- PM_FLSTR 232
- PM_INT 232
- PM_LONG 232
- PM_SHORT 232
- PM_SR6 284
- PM_SR8 284
- PM_TONE 284, 430
- PM_VLSTR 232
- positive answering machine detection 52
- positive voice detection 52
- pre-record beep 430
- processes per channel 225
- Pulse Code Modulation 284, 319

R

- recording
 - algorithm 319
 - asynchronous 320, 336
 - asynchronous termination event
 - TDX_RECORD 320, 336
 - bytes transferred 110
 - convenience function 326
 - default algorithm 319
 - default gain setting 319
 - default sampling rate 319
 - gain control 319
 - mode 319, 320, 336, 342
 - sampling rate 319
 - specifying mode 319
 - specifying number of bytes 503
 - stopping 318
 - synchronous 321, 336
 - synchronous termination 321
 - voice data 318, 330, 335, 341, 345
 - WAVE data 348
 - with A-law 319
 - with tone 319
 - with voice activity detector (VAD) 331
- resources
 - DSP fax 234
- rings
 - wait for specified number 460
- RLS_DTMF 91
- RLS_HOOK 91
- RLS_LCSSENSE 91
- RLS_RING 91
- RLS_RINGBK 91
- RLS_SILENCE 91
- RM_ALAW 319
- RM_SR6 319
- RM_SR8 319
- RM_TONE 319, 430
- routing functions
 - ag_getctinfo() 28
 - ag_getxmitslot() 30
 - ag_listen() 34
 - ag_unlisten() 38
 - dx_getctinfo() 210
 - dx_getxmitslot() 241
 - dx_listen() 263
 - dx_listenEx() 266
 - dx_unlisten() 452
 - dx_unlistenEx() 454

S

- sampling rates 515
- SC_TSINFO data structure 30, 34, 523
- sctools.c 463, 465
- setting hook state 408
 - asynchronous 408
 - synchronous 409
- short messaging services (SMS) 18
- SIGALRM 461
- sigset() 461
- SIT sequence
 - returning 62
- SIT tones
 - detection 74, 76, 78, 80, 82
- Softfax 519, 521
 - using dx_GetRscStatus() 234
- Special Information Tone (SIT) sequence
 - returning 62
- speed
 - adjusting 112
 - adjustment conditions 420
 - explicitly adjusting 125
 - retrieving current 212
- speed and volume
 - current 126
 - data structure 507
 - last modified 126
 - modification table
 - setting 511
 - resetting to origin 126
- speed and volume convenience functions
 - dx_addspddig() 112
 - dx_addvoldig() 121
- speed and volume function
 - dx_setsvmt() 424
- speed and volume functions 22
 - dx_adjsv() 125
 - dx_clrsvcond() 163
 - dx_getcursv() 212
 - dx_getsvmt() 238
 - dx_setsvcond() 420
- speed and volume modification table
 - resetting to defaults 424, 425
 - retrieving contents 238
 - specifying speed 424
 - specifying volume 424
 - updating 424
- speed control 511
- sr_getevtdatap() 402
- stop I/O functions
 - dial 438
 - termination reason
 - TM_USRSTOP 438
- stopping call progress analysis 439
- stopping I/O functions
 - synchronous 438
- streaming to board
 - creating stream buffer 281
 - deleting stream buffer 157
 - DX_STREAMSTAT data structure 505
 - function summary 18
 - getting status info 236
 - putting data in buffer 313
 - resetting internal data 354
 - setting water mark 436
- structure clearance functions 24
 - dx_clrcap() 159
 - dx_clrtp() 165
- structures
 - clearing 159, 165
 - digit buffer 215
 - DV_DIGIT 215
 - DV_DIGITEX 221
 - DX_CAP 159
 - DX_EBLK 224
 - DX_IOTT 283
 - event block 224
 - extended digit buffer 221
- stuck channel 351
- SV_ABSPOS 126
- SV_CURLASTMOD 126
- SV_CURORIGIN 126
- SV_RELCURPOS 126
- SV_RESETOG 126
- SV_SPEEDTBL 126
- SV_TOGGLE 126
- SV_TOGORIGIN 126
- SV_VOLUMETBL 126
- synchronous operation
 - dial 178
 - digit collection 216
 - play 285
 - playing tone 299
 - record 321, 336
 - setting hook state 409
 - stopping I/O functions 438, 439

T

TDM bus

time slot information structure 523

TDM bus routing functions 20

dx_getctinfo() 210

dx_getxmitslot() 241

dx_listen() 263

dx_listenEx() 266

dx_unlisten() 452

dx_unlistenEx() 454

TDX_CALLP event 178, 468

TDX_CREATETONE event 468

TDX_CREATETONE_FAIL event 468

TDX_CST event 468

TDX_DELETETONE event 468

TDX_DELETETONE_FAIL 468

TDX_DIAL event 178, 468

TDX_ERROR event 468

TDX_GETDIG event 468

TDX_HIGHWATER event 469

TDX_LISTEN event 468

TDX_LISTEN_FAIL event 468

TDX_LOWWATER event 469

TDX_NOSTOP event 468

TDX_PLAY event 285, 468

TDX_PLAYTONE event 298, 303, 468

TDX_QUERYTONE event 468

TDX_QUERYTONE_FAIL event 469

TDX_RECORD event 320, 336, 342, 469

TDX_RESET event 351

TDX_RESETERR event 351

TDX_RXDATA event 449, 469

TDX_SETHOOK 500

TDX_SETHOOK event 408, 469, 498

TDX_TXDATA event 449, 469

TDX_UNDERRUN event 469

TDX_UNLISTEN event 469

TDX_UNLISTEN_FAIL event 469

TDX_VAD event 469

termination

call progress analysis 58

stop I/O function 438

synchronous record 321

termination conditions 16

termination events 467

DX_CST data structure 408

TDX_SETHOOK 408

termination parameter table structure 481

terminations

asynchronous play 285

ATDX_TERMMSK() 103

end of data 103, 104, 357, 445, 448

function stopped 104, 357, 445, 449

I/O device error 103, 104, 357, 445, 449

I/O function 103

I/O functions 438

inter-digit delay 103, 104

loop current off 104

maximum DTMF count 103, 104

maximum function time 103, 104, 357, 445, 449

maximum period of non-silence 104

maximum period of silence 103, 104

normal termination 103, 104

pattern matched 104

specific digit received 103, 104

synchronous play 286

tone-on/off event 103, 104

tg_dflag 524

tg_freq1 524

TID_BUSY1 61, 62

TID_BUSY2 61, 62

TID_DIAL_INTL 61, 62

TID_DIAL_LCL 61, 62

TID_DIAL_XTRA 62

TID_DISCONNECT 61, 62

TID_FAX1 61, 62

TID_FAX2 61, 63

TID_RINGBK1 61, 63

TID_RINGBK2 62, 63

TID_SIT_ANY 62

TID_SIT_IC 62

TID_SIT_INEFFECTIVE_OTHER 62

TID_SIT_IO 62

TID_SIT_NC 62

TID_SIT_NC_INTERLATA 62

TID_SIT_NO_CIRCUIT 62

TID_SIT_NO_CIRCUIT_INTERLATA 62

TID_SIT_OPERATOR_INTERCEPT 62

TID_SIT_REORDER_TONE 62

TID_SIT_REORDER_TONE_INTERLATA 62

TID_SIT_RO 62

TID_SIT_RO_INTERLATA 62

TID_SIT_VACANT_CIRCUIT 62

TID_SIT_VC 62

time slot device information structure 475

time2frq 496

time3frq 496

- timefrq 495
- TM_DIGIT termination 103, 104
- TM_EOD termination 103, 104, 357, 445, 448
- TM_ERROR termination 103, 104, 357, 445, 449
- TM_IDDTIME termination 103, 104
- TM_LCOFF termination 104
- TM_MAXDTMF termination 104
- TM_MAXDTMFtermination 103
- TM_MAXNOSIL termination 104
- TM_MAXSIL termination 103, 104
- TM_MAXTIME termination 103, 104, 357, 445, 449
- TM_NORMTERM termination 103, 104
- TM_PATTERN termination 104
- TM_TONE termination 103, 104
- TM_USRSTOP termination 104, 357, 445, 449
- TN_GEN data structure 524, 525
 - used by dx_settone() 428
- TN_GENCAD data structure 525
- tone 526
 - adding 116
 - enabling detection 116
- tone definitions 141, 144, 148, 152
- tone generation template 524
- tone ID 107, 129, 168, 172, 316
- tone identifier 61
- tone template 428
- tone, pre-record beep 430
- TONE_DATA data structure 167, 315
- trailing edge notification
 - user-defined tones 129
- Transaction Record
 - enabling using dx_recm() 335
 - enabling using dx_recmf() 341
- transaction record feature 270
- TSF function 442
- two-way FSK 447

U

- unsolicited events 469
- upper2frq 495
- upper3frq 496
- upperfrq 495
- user digit buffer 478
- user-defined
 - cadence 133
- user-defined digits
 - collection 216, 221

- user-defined functions
 - installing 365, 433
- user-defined input/output data structure 513
- user-defined tone ID 107
- user-defined tones 116
 - cadence repetition 133
 - disabling detection 188
 - dual frequency 129
 - dual frequency cadence 132
 - enabling detection 191
 - first frequency 129
 - first frequency deviation 129
 - ID 129
 - leading or trailing edge notification 129
 - playing 302
 - also see playing tone 298
 - removing 174
 - second frequency 129
 - second frequency deviation 129
 - single frequency 135
 - single frequency cadence 138

V

- variable length string 232
- version number
 - firmware 86
- voice activity detector (VAD) 331
- voice coders 514
 - examples (Springware) 517
- volume
 - adjusting 121
 - adjustment conditions 420
 - explicitly adjusting 125
 - retrieving current 212
- volume control 511

W

- water mark 436
- WAVE files
 - playing 310
- Windows functions
 - close() 155

Z

- zero-train DPD 397