

Voice Software Reference - Features Guide for Linux

Copyright © 2001 Dialogic Corporation

05-1454-003

COPYRIGHT NOTICE

Copyright © 2001 Dialogic Corporation. All Rights Reserved.

All contents of this document are subject to change without notice and do not represent a commitment on the part of Dialogic Corporation. Every effort is made to ensure the accuracy of this information. However, due to ongoing product improvements and revisions, Dialogic Corporation cannot guarantee the accuracy of this material, nor can it accept responsibility for errors or omissions. No warranties of any nature are extended by the information contained in these copyrighted materials. Use or implementation of any one of the concepts, applications, or ideas described in this document 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 condone or encourage such infringement. Dialogic makes no warranty with respect to such infringement, nor does Dialogic waive any of its own intellectual property rights which may cover systems implementing one or more of the ideas contained herein. Procurement of appropriate intellectual property rights and licenses is solely the responsibility of the system implementer. The software referred to in this document is provided under a Software License Agreement. Refer to the Software License Agreement for complete details governing the use of the software.

All names, products, and services mentioned herein are the trademarks or registered trademarks of their respective organizations and are the sole property of their respective owners. DIALOGIC (including the Dialogic logo), DTI/124, and SpringBoard are registered trademarks of Dialogic Corporation. A detailed trademark listing can be found at: <http://www.dialogic.com/legal.htm>.

Publication Date: November, 2001

Part Number: 05-1454-003

Dialogic, an Intel company
1515 Route 10
Parsippany NJ 07054
U.S.A.

For **Technical Support**, visit the Dialogic support website at:
<http://support.dialogic.com>

For **Sales Offices** and other contact information, visit the main Dialogic website at:
<http://www.dialogic.com>

Table of Contents

Organization of This Guide	xi
1. Introduction	1
1.1. Dialogic Board Naming Conventions	1
1.1.1. Function Identifiers	2
1.1.2. Channel Identifiers	2
1.1.3. Suffix Identifiers	3
1.2. Virtual Boards	3
1.3. Generic Configuration File	4
1.3.1. Name	5
1.3.2. Ioport	6
1.4. Call Analysis	7
1.5. Global Tone Detection/Generation	8
1.5.1. Global Tone Detection	8
1.5.2. Global Tone Generation	8
1.6. R2 MF Signaling	9
1.7. Analog Display Services Interface (ADSI)	9
1.8. Speed and Volume Control	9
1.9. Caller ID	9
1.10. Global Dial Pulse Detection	10
1.11. Transaction Record	10
1.12. Silence Compressed Record	10
1.13. Echo Cancellation Resource	11
1.14. G.726 ADPCM Voice Coder	11
1.15. Voice Library Demo Programs	12
2. Call Analysis.....	13
2.1. What Does Call Analysis Detect?	13
2.2. How Does Call Analysis Work?	14
2.3. How to Enable PerfectCall Call Analysis	16
2.3.1. Modifying the Default Tone Definitions	16
2.3.2. Activating PerfectCall Call Progress	17
2.4. How to Use Call Analysis	18
2.4.1. Set Up the Call Analysis Parameter Structure (DX_CAP)	19
2.4.2. Use the dx_dial() Function to Initiate Call Analysis	20
2.4.3. Determine the Outcome of the Call	20
2.4.4. Obtain Additional Call Outcome Information	23

Voice Software Reference - Features Guide for Linux

2.5. How the DX_CAP Controls Call Analysis	24
2.5.1. Selecting SIT Frequency Detection, Positive Voice Detection, and Positive Answering Machine Detection.....	24
2.5.2. SIT Frequency Detection	25
2.5.3. Cadence Detection in Basic Call Analysis	32
2.5.4. Tone Detection in PerfectCall Call Analysis.....	44
2.5.5. Loop Current Detection	49
2.5.6. Positive Voice Detection.....	50
2.6. Call Analysis Errors.....	51
3. Global Tone Detection/Generation.....	53
3.1. Global Tone Detection.....	53
3.1.1. Defining GTD Tones	54
3.1.2. Building Tone Templates.....	54
3.1.3. Working with Tone Templates.....	56
3.1.4. Tone Event Retrieval	57
3.1.5. Memory Available for User-Defined Tone Templates.....	58
3.1.6. Applications.....	62
3.2. Global Tone Generation (GTG)	64
3.2.1. Global Tone Generation Functions	64
3.2.2. Building and Implementing a Tone Generation Template	64
4. R2 MF Signaling.....	67
4.1. Direct Dialing-In Service.....	68
4.2. R2 MF Multifrequency Combinations.....	68
4.3. R2 MF Signal Meanings	71
4.4. R2 MF Compelled Signaling	79
4.5. Using R2 MF Signaling with Voice Boards	81
4.6. Related Publications	82
5. Analog Display Services Interface.....	83
5.1. ADSI Protocol	83
5.2. Dialogic ADSI Support.....	84
5.3. Related Publications	87
6. Speed and Volume Control.....	89
6.1. Speed and Volume Convenience Functions.....	89
6.2. Speed and Volume Adjustment Functions.....	89
6.3. Speed and Volume Modification Tables.....	90
6.4. Play Adjustment Digits.....	95
6.5. Setting Play Adjustment Conditions	95
6.6. Explicitly Adjusting Speed and Volume.....	96

Table of Contents

7. Caller ID	97
7.1. Overview	97
7.2. Supported Formats	97
7.3. Related References	98
7.4. Theory of Operation	98
7.5. Accessing Caller ID Information	99
7.6. Enabling Channels to Use the Caller ID Feature	101
7.7. Error Handling	101
8. Global Dial Pulse Detection	103
8.1. DPD Parameters	103
8.2. Global DPD Demonstration Program	104
8.3. Global DPD Application Programming Interface	104
8.3.1. Programming Procedure and Example	105
8.3.2. Programming Considerations	111
9. Transaction Record	113
10. Silence Compressed Record	115
11. Echo Cancellation	117
11.1. Overview	117
11.1.1. Modes of Operation	119
11.1.2. Buffer Size Adjustments for Internet Telephony	121
11.2. ECR Application Models	122
11.2.1. How to Set Up the ECR Bridge	122
11.2.2. How to Set Up an ECR Play Over the SCbus	127
12. G.726 ADPCM Voice Coder	131
12.1. Dialogic Support for G.726	131
12.2. Enabling and Using the G.726 Voice Coder	131
13. Voice Library Demo Programs	133
13.1. D/40demo - Synchronous Demo	134
13.1.1. Boards Supported	134
13.1.2. Physical Connections	134
13.1.3. Running the d40demo Program	137
13.1.4. d40demo Program Overview	138
13.1.5. Source Code Overview	139
13.1.6. Initialization	141
13.1.7. Menu System Routine	143
13.1.8. Messaging System Routine	153
13.2. Other Synchronous Demos	155

Voice Software Reference - Features Guide for Linux

13.2.1. Custserv Demo	155
13.2.2. Horoscope Demo	157
13.3. Asynchronous Demo Programs <i>pansr</i> and <i>cbansr</i>	158
13.4. Caveats	160
Appendix A - Related Publications	161
Glossary	163
Index	179

List of Tables

Table 1. Valid Values for Generic Configuration File	7
Table 2. Special Information Tone Sequences	26
Table 3. Standard Bell System Network Call Progress Tones	56
Table 4. Maximum Memory Available for User-Defined Tone Templates on Dialogic Voice and Voice/Fax Boards	59
Table 5. Maximum Memory Available for Tone Templates for Tone-Creating Voice Features	60
Table 6. Forward Signals, CCITT Signaling System R2 MF tones	69
Table 7. Backward Signals, CCITT Signaling System R2 MF tones.....	70
Table 8. Purpose of Signal Groups and Changeover in Meaning	72
Table 9. Meanings for R2 MF Group I Forward Signals	75
Table 10. Meanings for R2 MF Group II Forward Signals.....	76
Table 11. Meanings for R2 MF Group A Backward Signals	77
Table 12. Meanings for R2 MF Group B Backward Signals	78
Table 13. Speed Modification Table	92
Table 14. Volume Modification Table	94
Table 15. Supported Caller ID Information	99

Voice Software Reference - Features Guide for Linux

List of Figures

Figure 1. Basic Call Analysis Components.....	15
Figure 2. PerfectCall Call Analysis Components.....	15
Figure 3. Call Analysis Outcomes for Basic Call Analysis.....	21
Figure 4. Call Analysis Outcomes for PerfectCall Call Analysis.....	22
Figure 5. Standard Busy Signal	34
Figure 6. Standard Single Ring.....	34
Figure 7. Type of Double Ring.....	34
Figure 8. Cadence Detection.....	35
Figure 9. Elements of Established Cadence	36
Figure 10. No Ringback Due to Continuous No Signal	39
Figure 11. No Ringback Due to Continuous Nonsilence	40
Figure 12. Cadence Detection Salutation Processing	43
Figure 13. Forward and Backward Interregister Signals.....	67
Figure 14. Multiple Meanings for R2 MF Signals	71
Figure 15. R2 MF Compelled Signaling Cycle.....	80
Figure 16. Example of R2 MF Signals for 4-digit DDI Application.....	81
Figure 17. SCR Parameters.....	116
Figure 18. Echo Canceller with Relevant Input and Output Signals.....	118
Figure 19. Echo Canceller Operating over an SCbus	118
Figure 20. ECR Bridge Example Diagram	122
Figure 21. ECR Play Over the SCbus.....	127
Figure 22. Voice Library Demo Directory Structure	133
Figure 23. Connecting D/41ESC Board to Phone.....	135
Figure 24. Connecting D/41ESC Board to Phone.....	136
Figure 25. Connecting D/160SC-LS Board to Phone	137

Voice Software Reference - Features Guide for Linux

Organization of This Guide

The Voice Features Guide for Linux describes the major features provided with the voice software for Linux.

Chapter 1 provides an overview of the voice features described in this guide.

Chapter 2 describes Call Analysis, including how to use the DX_CAP Call Analysis structure.

Chapter 3 describes Global Tone Detection and Global Tone Generation.

Chapter 4 describes the R2 MF international signaling system and the support provided for it by Dialogic voice boards.

Chapter 5 describes how to display information on a display-based telephone using the Analog Display Services Interface (ADSI).

Chapter 6 describes how to control play speed and play volume.

Chapter 7 describes how to use the Caller ID feature which can display useful information about the calling party.

Chapter 8 describes how to use Global Dial Pulse Detection (DPD) to detect dial pulses from either rotary or pulse phones, from any country's telephones.

Chapter 9 provides an overview of the Transaction Record feature, which allows voice activity on two channels to be summed and stored in a single file, or in a combination of files, devices, and/or memory.

Chapter 10 describes the Silence Compressed Record (SCR) feature, which reduces the size of recording files by eliminating silent pauses.

Chapter 11 describes the Echo Cancellation Resource feature, which allows a voice channel to dynamically perform echo cancellation on any external SCbus time slot signal.

Voice Software Reference - Features Guide for Linux

Chapter 12 describes the Dialogic G.726 bit exact voice coder. G.726 is an ITU-T recommendation that specifies an adaptive differential pulse code modulation (ADPCM) technique for recording and playing back audio files.

Chapter 13 describes the Voice Library demo programs and how to run them.

Appendix A provides a list of related Dialogic publications.

A Glossary and an Index are also provided.

1. Introduction

This chapter provides an overview of the major features of the voice software for Linux. It explains the naming systems used to designate Dialogic voice processing boards, and discusses the related concepts of virtual boards and virtual boards.

The following major features of the voice software for Linux are described in this guide:

- Call Analysis
- Global Tone Detection and Global Tone Generation
- R2 MF Signaling
- Analog Display Services Interface (ADSI)
- Speed and Volume Control
- Caller ID
- Global Dial Pulse Detection
- Transaction Record
- Silence Compressed Record (SCR)
- Echo Cancellation
- G.726 ADPCM Voice Coder support
- Voice Library Demo Programs

1.1. Dialogic Board Naming Conventions

Dialogic voice processing boards are identified by letter prefixes that specify the functions of the boards. Each letter prefix is followed by a forward slash and a number that specifies the number of channels the board can support. In most cases a suffix is appended to further identify the board. These alphanumeric identifiers are defined in the following paragraphs.

1.1.1. Function Identifiers

Many Dialogic boards have multiple functions. The following prefixes specify the primary functions of the boards:

Prefix	Board Function
D/	Voice store-and-forward (may have additional functions such as network interface)
DCB/	Audio conferencing
DTI/	Digital T-1 or E-1 telephony interface
LSI/	Loop-start (analog) interface board with tone and call progress capabilities
MSI/	Modular Station Interface board for connecting devices to phones, with conferencing support
SCX/	Switchless adapter board that converts SCbus TTL signals to SCxbus SCSI-3 RS-485 type signals, and vice-versa
VFX/	Combined voice and fax resource board with on-board loop-start interfaces

The following general terms specify product groups:

- **Antares** boards are stand-alone automatic speech recognition and text-to-speech boards.
- **DIALOG/HD** boards are high density (having 8 or more channels) voice boards, digital network interface boards, analog (loop start) network interface boards, and integrated voice and network boards.

1.1.2. Channel Identifiers

Following the function prefix and forward slash in the board name, a number is used to specify the number of channels the board can support. To determine the number of channels on a board, use the channel specifier number but ignore the last digit. For example, a D/160SC board can support 16 voice channels.

1.1.3. Suffix Identifiers

The following suffixes add information about the functions and capabilities of most boards:

50	(Antares only) Clock speed of the on-board DSP in megahertz
75	Has a 75-ohm digital telephone network interface
120	Has a 120-ohm digital telephone network interface
D	A revision level of the board
E	A revision level of the board
E1	Has a single digital E-1 telephone network interface
2E1	Has two digital E-1 telephone network interfaces (DualSpan board)
H	A revision level of the board
HD	A high density (having more than 8 channels) board
IDPD	Has Global Dial Pulse Detection capability
LS	Has loop start (analog) interfaces
PCI	Connects to a PCI slot on the PC motherboard
plus	Board with added features
R	(on MSI/SC boards) The power ring option
SC	Can communicate with other boards via the SCbus
T1	Has a single digital T-1 telephone network interface
2T1	Has two digital T-1 telephone network interfaces (DualSpan board)

1.2. Virtual Boards

The Dialogic driver views voice boards with more than four channels as multiple emulated D/4x and D/2x boards, where a D/4x board is equivalent to a D/4xD or D/4xE board and a D/2x board is equivalent to a D/2xD or D/2xE board. The emulated D/4x and D/2x boards are called “virtual boards.” To the driver, for example:

- D/80SC boards emulate two D/4x boards.
- D/160SC boards emulate four D/4x boards.
- D/240SC and D/240PCI-T1 boards emulate six D/4x boards.

Voice Software Reference - Features Guide for Linux

- D/300SC-E1, D/300SC-2E1, and D/300PCI-E1 boards emulate seven D/4x boards and one D/2x board.
- D/320SC boards emulate eight D/4x boards.
- D/480SC-2T1 boards emulate 12 D/4x boards.
- D/600SC-2E1 boards emulate 14 D/4x boards and 2 D/2x boards.
- D/640SC boards emulate 16 D/4x boards.

The emulation feature of these boards allows them to be used with code written for D/2x and D/4x boards.

1.3. Generic Configuration File

This section describes the format of the Generic Configuration file, *.voxcfg*, an internal file which is generated at download time. The Generic Configuration file is located in the */usr/dialogic/cfg* directory. The *.voxcfg* file can be browsed to determine which virtual boards were assigned to a given physical board.

CAUTION

The user should not edit this file!

A board may be viewed by the system as one or more virtual boards. The Generic Configuration File is an ASCII file that contains one line for each virtual board in the system. The following number of lines are required for each board:

- 1 line for each two- or four-channel resource board
- 2 lines for each D/80SC board
- 4 lines for each D/160SC-LS board
- 6 lines for each D/240SC board
- 7 lines for each D/240SC-T1 or D/240PCI-T1 board
- 8 lines for each D/240SC-2T1 board
- 8 lines for each D/320SC board
- 9 lines for each D/300SC-E1, D/300PCI-E1, and D/300PSC-E1 board

1. Introduction

- 10 lines for each D/300SC-2E1 board
- 20 lines for each D/600SC-2E1 board
- 16 lines for each D/640SC board
- 2 lines for each DCB/320SC board
- 3 lines for each DCB/640SC board
- 4 lines for each DCB/960SC board
- 1 line for each DTI/240SC, DTI/241SC, DTI/300SC, DTI/301SC board
- 2 lines for each DTI/480SC, DTI/481SC, DTI/600SC, DTI/601SC board
- 8 lines for each LSI/81SC
- 16 lines for each LSI/161SC board
- 1 line for each MSI/160SC and MSI/240SC board
- 1 line for each SCX/160 board

Comments are included in the configuration file by placing a pound sign (#) as the first character on each comment line. The default Generic Configuration File contains comments explaining the file format.

The following sections describe the **Name** and **Ioport** fields in the Generic Configuration file in detail.

1.3.1. Name

The **Name** field contains the device name that is assigned to the real or emulated D/4x board. This name is used by the **dx_open()** or **dt_open()** library function to open the device.

Names may be up to 15 characters long. Specify board names as follows:

- For D/41ESC boards, specify a unique name for each board.
- For D/2x and D/4x boards, specify a unique name for each board.
- For DIALOG/HD voice boards, specify a unique name for each of the several emulated D/4x boards.

Voice Software Reference - Features Guide for Linux

- For DIALOG/HD network interface daughtercards, specify a unique name for each daughtercard.

Channel device names are generated automatically and named by appending “<Subdev><n>” to the associated board’s device name. <Subdev> is the subdevice specified in the **Subdev** field of the configuration file, and <n> is the channel number starting at one. For example, the board called dxxxB1 above will have channels named dxxxB1C1, dxxxB1C2, dxxxB1C3, and dxxxB1C4 by default. The 24 trunk interface lines of the dtiB1 board will be named dtiB1T1 through dtiB1T24.

For DIALOG/HD boards, the **Name** field in the Generic Configuration File must not be the same as the **Name** field for the same device (or any other device) in the SpanCard Configuration File.

Within these rules, any name is permitted. A list of recommended entries for the **Name** parameter is provided in *Table 1*.

1.3.2. Ioport

The **Ioport** field is used differently for different boards.

- D/21D, D/21H, D/41D, and D/41H boards do not use **Ioport** at all; the field must be set to zero for these boards.
- For D/41EPCI boards, this field stores the thumbwheel identifier number. This is a hexadecimal number in the range 0 to 1F. Refer to the *Quick Installation Card* for information about setting the thumbwheel identifier number.
- For DIALOG/HD boards, the **Ioport** for the baseboard and for any network interface daughtercard is the thumbwheel identifier number. This is a hexadecimal number in the range 0 to F. Refer to the *Quick Installation Card* for information about setting the thumbwheel identifier number.

For voice daughtercards of DIALOG/HD boards, the **Ioport** value is the thumbwheel identifier number plus 20 (hex); it therefore falls into the range 20 to 2F.

Table 1. Valid Values for Generic Configuration File

Field	Values	Comments	
Name	dxxxB1	dtiB1	Names of the devices used by dx_open() and dt_open() to open the devices. (This field is described in detail in the accompanying text).
	dxxxB2	dtiB2	
	dxxxB3	dtiB3	
	.	.	
	.	.	
	.	.	
Nchn	2 (for D/2x boards) 4 (for all other voice boards) 24 (for T-1 interface card) 32 (for E-1 interface card)	Number of subdevices (channels) on the device. (Voice boards with more than four channels emulate multiple four-channel boards.)	
Ioport	DIALOG/HD boards: 0 through F	For a DIALOG/HD voice baseboard or network interface daughtercard, enter the thumbwheel ID number (in hexadecimal) of the board, as it was set at hardware configuration time.	
	20 through 2F	For a DIALOG/HD voice daughtercard, enter the thumbwheel number plus 20H (hexadecimal).	
	All other voice boards: 0	Specify 0 for all other voice boards.	

1.4. Call Analysis

Call Analysis is available on all voice boards. The following Call Analysis features are available on DSP boards only. It monitors the progress of an outbound call after it is dialed into the Public Switched Telephone Network (PSTN).

There are two forms of Call Analysis: Basic and PerfectCall. PerfectCall Call Analysis uses an improved method of signal identification and can detect fax

machines and answering machines. Basic Call Analysis provides backward compatibility for older applications written before PerfectCall Analysis became available.

NOTE: PerfectCall was formerly called “Enhanced” Call Analysis.

Call Analysis is initiated using the **dx_dial()** function which uses input from the Call Analysis Parameter (DX_CAP) data structure.

See Chapter 2. *Call Analysis* for detailed information about Call Analysis.

1.5. Global Tone Detection/Generation

Global Tone Detection (GTD) and Global Tone Generation (GTG) are available on all voice boards.

1.5.1. Global Tone Detection

Global Tone Detection allows a voice board to detect single- or dual-frequency tones other than DTMF tones 0-9, a-d, *, and #. Through GTD, a user can define the characteristics of a tone in order to detect a tone with the same characteristics. The characteristics of a tone can be defined and tone detection can be enabled using GTD functions provided in the Voice Library.

See Chapter 3. *Global Tone Detection/Generation* for detailed information about Global Tone Detection.

1.5.2. Global Tone Generation

Global Tone Generation permits the creation of user-defined tones using the TN_GEN template data structure, and allows the user to play the tone using the **dx_playtone()** function. The **dx_bldtngen()** function can be used to build the TN_GEN template.

See Chapter 3. *Global Tone Detection/Generation* for detailed information about Global Tone Generation.

1.6. R2 MF Signaling

R2 MF signaling is an international signaling system that is used in Europe and Asia to permit the transmission of numerical and other information relating to the called and calling subscribers' lines.

See Chapter 4. *R2 MF Signaling* for a description of the aspects of R2 MF signaling and how to use R2 MF signaling with the voice boards.

1.7. Analog Display Services Interface (ADSI)

The Analog Display Services Interface (ADSI) allows information to be transmitted for display on a display-based telephone connected to an analog loop-start line. The telephone must be a true ADSI-compliant device.

See Chapter 5. *Analog Display Services Interface* for a summary of the ADSI protocol followed by directions for implementing ADSI support using Dialogic functions.

1.8. Speed and Volume Control

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

Several functions and data structures can be used to control the speed and volume of play on a channel. See Chapter 6. *Speed and Volume Control* for instructions about using the functions and data structures to control play speed and play volume.

1.9. Caller ID

An application can enable the Caller ID feature on specific channels to process Caller ID information as it is received with an incoming call. Caller ID information can include the calling party's Directory Number (DN), the date and time of the call, and the calling party's subscriber name. See Chapter 7. *Caller ID* for more information on Caller ID.

1.10. Global Dial Pulse Detection

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

See Chapter 8. *Global Dial Pulse Detection* for more information on Global DPD.

1.11. Transaction Record

The Transaction Record feature allows voice activity on two channels to be summed and stored in a single file, or in a combination of files, devices, and/or memory.

NOTE: The Transaction Record feature is not supported on the D/41ESC, VFX/40ESC, and VFX/40ESCplus boards.

See Chapter 9. *Transaction Record* for more information on the Transaction Record feature.

1.12. Silence Compressed Record

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

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

You cannot enable the SCR feature through the Dialogic Voice API. It is enabled in the *voice.prm* file which is downloaded to the board during initialization. You must edit this file and set appropriate values for the SCR parameters for use in your working environment before initializing the board. Instructions for enabling

this feature are given in Chapter 10. *Silence Compressed Record* and the *Release 2 Software Installation Reference for Linux*.

1.13. Echo Cancellation Resource

Echo cancellation is a functional component of a Dialogic voice channel. The Dialogic Echo Cancellation Resource (ECR) feature is a new mode for voice channels. In ECR mode, a voice channel can dynamically perform echo cancellation on any external SCbus time slot signal.

Prior to the implementation of the ECR feature in the Dialogic Voice Library, each voice channel device had a single transmit (TX) SCbus time slot assigned to it for data communication across the SCbus. To connect one device to another across the SCbus, an application would call `xx_listen()` (where `xx_` is `ag_`, `dt_`, `dx_`, or `ms_`) on one voice or network device to connect to a second device's transmit channel. Any signal transmitted by the second device on its transmit channel (TX channel) would be received by the first device's receive channel (RX channel). For a full-duplex connection, the second device would then call `xx_listen()` to connect its receive channel to the first device's transmit channel.

The Dialogic ECR feature provides the ability to utilize echo cancellation on signals external to the voice channel. The echo cancellation capability becomes a system-wide resource that may be applied to any SCbus PCM stream. The addition of the ECR feature allows the application to dynamically configure a voice channel as either an echo cancellation device (ECR mode) or as a standard voice processing channel (SVP mode). In ECR mode, a portion of the standard voice functionality remains available while another portion of it becomes unavailable. See Chapter 11. *Echo Cancellation* for a detailed description of this feature and instructions for enabling it.

1.14. G.726 ADPCM Voice Coder

G.726 is an ITU-T recommendation that specifies an adaptive differential pulse code modulation (ADPCM) technique for recording and playing back audio files. It is useful for applications that require speech compression, encoding for noise immunity, and uniformity in transmitting voice and data signals.

Voice Software Reference - Features Guide for Linux

Dialogic provides a G.726 bit exact voice coder that is compliant with the ITU-T G.726 recommendation. Messages are stored at 32 Kbps using G.726 ADPCM.

Audio encoded in the G.726 bit exact format complies with Voice Profile for Internet Messaging (VPIM), a communications protocol that makes it possible to send and receive messages from disparate messaging systems over the Internet. G.726 bit exact is the audio encoding and decoding standard supported by VPIM. See Chapter 12. *G.726 ADPCM Voice Coder* for a detailed description of this feature and instructions for enabling it.

1.15. Voice Library Demo Programs

Five programs that demonstrate the use of Voice Library functions are provided with the voice software for Linux. The source and executable versions of each program, and a makefile to compile the source, are provided. See Chapter 13. *Voice Library Demo Programs* for instructions about running each of the demo programs.

2. Call Analysis

Call Analysis is a standard feature. It monitors the progress of an outbound call after it is dialed into the Public Switched Telephone Network (PSTN), where a wide variety of signal possibilities can occur. By using Call Analysis you can determine the following:

- if the line is answered and, in many cases, how the line is answered
- if the line rings but is not answered
- if the line is busy
- if there is a problem in completing the call

There are two forms of Call Analysis: Basic and PerfectCall. PerfectCall Call Analysis uses an improved method of signal identification, and can also detect fax machines and answering machines. Basic Call Analysis provides backward compatibility for older applications; any application which was written before PerfectCall Analysis became available will continue to work unchanged. However, it is recommended that all new applications be designed for PerfectCall Call Analysis.

NOTE: PerfectCall was formerly called “Enhanced” Call Analysis.

Call Analysis is initiated when a call is dialed using the **dx_dial()** function. This function uses input from the Call Analysis Parameter structure (DX_CAP). You can adjust the DX_CAP parameters to fit the needs of your application. When the Voice Driver determines the outcome of the call, information is returned using Extended Attribute functions. To find out how to use Call Analysis see Section 2.4. *How to Use Call Analysis*.

2.1. What Does Call Analysis Detect?

Call Analysis determines the outcome of the call from among the following possibilities:

Intercept A Special Information Tone (SIT) was detected; an invalid number was dialed or there was a problem completing the call. This is also known as an **operator intercept**.

No Ringback	No discernible signal pattern was detected.
Connect	The phone was answered.
No Answer	The line was ringing but was not answered.
Busy	A busy signal was detected.

The outcome of the call is returned to the application when Call Analysis has completed.

2.2. How Does Call Analysis Work?

Call Analysis uses the following techniques to determine the progress of the call:

- Cadence Detection
- Frequency Detection
- Loop Current Detection
- Positive Voice Detection and Positive Answering Machine Detection

Figure 1 and *Figure 2* illustrate these processes.

Frequency Detection, Cadence Detection, Loop Current Detection, Positive Voice Detection, and Positive Answering Machine Detection can all operate simultaneously during Call Analysis. See Section 2.5. *How the DX_CAP Controls Call Analysis* for more information.

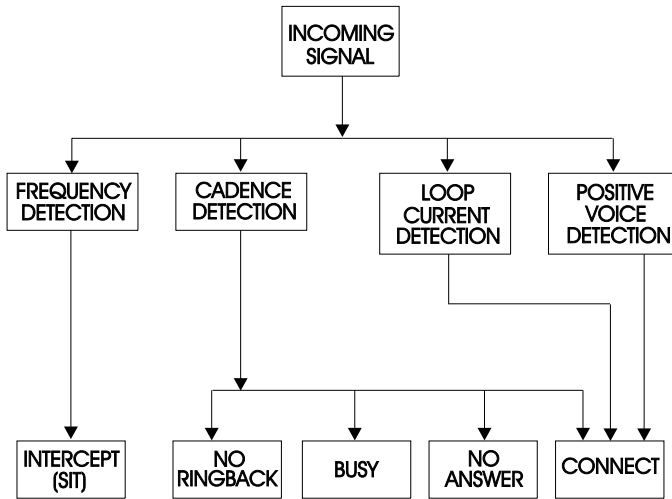


Figure 1. Basic Call Analysis Components

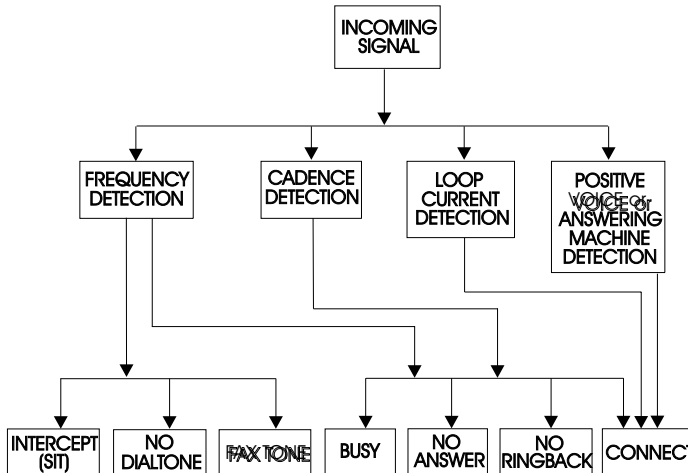


Figure 2. PerfectCall Call Analysis Components

The figures show that Cadence Detection is the sole means of detecting a no ringback, busy, or no answer when using basic call analysis. PerfectCall Call

Analysis uses Cadence Detection plus Frequency Detection to identify all of these signals plus fax machine frequencies. A connect can be detected through the complementary methods of Cadence Detection, Frequency Detection, Loop Current Detection, Positive Voice Detection, and Positive Answering Machine Detection (Positive Answering Machine Detection is available with PerfectCall Call Analysis only).

2.3. How to Enable PerfectCall Call Analysis

If you wish to use PerfectCall Call Analysis on a specified channel, perform the following steps. This procedure needs to be followed only once per channel in an application; thereafter, any outgoing calls made using **dx_dial()** will benefit from PerfectCall Call Analysis.

1. Make any desired modifications to the default dial tone, busy tone, fax tone, and ringback signal definitions. The **dx_chgfreq()**, **dx_chgdur()**, and **dx_chgrepcnt()** functions make these modifications.
2. Execute the **dx_initcallp()** function to activate PerfectCall Call Analysis. PerfectCall Call Analysis stays active until **dx_deltone()** is called.

NOTE: To disable PerfectCall Call Analysis, call the **dx_deltone()** function.

2.3.1. Modifying the Default Tone Definitions

PerfectCall Call Analysis makes use of Global Tone Detection (GTD) tone definitions for three different types of dial tones, two busy tones, one ringback tone, and two fax tones. The tone definitions specify the frequencies, durations, and repetition counts necessary to identify each of these signals. Each signal may consist of a single tone or a dual tone.

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

2. Call Analysis

- **dx_chgfreq()** specifies frequencies and tolerances for one or both frequencies of a single- or dual-frequency tone.
- **dx_chgdur()** specifies the cadence (on time, off time, and acceptable deviations) for a tone.
- **dx_chgrepent()** specifies the repetition count required to identify a tone.

These functions only change the tone definitions; they do not, by themselves, alter the behavior of PerfectCall Call Analysis. When the **dx_initcallp()** function is invoked to activate PerfectCall Call Analysis on a particular channel (see the following section), it uses the current tone definitions to initialize that channel. Multiple calls to **dx_initcallp()** may therefore use varying tone definitions, and several channels can operate simultaneously with different tone definitions.

Details on these functions may be found in the Function Reference chapter in the *Voice Software Reference Feature Guide for Linux*.

2.3.2. Activating PerfectCall Call Progress

PerfectCall Call Analysis is activated on a per-channel basis. For each channel on which PerfectCall Call Analysis is desired, the function **dx_initcallp()** must be called.

The **dx_initcallp()** function initializes PerfectCall Call Analysis on the specified channel, using the current tone definitions for local dial tone, international dial tone, extra dial tone, two busy signals, ringback, and two fax tones. Once the channel has been initialized with these tone definitions, this initialization cannot be altered. The only way to change the tone definitions in effect for a given channel is to issue a **dx_deltones()** call for that channel, then invoke another **dx_initcallp()** with different tone definitions.

Note that **dx_deltones()** deletes **all** Global Tone Detection definitions for the given channel, and not just those involved with PerfectCall Call Analysis.

Refer to the Function Reference chapter in the *Voice Programmer's Guide for Linux* for more information on **dx_initcallp()** and **dx_deltones()**.

2.4. How to Use Call Analysis

The following procedure describes how to initiate an outbound call with Call Analysis:

1. Set up the Call Analysis Parameter structure (`DX_CAP`), which contains parameters that control the operation of Call Analysis.
2. Execute the `dx_dial()` function to initiate Call Analysis.
 - **If running `dx_dial()` asynchronously**, use the Event Management functions to determine when dialing with Call Analysis is complete (`TDX_CALLP` termination event).
 - **If running `dx_dial()` synchronously**, wait for `dx_dial()` to return a value greater than 0 to indicate successful completion.

See the `dx_dial()` function description in the *Voice Programmer's Guide for Linux* for information about asynchronous and synchronous operation.

3. Use `ATDX_CPTERM()` to determine the outcome of the call:

an intercept	<code>CR_CEPT</code>
no ringback	<code>CR_NORB</code>
busy signal	<code>CR_BUSY</code>
no answer	<code>CR_NOANS</code>
fax machine	<code>CR_FAXTONE</code>
no dial tone	<code>CR_NODIALTONE</code>
connect	<code>CR_CNCT</code>
Call Analysis stopped	<code>CR_STOPD</code>
Call Analysis error	<code>CR_ERROR</code>

NOTE: When running `dx_dial()` synchronously, these results are also returned by `dx_dial()`.

4. Obtain additional termination, frequency, or cadence information (such as the length of the salutation) as desired using Extended Attribute functions.

Each of these steps is described in detail in the following pages.

NOTE: For information about the following, see the *Voice Programmer's Guide for Linux*:

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

2.4.1. Set Up the Call Analysis Parameter Structure (DX_CAP)

dx_dial(), which enables Call Analysis after dialing, uses the parameters in the DX_CAP structure. To use the default values for DX_CAP, specify NULL in the function.

If you want to customize the parameters for your environment, you must set up the Call Analysis Parameter structure before calling **dx_dial()**. By adjusting the DX_CAP parameters, you can:

- Eliminate Call Analysis functions that do not pertain to your environment.
- Optimize performance of the required functions.
- Support nonstandard system configurations.
- Perform additional functions, such as determining whether the called party is a business, residence, or answering machine.

To set up the DX_CAP structure for Call Analysis:

1. Execute the **dx_clrcap()** function to clear the DX_CAP and initialize the parameters to 0. The value 0 indicates that the default value will be used for that particular parameter. **dx_dial()** can also be set to run with default Call Analysis parameter values, by specifying a NULL pointer to the DX_CAP structure.
2. Set the parameter to another value if you do not want to use the default value for a given parameter.

For more detailed information on the DX_CAP block parameters, refer to *Section 2.5. How the DX_CAP Controls Call Analysis*

2.4.2. Use the dx_dial() Function to Initiate Call Analysis

Enable Call Analysis by calling **dx_dial()** with the **mode** function argument set to DX_CALLP. Termination of dialing with Call Analysis is indicated differently depending on whether the function is running asynchronously or synchronously. More information on **dx_dial()** can be found in the *Voice Programmer's Guide for Linux*.

2.4.3. Determine the Outcome of the Call

Once **dx_dial()** with Call Analysis has terminated, use the Extended Attribute function **ATDX_CPTERM()** to determine the outcome of the call. **ATDX_CPTERM()** will return one of the following Call Analysis Termination results:

CR_BUSY	Called line was busy.
CR_CNCT	Called line was connected.
CR_FAXTONE	Called line was answered by a fax machine or a modem (PerfectCall Call Analysis only).
CR_NOANS	Called line did not answer.
CR_NODIALTONE	Called line failed to produce a dial tone (PerfectCall Call Analysis only).
CR_NORB	Called line did not ring.
CR_CEPT	Called line received operator intercept (SIT). The Extended Attribute functions provide information on the detected frequencies and durations.
CR_STOPD	Call Analysis stopped due to dx_stopch() .
CR_ERROR	Call Analysis error occurred. ATDX_CPEERROR() returns the type of the Call Analysis error.

2. Call Analysis

Figure 3 illustrates the possible outcomes of Call Analysis. Figure 4 illustrates the possible outcomes of PerfectCall Call Analysis.

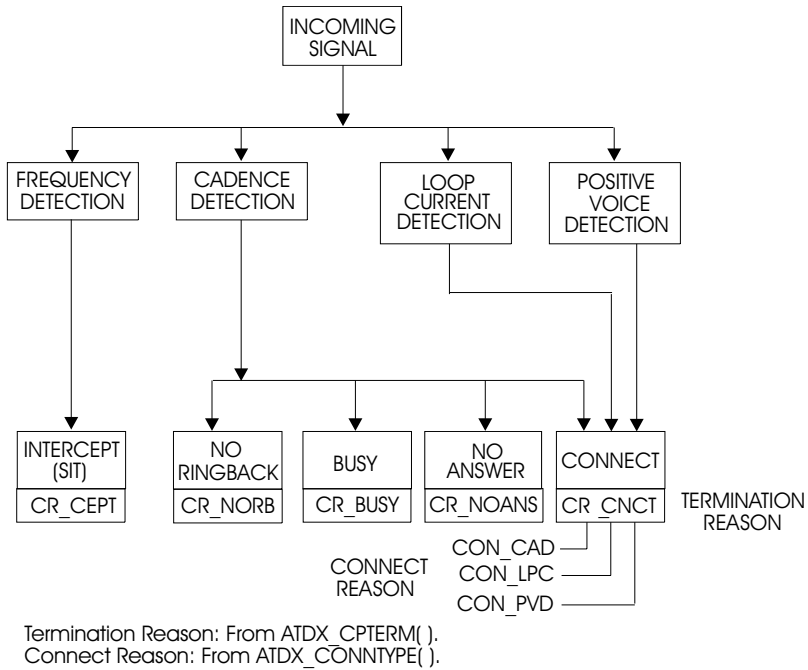


Figure 3. Call Analysis Outcomes for Basic Call Analysis

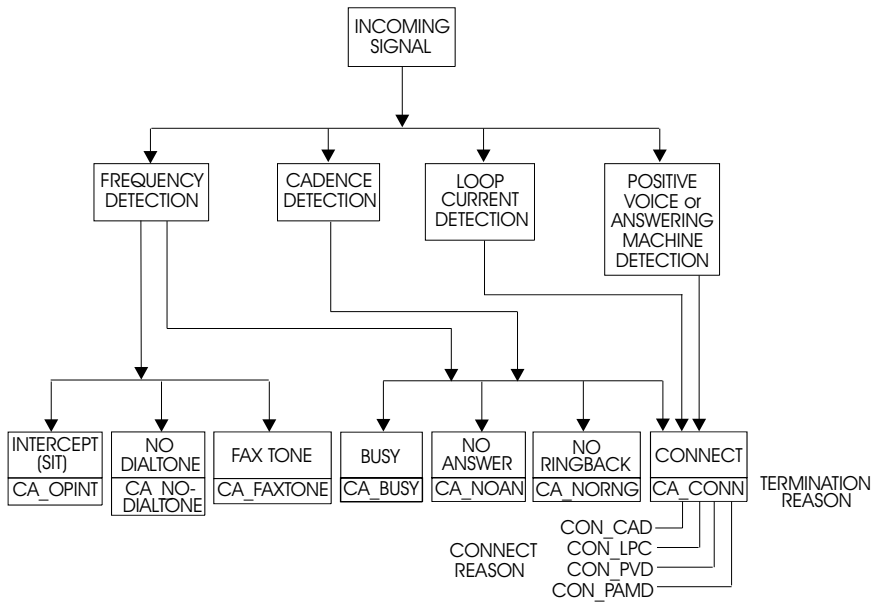


Figure 4. Call Analysis Outcomes for PerfectCall Call Analysis

NOTES: 1. Termination Reason: `ATDX_CPTERM()`

2. Connect Reason: `ATDX_CONNTYPE()`

2.4.4. Obtain Additional Call Outcome Information

Additional Call Analysis information can be retrieved using the following Extended Attribute functions:

ATDX_ANSRSIZ()	• Returns duration of answer
ATDX_CPEROR()	• Returns call analysis error
ATDX_CPTERM()	• Returns last call analysis termination
ATDX_CONNTYPE()	• Returns connection type
ATDX_CRTNID()	• Returns the identifier of the tone that caused the most recent Call Analysis termination
ATDX_DTNFAIL()	• Returns the dial tone character that indicates which dial tone Call Analysis failed to detect
ATDX_FRQDUR()	• Returns duration of first frequency detected
ATDX_FRQDUR2()	• Returns duration of second frequency detected
ATDX_FRQDUR3()	• Returns duration of third frequency detected
ATDX_FRQHZ()	• Returns frequency detected in Hz of first detected tone
ATDX_FRQHZ2()	• Returns frequency of second detected tone
ATDX_FRQHZ3()	• Returns frequency of third detected tone
ATDX_LONGLOW()	• Returns duration of longer silence
ATDX_FRQOUT()	• Returns percent of frequency out of bounds
ATDX_SHORTLO()	• Returns duration of shorter silence
ATDX_SIZEHI()	• Returns duration of non-silence

For a discussion of how frequency and cadence information returned by these Extended Attribute functions relate to the DX_CAP parameters, refer to *Section 2.5. How the DX_CAP Controls Call Analysis*.

2.5. How the DX_CAP Controls Call Analysis

The DX_CAP structure, as defined in the header file, is listed in the *Voice Programmer's Guide for Linux*.

The following sections describe the DX_CAP parameters that control Frequency Detection, Cadence Detection, Loop Current Detection, Positive Voice Detection, and Positive Answering Machine Detection.

2.5.1. Selecting SIT Frequency Detection, Positive Voice Detection, and Positive Answering Machine Detection

The Call Analysis Parameter structure (DX_CAP) parameter **ca_intflg** (intercept mode flag) is used to enable or disable Frequency Detection, Positive Voice Detection, and/or Positive Answering Machine Detection for Call Analysis.

The **ca_intflg** parameter also determines when SIT Frequency Detection should terminate and return an **intercept** immediately upon detection of the specified frequencies or wait for a connect indicated by Cadence Detection, Loop Current Detection, Positive Voice Detection, or Positive Answering Machine Detection.

The following defines are provided for use with the **ca_intflg** parameter:

2. Call Analysis

ca_intflg	Intercept Mode Flag: This parameter enables or disables SIT Frequency Detection, Positive Voice Detection (PVD), and/or Positive Answering Machine Detection (PAMD), and selects the mode of operation for Frequency Detection. Default: 1 (DX_OPTEN). The following modes are possible:
DX_OPTEN	Enable Frequency Detection (wait for a connect to terminate Call Analysis and return an intercept). If a valid frequency is detected, Call Analysis waits for detection of a connect using Cadence Detection or Loop Current Detection before returning an intercept .
DX_OPTDIS	Disable Frequency Detection and PVD.
DX_OPTNOCON	Enable Frequency Detection (terminate Call Analysis and return an intercept immediately upon detecting tones). Frequency Detection returns an intercept immediately after detecting a valid frequency.
DX_PVDENABLE	Enable PVD.
DX_PVDOPTEN	Enable PVD and DX_OPTEN.
DX_PVDOPTNOCON	Enable PVD and DX_OPTNOCON.
DX_PAMDENABLE	Enable PAMD.
DX_PAMDOPTEN	Enable PAMD and DX_OPTEN.

2.5.2. SIT Frequency Detection

SIT Frequency Detection operates simultaneously with all other Call Analysis detection methods. The purpose of Frequency Detection is to detect the tri-tone Special Information Tone (SIT) sequences and other single-frequency tones. Detection of a SIT sequence indicates an operator intercept or other problem in completing the call.

Voice Software Reference - Features Guide for Linux

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

Tri-Tone SIT Sequences

Table 2 provides tone information for the four SIT sequences. The frequencies are represented in Hz and the length of the signal is in 10 ms units.

Table 2. Special Information Tone Sequences

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

The length of the first tone is not dependable; often it is shortened or cut.

Setting Tri-Tone Frequency Detection Parameters

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

2. Call Analysis

To detect all four tri-tone SIT sequences:

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

NOTE: Occasionally, the first tone can also be truncated by a delay in the onset of Call Analysis due to the setting of `ca_stdely`.

- After an SIT sequence is detected, `ATDX_CPTERM()` will return `CR_CEPT` to indicate an operator intercept, and you can determine which SIT sequence was detected by obtaining the actual detected frequency and duration for the tri-tone sequence using Extended Attribute functions.

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

General

ca_stdely Start Delay: The delay after dialing has been completed and before starting Frequency Detection. This parameter also determines the start of Cadence Detection and Positive Voice Detection. Default: 25 (10 ms units). Note that this can affect detection of the first element of an operator intercept tone.

First Tone

- ca_lowerfrq** Lower Frequency: Lower bound for first tone in Hz. Default: 900.
- ca_upperfrq** Upper Frequency: Upper bound for first tone in Hz. Default: 1000. Adjust higher for additional operator intercept tones.
- ca_timefrq** Time Frequency: Minimum time for first tone to remain in bounds. The minimum amount of time required for the audio signal to remain within the frequency detection range for it to be detected. The audio signal must not be greater than **ca_upperfrq** or lower than **ca_lowerfrq** for at least the time interval specified in **ca_timefrq**. Default: 5 (10 ms units).
- ca_mxtimefrq** Maximum Time Frequency: Maximum allowable time for first tone to be present. Default: 0 (10 ms units).

Second Tone

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

- ca_lower2frq** Lower Bound for second Frequency: Lower bound for second tone in Hz. Default: 0.
- ca_upper2frq** Upper Bound for second Frequency: Upper bound for second tone in Hz. Default: 0.
- ca_time2frq** Time for second Frequency: Minimum time for second tone to remain in bounds. Default: 0 (10 ms units).
- ca_mxtime2frq** Maximum Time for second Frequency: Maximum allowable time for second tone to be present. Default: 0 (10 ms units).

Third Tone

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

ca_lower3frq	Lower Bound for third Frequency: Lower bound for third tone in Hz. Default: 0.
ca_upper3frq	Upper Bound for third Frequency: Upper bound for third tone in Hz. Default: 0.
ca_time3frq	Time for third Frequency: Minimum time for third tone to remain in bounds. Default: 0 (10 ms units).
ca_mxtime3frq	Maximum Time for third Frequency: Maximum allowable time for third tone to be present. Default: 0 (10 ms units).

Tri-Tone Frequency Information Returned by Extended Attribute Functions

Upon detection of the specified sequence of frequencies, Extended Attribute functions can be used to provide the exact frequency and duration of each tone in the sequence. The frequency and duration information will allow exact determination of all four SIT sequences.

The following Extended Attribute functions are used to provide information on the frequencies detected by Call Analysis.

First Tone (**ca_lowerfrq** and **ca_upperfrq**)

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

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

Second Tone (ca_lower2frq and ca_upper2frq)

ATDX_FRQHZ2() Frequency Hertz 2: Frequency in Hz of the tone detected in the tone detection range specified by the DX_CAP **ca_lower2frq** and **ca_upper2frq** parameters; usually the second tone of an SIT sequence.

ATDX_FRQDUR2() Frequency Duration 2: Duration of the tone detected in the tone detection range specified by the DX_CAP **ca_lower2frq** and **ca_upper2frq** parameters; usually the second tone of an SIT sequence (10 ms units).

Third Tone (ca_lower3frq and ca_upper3frq)

ATDX_FRQHZ3() Frequency Hertz 3: Frequency in Hz of the tone detected in the tone detection range specified by the DX_CAP **ca_lower3frq** and **ca_upper3frq** parameters; usually the third tone of an SIT sequence.

ATDX_FRQDUR3() Frequency Duration 3: Duration of the tone detected in the tone detection range specified by the DX_CAP **ca_lower3frq** and **ca_upper3frq** parameters; usually the third tone of an SIT sequence (10 ms units).

Global Tone Detection Tone Memory Usage

If you use Call Analysis to identify the tri-tone SIT sequences, Call Analysis will create tone detection templates internally, and this will reduce the number of tone templates that can be created using Global Tone Detection functions. See *Chapter 3. Global Tone Detection/Generation* for information relating to memory usage for Global Tone Detection.

2. Call Analysis

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

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

Frequency Detection Errors

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

Setting Single Tone Frequency Detection Parameters

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

The following fields in the DX_CAP are used for Frequency Detection. Frequencies are specified in hertz, and time is specified in 10 ms units.

ca_stdely	Start Delay: The delay after dialing has been completed and before starting Frequency Detection. This parameter also determines the start of Cadence Detection. Default: 25 (10 ms units).
ca_infltr	Not used.

Voice Software Reference - Features Guide for Linux

ca_lowerfrq	Lower Frequency: Lower bound for tone in Hz. Default: 900.
ca_upperfrq	Upper Frequency: Upper bound for tone in Hz. Default: 1000.
ca_timefrq	Time Frequency: Minimum time for tone to remain in bounds. The minimum amount of time required for the audio signal to remain within the frequency detection range for it to be detected. The audio signal must not be higher than ca_upperfrq or lower than ca_lowerfrq for at least the time interval specified in ca_timefrq , allowing for ca_rejctfrq . Default: 5 (10 ms units).
ca_rejctfrq	Allowable percentage of bad signal: Sets the percentage of time that the frequency can be out of bounds.

Single Tone Frequency Information Returned

Upon detection of a frequency in the specified range, the Extended Attribute functions can be used to provide the exact frequency that was detected.

The following Extended Attribute functions return information on the single tone detected in the tone detection range specified by the DX_CAP **ca_lowerfrq** and **ca_upperfrq** fields.

ATDX_FRQOUT()	• Not used.
ATDX_FRQHZ()	• Frequency Hertz: Frequency in Hz of the tone detected in the tone detection range specified by the DX_CAP ca_lowerfrq and ca_upperfrq parameters; usually the first tone of an SIT sequence.

2.5.3. Cadence Detection in Basic Call Analysis

The Cadence Detection algorithm has been optimized for use in the United States standard network environment.

2. Call Analysis

NOTE: This discussion of Cadence Detection and the Call Progress Characterization (CPC) utility is relevant to Basic Call Analysis only. For PerfectCall Call Analysis, refer to "Tone Detection in PerfectCall Call Analysis" below. However, The CPC program can be used to collect call progress data in both Basic Call Analysis and PerfectCall Call Analysis.

If your system is operating in another type of environment (such as behind a PBX), you can customize the Cadence Detection algorithm to suit your system through the adjustment of the Cadence Detection parameters. The Call Progress Characterization (CPC) utility, which runs under MS-DOS, can be used to determine the Cadence Detection parameter requirements for your system.

NOTE: For a detailed description of all the Cadence Detection parameters, and of the CPC program, refer to the following sections of the *Voice CPC Software Reference* in the *Voice Software Reference for MS-DOS*, Vol. II, part no. 05-0004:

Chapter 3—The Call Progress Characterization Program

Section 5.1—Cadence Detection

In the *Voice CPC Software Reference*, parameters are referred to by their root name (e.g. "**ca_stdely**" is referred to as "**stdely**").

The following section discusses Cadence Detection and some of the most commonly adjusted Cadence Detection parameters. An entire listing of the DX_CAP, including all Cadence Detection parameters, can be found in the *Voice Programmer's Guide for Linux*.

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

NOTES: 1. Sound is referred to as **nonsilence**.

NOTES: 2. The algorithm used for cadence detection is disclosed and protected under U.S. patent 4,477,698 of Melissa Electronic Labs, and other patents pending for Dialogic Corporation.

Typical Cadence Patterns

Figure 5, Figure 6, and Figure 7 show some typical cadence patterns.

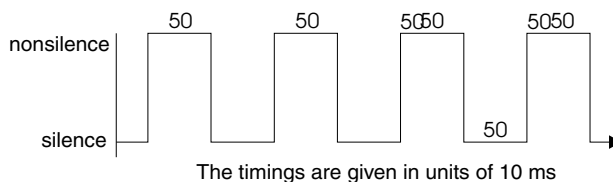


Figure 5. Standard Busy Signal

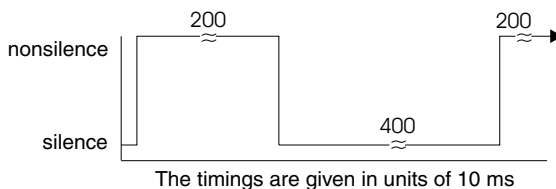


Figure 6. Standard Single Ring

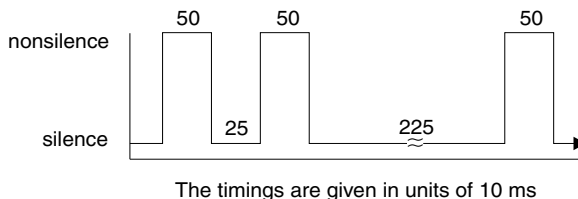


Figure 7. Type of Double Ring

Elements of a Cadence

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

two silence and two nonsilence periods in the audio signal. *Figure 8* illustrates cadence detection.

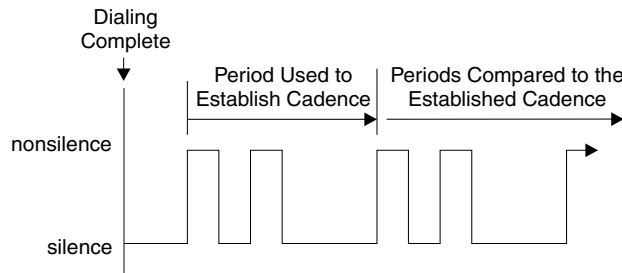


Figure 8. Cadence Detection

Once the cadence is established, the cadence values can be retrieved using the following Extended Attribute functions:

ATDX_SIZEHI()	• Length of the nonsilence period (in 10 ms units) for the detected cadence.
ATDX_SHORTLOW()	• Length of the shortest silence period for the detected cadence (in 10 ms units).
ATDX_LONGLOW()	• Length of the longest silence period for the detected cadence (in 10 ms units).

Only one nonsilence period is used to define the cadence because the nonsilence periods have the same duration.

Figure 9 shows the elements of an established cadence.

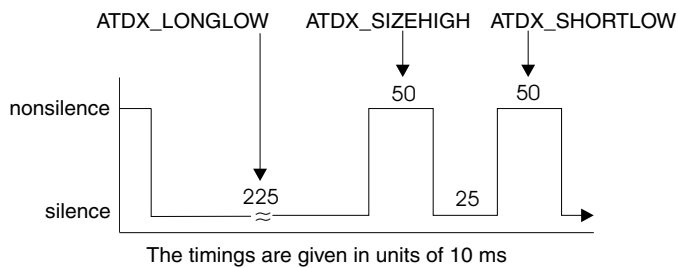


Figure 9. Elements of Established Cadence

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

Outcomes of Cadence Detection

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

- No Ringback
- Connect
- Busy
- No Answer

Although Loop Current Detection and Positive Voice Detection provide complementary means of detecting a connect, Cadence Detection provides the only way in Basic Call Analysis to detect a no ringback, busy, or no answer.

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

2. Call Analysis

- No Ringback** While the cadence is being established, Cadence Detection determines whether the signal is continuous silence or nonsilence. In this case, Cadence Detection returns a **no ringback**, indicating there is a problem in completing the call.
- Connect** While the cadence is being established, Cadence Detection determines whether the audio signal departs from acceptable network standards for busy or ring signals. In this case, Cadence Detection returns a **connect**, indicating that there was a “break” from general cadence standards.

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

- Connect** After the cadence has been established, Cadence Detection determines whether the audio signal departs from the established cadence. In this case, Cadence Detection returns a **connect**, indicating that there was a break in the established cadence.
- No Answer** After the cadence has been established, Cadence Detection determines whether the cadence belongs to a single or double ring. In this case, Cadence Detection can return a **no answer**, indicating there was no break in the ring cadence for a specified number of times.
- Busy** After the cadence has been established, Cadence Detection determines whether the cadence belongs to a slow busy signal. In this case, Cadence Detection can return a **busy**, indicating that the busy cadence was repeated for a specified number of times.

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

Setting Selected Cadence Detection Parameters

Only the most commonly adjusted Cadence Detection parameters are discussed here; an entire listing of the DX_CAP can be found in the *Voice Programmer's Guide for Linux*.

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

The Call Progress Characterization (CPC) utility, which runs under MS-DOS, can be used to determine the correct cadence detection parameter settings for your system.

NOTE: For a detailed description of all the Cadence Detection parameters, and of the CPC program, refer to the following sections of the *Voice CPC Software Reference* in the *Voice Software Reference for MS-DOS*, Vol. II, part no. 05-0004:

- *Chapter 3—The Call Progress Characterization Program*
- *Section 5.1—Cadence Detection*

In the *Voice CPC Software Reference*, parameters are referred to by their root name (e.g. "**ca_stdely**" is referred to as "**stdely**").

General Cadence Detection Parameters

ca_stdely Start Delay: The delay after dialing has been completed and before starting Cadence Detection. This parameter also determines the start of Frequency Detection and Positive Voice Detection.
Default: 25 (10 ms units) = 0.25 seconds.
Be careful with this variable. Setting this variable too small may allow switching transients or, if too long, miss critical signaling.

2. Call Analysis

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

ca_higlth High Glitch: The maximum nonsilence period to ignore. Used to help eliminate spurious nonsilence intervals. Default: 19 (in 10 ms units).

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

Cadence Detection Parameters Affecting a No Ringback

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

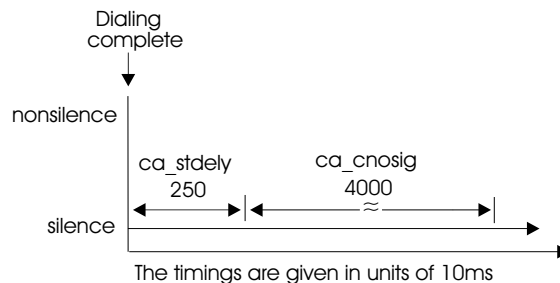


Figure 10. No Ringback Due to Continuous No Signal

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

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

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

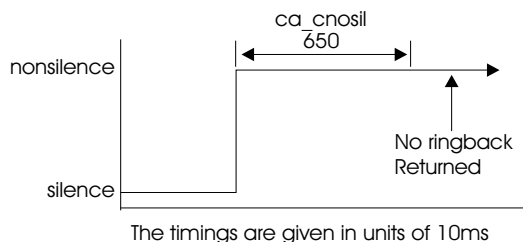


Figure 11. No Ringback Due to Continuous Nonsilence

Cadence Detection Parameters Affecting a No Answer or Busy

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

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

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

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

Cadence Detection Parameters Affecting a Connect

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

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

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

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

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

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

Voice Software Reference - Features Guide for Linux

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

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

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

ca_hedge Hello Edge: The point at which a connect will be returned to the application, either the rising edge (immediately when a connect is detected) or the falling edge (after the end of the salutation).
1 = rising edge. 2 = falling edge. Default: 2 (connect returned on falling edge of salutation). Try changing this if the called party has to say "Hello" twice to trigger the answer event.

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

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

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

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

Figure 12 shows how the **ca_ansrdgl** parameter works.

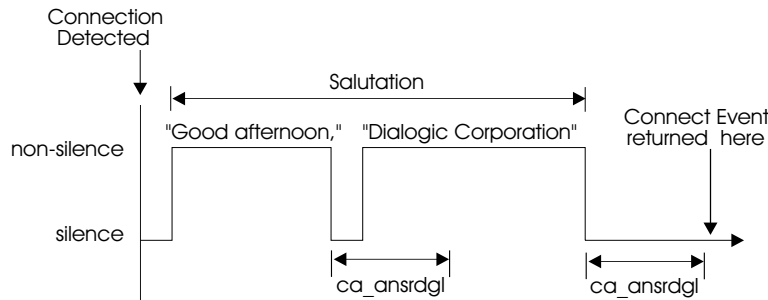


Figure 12. Cadence Detection Salutation Processing

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

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

NOTE: When a connect is detected through Positive Voice Detection or Loop Current Detection, the DX_CAP parameters **ca_hedge**, **ca_ansrdgl**, and **ca_maxansr** are ignored.

Cadence Information Returned Using Extended Attribute Functions

ATDX_SIZEHI()	• Size High: Duration of the cadence non-silence period (in 10 ms units).
ATDX_SHORTLOW()	• Short Low: Duration of the cadence shorter silence period (in 10 ms units).
ATDX_LONGLOW()	• Long Low: Duration of the cadence longer silence period (in 10 ms units).
ATDX_ANSRSIZ()	• Answer Size: Duration of answer if a connect occurred (in 10 ms units).
ATDX_CONNTYPE()	• Connection type. If ATDX_CONNTYPE() returns CON_CAD , the connect was due to Cadence Detection.

2.5.4. Tone Detection in PerfectCall Call Analysis

PerfectCall Call Analysis uses a combination of Cadence Detection and Frequency Detection to identify certain signals during the course of an outgoing call. Cadence Detection identifies repeating patterns of sound and silence, and Frequency Detection determines the pitch of the signal. Together, the cadence and frequency of a signal make up its “tone definition.”

Unlike Basic Call Analysis, which uses fields in the **DX_CAP** structure to store signal cadence information, PerfectCall Call Analysis uses tone definitions which are contained in the Voice Driver itself. Functions are available to modify these default tone definitions.

Types of Tones

Tone definitions are used to identify several kinds of signals. The following list shows the defined tones and their tone identifiers. Tone identifiers are returned by the **ATDX_CRTNID()** function.

- **TID_DIAL_LCL** Local dial tone

2. Call Analysis

- TID_DIAL_INTL International dial tone
- TID_DIAL_XTRA Special (or “extra”) dial tone
- TID_BUSY1 Single tone busy signal
- TID_BUSY2 Dual tone busy signal
- TID_RNGBK1 Ringback tone
- TID_FAX1 A fax CNG tone
- TID_FAX2 A fax CED or modem tone

The tone identifiers are used in calls to the functions **dx_chgfreq()**, **dx_chgdur()**, and **dx_chgrepent()** to change the tone definitions. Refer to these functions in the Function Reference chapter in the *Voice Programmer's Guide for Linux*, and to “How to Enable PerfectCall Call Analysis” (above), for details.

Dial Tone Detection

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

- L** Wait for a local dial tone
- I** Wait for an international dial tone
- X** Wait for a special (“extra”) dial tone

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

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

- ca_dtn_npres** Dial Tone Not Present: The length of time to wait before declaring the dial tone not present (in 10 ms units). If a dial tone of sufficient length (**ca_dtn_pres**) is not found within this period of time, Call Analysis terminates with the reason CR_NODIALTONE. The dial tone character (**L**, **I**, or **X**) for the missing dial tone can be obtained using **ATDX_DTNFAIL()**. Default value: 300 (three seconds).
- ca_dtn_deboff** Dial Tone Debounce: The maximum duration of a break in an otherwise continuous dial tone before it is considered invalid (in 10 ms units). This parameter is used for ignoring short drops in dial tone. If a drop longer than **ca_dtn_deboff** occurs, then dial tone is no longer considered present, and another dial tone must begin and be continuous for **ca_dtn_pres**. Default value: 10 (100 msec).

Ringback Detection

PerfectCall Call Analysis uses the tone definition for ringback to identify the first ringback signal of an outgoing call. At the end of the first ringback (that is, normally, at the beginning of the second ringback), a timer goes into effect. The system continues to identify ringback signals (but does not count them). If a break occurs in the ringback cadence, the call is assumed to have been answered, and Call Analysis terminates with the reason CR_CONN (connect); the connection type returned by the **ATDX_CONNTYPE()** function will be CON_CAD (cadence break).

However, if the timer expires before a connect is detected, then the call is deemed unanswered, and Call Analysis terminates with the reason CR_NOAN.

The following DX_CAP fields govern this behavior:

- ca_stdely** Start Delay: The delay after dialing has been completed before starting Cadence Detection, Frequency Detection, and Positive Voice Detection (in 10 ms units). Default: 25 (0.25 seconds).

2. Call Analysis

ca_cnosing	Continuous No Signal: The maximum length of silence (no signal) allowed immediately after the ca_stdely period (in 10 ms units). If this duration is exceeded, Call Analysis is terminated with the reason CR_NORNG (no ringback detected). Default value: 4000 (40 seconds).
ca_noanswer	No Answer: The length of time to wait after the first ringback before deciding that the call is not answered (in 10 ms units). If this duration is exceeded, Call Analysis is terminated with the reason CR_NOAN (no answer). Default value: 3000 (30 seconds).
ca_maxintering	Maximum Inter-ring: The maximum length of time to wait between consecutive ringback signals (in 10 ms units). If this duration is exceeded, Call Analysis is terminated with the reason CR_CONN (connected). Default value: 800 (8 seconds).

Busy Tone Detection

There are two busy tones defined in PerfectCall Call Analysis: TID_BUSY1 and TID_BUSY2. If either of them is detected while Frequency Detection and Cadence Detection are active, then Call Analysis is terminated with the reason CR_BUSY. **ATDX_CRTNID()** identifies which busy tone was detected.

No DX_CAP fields affect busy tone detection.

Positive Answering Machine Detection

Positive Answering Machine Detection (PAMD) is available only with PerfectCall Call Analysis. Whenever PAMD is enabled, Positive Voice Detection (PVD) is also enabled. PAMD is enabled by setting the **ca_intflg** field of the DX_CAP structure to one of the following values:

- **DX_PAMDENABLE**: Enable PAMD and PVD without enabling SIT Frequency Detection.
- **DX_PAMDOPTEN**: Enable PAMD and PVD, and enable SIT frequency detection.

Voice Software Reference - Features Guide for Linux

When enabled, detection of an answering machine will result in the termination of Call Analysis with the reason CR_CONN (connected); the connection type returned by the **ATDX_CONNTYPE()** function will be CON_PAMD.

To distinguish between a greeting by a live human and one by an answering machine, one of two methods is used:

- **PAMD_QUICK**: The quick method examines only the events surrounding the connect time and makes a rapid judgment as to whether or not an answering machine is involved.
- **PAMD_FULL**: The long method looks at the full greeting to determine whether it came from a human or a machine.

The slower method gives a very accurate determination; however, in situations where a fast decision is more important than accuracy, **PAMD_QUICK** might be preferred.

The following DX_CAP fields govern positive answering machine detection:

ca_pamd_spdval PAMD Speed Value: Whether to make a quick decision on Positive Answering Machine Detection. Possible values:

PAMD_FULL: Look at greeting (long method)

PAMD_QUICK: Look at connect only (quick method)

Default value: **PAMD_FULL**.

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

ca_pamd_failtime PAMD fail time: Maximum time to wait for Positive Answering Machine Detection or Positive Voice Detection after a cadence break. Default Value: 400 (in 10 ms units).

2. Call Analysis

ca_pamd_minring Minimum PAMD ring: Minimum allowable ring duration for Positive Answering Machine Detection. Default Value: 190 (in 10 ms units).

Fax or Modem Tone Detection

Detection of fax and modem tones is available only with PerfectCall Call Analysis. Two tones are defined: TID_FAX1 and TID_FAX2. If either of them is detected while Frequency Detection and Cadence Detection are active, then Call Analysis is terminated with the reason CR_FAXTONE. **ATDX_CRTNID()** identifies which fax or modem tone was detected.

No DX_CAP fields affect fax or modem tone detection.

2.5.5. Loop Current Detection

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

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

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

To disable Loop Current Detection, set **ca_lcdly** to -1.

Loop Current Detection Parameters Affecting a Connect

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

Voice Software Reference - Features Guide for Linux

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

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

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. Default: 10 (10 ms units).

If the **ATDX_CONNTYPE()** function returns **CON_LPC**, the connect was due to Loop Current Detection.

NOTE: When a connect is detected through Positive Voice Detection or Loop Current Detection, the **DX_CAP** parameters **ca_hedge**, **ca_ansrdgl**, and **ca_maxansr** are ignored.

2.5.6. Positive Voice Detection

Positive Voice Detection can detect when a call has been answered by determining whether an audio signal is present that has the characteristics of a live or recorded human voice. This provides a very precise method for identifying when a connect occurs.

PVD is especially useful in those situations where answer supervision is not available for Loop Current Detection to identify a connect, and where the cadence is not clearly broken for Cadence Detection to identify a connect (for example, when the nonsilence of the cadence is immediately followed by the nonsilence of speech).

If the **ATDX_CONNTYPE()** function returns **CON_PVD**, the connect was due to Positive Voice Detection.

NOTE: When a connect is detected through Positive Voice Detection or Loop Current Detection, the **DX_CAP** parameters **ca_hedge**, **ca_ansrdgl**, and **ca_maxansr** are ignored.

2.6. Call Analysis Errors

If `ATDX_CPTERM()` returns `CR_ERROR`, you can use `ATDX_CPELOR()` to determine the Call Analysis error that occurred.

<code>CR_MEMERR</code>	Out of memory trying to create temporary Special Information Tone (SIT) templates (exceeds maximum number of templates).
<code>CR_TMOUTON</code>	Time-out waiting for a SIT.
<code>CR_TMOUTOFF</code>	SIT too long (exceeds a <code>ca_mxtimefrq</code> parameter).
<code>CR_UNEXPTN</code>	Unexpected SIT (the sequence of detected tones did not correspond to the SIT sequence).
<code>CR_MXFRQERR</code>	Invalid <code>ca_mxtimefrq</code> field in <code>DX_CAP</code> . If the <code>ca_mxtimefrq</code> parameter for each SIT is nonzero, it must have a value greater than or equal to the <code>ca_timefrq</code> parameter for the same SIT.
<code>CR_UPFRQERR</code>	Invalid upper frequency selection. This value must be nonzero for detection of any SIT.
<code>CR_LGTUERR</code>	Lower frequency greater than upper frequency.
<code>CR_OVRLPERR</code>	Overlap in selected SITs.

3. Global Tone Detection/Generation

Global Tone Detection (GTD) and Global Tone Generation (GTG) provide the ability to generate and detect single- or dual-frequency tones. These features are available on all voice boards.

The functions associated with Global Tone Detection and Global Tone Generation are categorized in the *Voice Programmer's Guide for Linux*. The *Voice Programmer's Guide for Linux* also describes the Call Analysis functions that work with Global Tone Detection.

3.1. Global Tone Detection

Global Tone Detection (GTD) is a feature that allows a user to define the characteristics of a tone in order to detect a tone with the same characteristics. The characteristics of a tone are defined using GTD tone templates. The tone templates contain parameters that allow the user to assign frequency bounds and cadence components. Single- and dual-frequency tones are detected by comparing all incoming sounds to the GTD tone templates.

GTD operates on a channel-by-channel basis and is active when the channel is off-hook, unless the system uses a DTI/xxx board, in which case GTD is always active. GTD works simultaneously with DTMF and MF tone detection.

The driver responds to a detected tone by producing either a **tone event** on the event queue or a **digit** on the digit queue. The particular response depends upon the GTD tone configuration.

Use the Global Tone Detection functions to access tone templates and enable detection of single- and dual-frequency tones that fall outside those automatically provided with the Voice Driver. This includes tones outside the standard DTMF set of 0-9, a-d, *, and #, and the standard MF tones 0-9, *, and a-c.

3.1.1. Defining GTD Tones

GTD tones can have an associated ASCII digit (and digit type) specified using the **digit** and **digtype** parameters in the **dx_addtone()** function. When the tone is detected, the digit is placed in the DV_DIGIT buffer and can be retrieved using the **dx_getdig()** or **dx_getdigEx()** function. When the tone is detected, either the tone event or the digit associated with the tone can be used as a termination condition to terminate I/O functions.

Termination conditions are set using the DV_TPT data structure. See *Appendix A* in the *Voice Programmer's Guide for Linux* for information about the DV_TPT data structure.

NOTE: If you want to terminate on multiple tones (or digits), you need to specify the terminating conditions for each tone in a separate DV_TPT data structure.

3.1.2. Building Tone Templates

When creating the tone template you can define the following:

- single- or dual-frequency (300 - 3500 Hz)
- optional ASCII digit associated with the tone template
- cadence components

Adding a tone template to a channel enables detection of a tone on that channel. Although only one tone template can be created at a time, multiple tone templates can be added to a channel. Each channel can have a different set of tone templates. Once created, tone templates can be selectively enabled or disabled.

NOTE: A particular tone template cannot be changed or deleted. A tone template can be disabled on a channel, but to delete a tone template, all tone templates on that channel must be deleted.

The following functions are used to define tone templates:

- **dx_bldst()** build a single-frequency tone description
- **dx_blddt()** build a dual-frequency tone description

3. Global Tone Detection/Generation

- **dx_bldstcad()** build a single-frequency tone cadence description
- **dx_blddtcad()** build a dual-frequency tone cadence description
- **dx_setgtdamp()** set the GTD amplitude

NOTES: 1. GTD build functions define new tone templates, and **dx_addtone()** adds the tone templates to a channel.

NOTES: 2. Use **dx_addtone()** to enable detection of the tone template on a channel.

NOTES: 3. After building a tone template using a **dx_bld...()** function, **dx_addtone()** must be called to add this tone template to a channel. If the template is not added, the next call to a **dx_bld...()** function will overwrite the tone definition contained in the previous template.

dx_bldst() defines a simple single-frequency tone. Subsequent calls to **dx_addtone()** will use this tone until another tone is defined. Thus, you can build a tone template and add it to several different channels.

dx_blddt() defines a simple dual-frequency tone. Subsequent calls to **dx_addtone()** will use this tone until another tone is defined. Thus, you can build a tone template and add it to several different channels.

NOTE: The D/21D and D/41D boards may not be able to detect dual tones with frequency components closer than approximately 120 Hz as dual tones. The other Dialogic voice boards may not be able to detect dual tones with frequency components closer than approximately 60 Hz as dual tones. Use a single tone description to detect dual tones that are closer together than the limits specified above.

dx_bldstcad() defines a simple single-frequency cadence tone. Subsequent calls to **dx_addtone()** will use this tone until another tone is defined. Thus, you can build a tone template and add it to several different channels. A single-frequency cadence tone has single-frequency signals with specific on/off characteristics.

dx_blddtcad() defines a simple dual-frequency cadence tone. Subsequent calls to **dx_addtone()** will use this tone until another tone is defined. Thus, you can build a tone template and add it to several different channels. A dual-frequency cadence tone has dual-frequency signals with specific on/off characteristics.

The minimum on- and off-time for cadence detection is 40 ms on and 40 ms off.

dx_setgtdamp() sets the amplitudes used by GTD. The amplitudes set using **dx_setgtdamp()** will be the default amplitudes that will apply to all tones built using the **dx_bld...()** functions. The amplitudes will remain valid for all tones built until **dx_setgtdamp()** is called again and the amplitudes are changed.

Table 3 lists some standard Bell System Network Call Progress Tones. The frequencies are useful when creating the tone templates.

Table 3. Standard Bell System Network Call Progress Tones

Tone	Frequency(Hz)	On Time (ms)	Off Time (ms)
Dial	350 + 440	Continuous	
Busy	480 + 620	500	500
Congestion (Toll)	480 + 620	200	300
Reorder (Local)	480 + 620	300	200
Ringback	440 + 480	2000	4000

3.1.3. Working with Tone Templates

Use the following functions to add/delete tone templates or to enable/disable tone detection:

- **dx_addtone()** add a tone template
- **dx_deltone()** delete tone templates
- **dx_distone()** disable detection of a tone
- **dx_enbtone()** enable detection of a tone

dx_addtone() adds the tone template that was defined by the most recent GTD build-tone function call to the specified channel. Adding a tone template to a channel downloads it to the board and enables detection of tone-on and tone-off events for that tone template.

3. Global Tone Detection/Generation

Cautions

1. Each tone template must have a unique identification.
2. Errors will occur if you use **dx_addtone()** to change a tone template that has previously been added.

dx_deltone() removes all tone templates previously added to a channel with **dx_addtone()**. If no tone templates were previously enabled for this channel, the function has no effect.

NOTE: **dx_deltone()** does not affect tones defined by build-tone template functions and tone templates not yet defined. If you have added tones for PerfectCall Call Analysis, these tones are also deleted.

dx_distone() disables the detection of DX_TONEON and/or DX_TONEOFF events on a channel. Detection capability for user-defined tones is enabled on a channel by default when **dx_addtone()** is called.

dx_enbtone() enables the detection of DX_TONEON and/or DX_TONEOFF events on a channel. Detection capability for tones is enabled on a channel by default when **dx_addtone()** is called. The function can re-enable tones disabled by **dx_distone()**.

DX_TONEON and DX_TONEOFF events are Call Status Transition (CST) events.

3.1.4. Tone Event Retrieval

To retrieve events:

1. Call **dx_addtone()** or **dx_enbtone()** to enable tone-on/off detection.
2. Call **dx_getevt()** to wait for CST event(s). Events are returned in the DX_EBLK structure.

NOTE: These procedures are the same as the retrieval of any other CST event, except that **dx_addtone()** or **dx_enbtone()** are used to enable event detection instead of **dx_setevtmsk()**.

You can optionally specify an associated ASCII digit (and digit type) with the tone template. In this case, the tone template is treated like DTMF tones. When the digit is detected, it is placed in the digit buffer and can be used for termination. When an associated ASCII digit is specified, tone events will not be generated for that tone.

3.1.5. Memory Available for User-Defined Tone Templates

Guidelines for the maximum amount of memory available for user-defined tone templates on Dialogic voice and voice/fax boards are given in this section.

The number of voice channels available on each type of board is as follows:

- The 24-channel voice boards actually contain 8 voice channels on the baseboard and 16 voice channels on the daughterboard.
- The 30-channel voice boards actually contain 14 voice channels and 2 E-1 signaling channels on the baseboard and 16 voice channels on the daughterboard.
- The 32-channel voice board actually contains 16 voice channels on the baseboard and 16 voice channels on the daughterboard.
- The 48-channel voice board is a package of two 24-channel voice boards sharing a common backplane slot. It contains 8 voice channels on each of the two baseboards and 16 voice channels on each of the two daughterboards.
- The 60-channel voice board is a package of two 30-channel voice boards sharing a common backplane slot. It contains 14 voice channels and 2 E-1 signaling channels on each of the two baseboards and 16 voice channels on each of the two daughterboards.

The memory blocks on these boards are available only to the channels on the same board.

Table 4 gives the maximum amount of memory available for user-defined tone templates on Dialogic boards. *Table 5* shows the maximum memory available for tone templates for each tone-creating voice feature.

3. Global Tone Detection/Generation

Table 4. Maximum Memory Available for User-Defined Tone Templates on Dialogic Voice and Voice/Fax Boards

Hardware	Memory Blocks Available	
	Per Board	Per Channel
D/21D, D/21H	142	71
D/41D, D/41H, D/41ESC D/41EPCI D/42-NS DIALOG/4	142	35
D/80SC	284	35
D/160SC D/160SC-LS D/240PCI-T1 D/240SC D/240SC-T1 D/240SC-2T1 D/300PCI-E1 D/300PSC-E1 D/300SC-E1 D/300SC-2E1 D/320SC D/480SC-2T1 D/600SC-2E1 D/640SC	510 for each group of 16 channels	31
VFX/40ESC, VFX/40ESCplus	142	35

NOTE: The numbers in *Table 4* represent the number of memory blocks available to the user after all predefined tones and their indices have been allocated. Predefined tones include DTMFs.

A single memory block may hold either a single tone template or a set of indices.

Table 5. Maximum Memory Available for Tone Templates for Tone-Creating Voice Features

Feature	Memory Blocks Available per Channel
SIT: 1 Tone	2
SIT: 3 Tones	5
PerfectCall CPA	17
R2 MF	17
Socotel	19
User Defined	see notes below

Estimating Memory

Use the following procedure and guidelines to estimate the memory used for each tone on each channel.

- Calculate the total frequency range covered by the tone. For single tones, this is twice the deviation (unless the range is truncated by the GTD detection range); for dual tones, this is twice the deviation of **each** of the two tones minus any overlap (and minus any truncation by the GTD detection range).

For example:

- Single Tone: 400 Hz (125 Hz deviation) = bandwidth of 275 - 525 Hz, total of 250Hz.
- Dual Tone: 450 Hz (50 Hz deviation) and 1000 Hz (75 Hz deviation) = bandwidth of 400 - 500 Hz and 925 - 1075 Hz, total of 250Hz.
- Dual Tone: 450 Hz (100 Hz deviation) and 600 Hz (100 Hz deviation) = bandwidth of 350 - 550 Hz and 500 - 700 Hz; eliminating overlap, total range = 350 - 700 Hz, total of 350 Hz.

3. Global Tone Detection/Generation

Each tone costs, on average,

$1 + \text{round up} [1/16 * \text{round up} (\text{total frequency range} / 62.5)]$

This allows for:

- 1 memory block for the actual template
- 1/16 portion of a memory block for an index entry
- range/62.5 multiple indexing for speed

In practice, the 1/16 should be added across tones that have frequency overlap, rather than rounded up for each tone.

NOTE: The firmware will often use memory more efficiently than described by the guidelines given above. These guidelines estimate the maximum memory usage for user-defined tones; the actual usage may be lower.

Tone Issues

When creating user-defined tones or using one of the features that create GTD tones (*Table 4* and *Table 5*), keep the following issues in mind:

- If you use Call Progress Analysis to identify the tri-tone Special Information Tone (SIT) sequences, Call Progress Analysis will create GTD tones internally, and this will reduce the number of user-defined tones you can create. Call Progress Analysis creates one GTD tone for each single-frequency tone that you define in the Channel Parameter Block (CPB). For example, if detecting the SIT tri-tone sequences per channel (3 frequencies), the GTD memory will be reduced by five blocks.
- If you initiate Call Progress Analysis and there is not enough memory to create the SIT tones internally, you will get a T_CAERROR event and the **evtdata** field will contain MEMERR.
- Call Progress Analysis SIT detection releases GTD memory when Call Progress Analysis has completed. The other features do not release GTD memory until a **dl_deltone()** is performed.
- If you use **dl_initcallp()** to activate PerfectCall Call Progress Analysis to detect the different call progress signals (dial tone, busy tone, ringback tone, fax or modem tone), it creates GTD tones. This will reduce the number of user-defined tones you can create. PerfectCall Call Progress Analysis creates

8 GTD tones; this uses 17 memory blocks on each channel on which it is activated.

- If you activate PerfectCall Call Progress Analysis and there is not enough memory to create the GTD tones, you will get an E_MAXTMPLT error. This indicates that you are trying to exceed the maximum number of GTD tones.
- The **dl_deltones()** function deletes all user-defined tones from a specified channel and releases the memory that was used for those user-defined tones. When an associated ASCII digit is specified, tone events will not be generated for that tone.
- If you create R2 MF user-defined tones using **r2_creatfsig()**, the voice boards will usually be able to create all 15 R2 MF user-defined tones due to the overlap in frequencies for the R2 MF signals. If these boards do not have sufficient memory, they may be able to support R2 MF signaling through a reduced number of R2 MF user-defined tones.

If you use **dl_addtone()** or **r2_creatfsig()** to define a user-defined tone that exceeds the available memory, you will get an E_MAXTMPLT error.

- If you use **dx_blddt()** (or one of the **dx_bld...()** build-tone functions) or **r2_creatfsig()** to define a user-defined tone that alone or with existing user-defined tones exceeds the available memory, you will get an EDX_MAXTMPLT error.
- Dialogic voice boards support a full set of inbound or outbound R2 MF signals, or the complete Socotel signal set.
- The tone detection range should be limited to a maximum of 200 Hz per tone to reduce the chance of exceeding the available memory.

3.1.6. Applications

Two sample applications for Global Tone Detection are described in this section. The first application describes how to use GTD to detect a fast-busy signal to determine when a disconnect occurs. The second application describes how to use GTD for leading edge detection of a tone using debounce time.

3. Global Tone Detection/Generation

Disconnect Supervision

Global Tone Detection can be used for disconnect supervision. When a telephone call terminates, the central office may send a momentary drop in loop current to signal the disconnect. In configurations where the voice board is connected to a Private Branch Exchange (PBX), it is likely that there will be no drop in loop current for the voice board to detect. Instead, the PBX may initiate a fast-busy signal to indicate the disconnect. Global Tone Detection can be used to detect this fast-busy signal. Perform the following to detect the signal:

1. Determine the frequencies of the signal.
2. Characterize the on/off durations and tolerances of the signal cadence.
3. Use a build-tone function to define the characteristics of a single or dual tone with cadence in a tone template.
4. Use the **dx_addtone()** function to add the GTD tone template for Global Tone Detection on each channel.

Leading Edge Detection Using Debounce Time

Rather than detecting a signal immediately, an application may want to wait for a period of time (debounce time) before the **DX_TONEON** event is generated indicating the detection of the signal. The **dx_bldstcad()** and **dx_blddtcad()** functions can detect leading edge debounce on-time. A tone must be present at a given frequency and for a period of time (debounce time) before a **DX_TONEON** event is generated. The debounce time is specified using the **ontime** and **ondev** parameters in the **dx_bldstcad()** or **dx_blddtcad()** functions.

To use this application specify the following values for the **dx_bldstcad()** or **dx_blddtcad()** function parameters listed below:

ontime	1/2 of the desired debounce time
ondev	-1/2 of the desired debounce time
offtime	0
offdev	0
repnt	0

NOTE: This application cannot work with the functions **dx_blddt()** and **dx_bldst()** since these functions do not have timing field parameters.

3.2. Global Tone Generation (GTG)

Global Tone Generation enables the creation of user-defined tones. The Tone Generation template, **TN_GEN**, is used to define the tones with the following information:

- Single or dual tone
- Frequency fields
- Amplitude for each frequency
- Duration of tone

3.2.1. Global Tone Generation Functions

The following functions are used to generate tones:

- **dx_bldtngen()** build a tone generation template
- **dx_playtone()** play a tone

dx_bldtngen() is a convenience function that sets up the tone generation template data structure (**TN_GEN**) by allowing the assignment of specified values to the appropriate fields. The tone generation template is placed in the user's return buffer and can then be used by the **dx_playtone()** function to generate the tone.

dx_playtone() plays a tone specified by the tone generation template (pointed to by **tngenp**). Termination conditions are set using the **DV_TPT** structure. The reason for termination is returned by the **ATDX_TERMMSK()** function. **dx_playtone()** returns a 0 to indicate that it has completed successfully.

3.2.2. Building and Implementing a Tone Generation Template

The tone generation template defines the frequency, amplitude, and duration of a single- or dual-frequency tone to be played. You can use the convenience function **dx_bldtngen()** to set up the structure. Use **dx_playtone()** to play the tone.

3. Global Tone Detection/Generation

The TN_GEN data structure is:

```
typedef struct {
    unsigned short tg_dflag;      /* dual tone - 1, single tone - 0 */
    unsigned short tg_freq1;     /* frequency of tone 1 (in Hz) */
    unsigned short tg_freq2;     /* frequency of tone 2 (in Hz) */
    short int      tg_ampl1;     /* amplitude of tone 1 (in dB) */
    short int      tg_ampl2;     /* amplitude of tone 2 (in dB) */
    short int      tg_dur;       /* duration (in 10 ms) */
} TN_GEN;
```

After you build the TN_GEN data structure, there are two ways to define each tone template:

1. Include the values in the structure, or
2. Pass the values to TN_GEN using the **dx_bldtngen()** function.

After defining the template, pass TN_GEN to **dx_playtone()** to play the tone.

If you include the values in the structure, you must create a structure for each tone template. If you pass the values using the **dx_playtone()** function, then you can reuse the structure. If you are only changing one value in a template with many variables, it may be more convenient to use several structures in the code instead of reusing just one.

4. R2 MF Signaling

R2 MF signaling is an international signaling system that is used in Europe and Asia to permit the transmission of numerical and other information relating to the called and calling subscribers' lines. R2 MF signaling support is available on all Dialogic voice boards.

R2 MF signals that control the call setup are referred to as “interregister signals”. In the case of the signals sent between the central office (CO) and the customer premises equipment (CPE), the CO is referred to as the “outgoing register” and the CPE as the “incoming register”. Signals sent from the CO are “forward signals”; signals sent from the CPE are “backward signals”. The outgoing register (CO) sends forward interregister signals and receives backward interregister signals. The incoming register (CPE) receives forward interregister signals and sends backward interregister signals. See *Figure 13*.

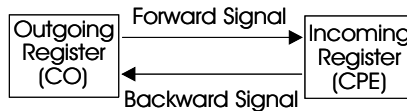


Figure 13. Forward and Backward Interregister Signals

The focus of this section is on the forward and backward interregister signals, and more specifically, the **address** signals, which can provide the telephone number of the called subscriber's line. For national traffic, the address signals can also provide the telephone number of a calling subscriber's line for automatic number identification (ANI) applications.

R2 MF signals that are used for supervisory signaling on the network are called **line signals**. Line signals are beyond the scope of this document.

4.1. Direct Dialing-In Service

Since R2 MF address signals can provide the telephone number of the called subscriber's line, the signals may be used by applications providing direct dialing-in (DDI) service, also known as direct inward dialing (DID), and dialed number identification service (DNIS).

DDI service allows an outside caller to dial an extension within a company without requiring an operator to transfer the call. The central office (CO) passes the last 2, 3, or 4 digits of the dialed number to the customer premises equipment (CPE) and the CPE completes the call.

4.2. R2 MF Multifrequency Combinations

R2 MF signaling uses a multifrequency code system based on six fundamental frequencies in the forward direction (1380, 1500, 1620, 1740, 1860, and 1980 Hz) and six frequencies in the backward direction (1140, 1020, 900, 780, 660, and 540 Hz).

Each signal is composed of two out of the six fundamental frequencies, which results in 15 different tone combinations in each direction. Although R2 MF is designed for operation on international networks with 15 multifrequency combinations in each direction, in national networks it can be used with a reduced number of signaling frequencies (e.g., 10 multifrequency combinations). See the following tables for lists of the signal tone pairs:

- *Table 6: Forward Signals*
- *Table 7: Backward Signals*

4. R2 MF Signaling

Table 6. Forward Signals, CCITT Signaling System R2 MF tones

R2 MF TONES			DIALOGIC INFORMATION		
Tone Number	Tone Pair Frequencies (Hz)		Group I Define	Group II Define	Tone Detect. ID
1	1380	1500	SIGI_1	SIGII_1	101
2	1380	1620	SIGI_2	SIGII_2	102
3	1500	1620	SIGI_3	SIGII_3	103
4	1380	1740	SIGI_4	SIGII_4	104
5	1500	1740	SIGI_5	SIGII_5	105
6	1620	1740	SIGI_6	SIGII_6	106
7	1380	1860	SIGI_7	SIGII_7	107
8	1500	1860	SIGI_8	SIGII_8	108
9	1620	1860	SIGI_9	SIGII_9	109
10	1740	1860	SIGI_0	SIGII_0	110
11	1380	1980	SIGI_11	SIGII_11	111
12	1500	1980	SIGI_12	SIGII_12	112
13	1620	1980	SIGI_13	SIGII_13	113
14	1740	1980	SIGI_14	SIGII_14	114
15	1860	1980	SIGI_15	SIGII_15	115

Table 7. Backward Signals, CCITT Signaling System R2 MF tones

R2 MF TONES			DIALOGIC INFORMATION	
Tone Number	Tone Pair Frequencies (Hz)		Group A Define	Group B Define
1	1140	1020	SIGA_1	SIGB_1
2	1140	900	SIGA_2	SIGB_2
3	1020	900	SIGA_3	SIGB_3
4	1140	780	SIGA_4	SIGB_4
5	1020	780	SIGA_5	SIGB_5
6	900	780	SIGA_6	SIGB_6
7	1140	660	SIGA_7	SIGB_7
8	1020	660	SIGA_8	SIGB_8
9	900	660	SIGA_9	SIGB_9
10	780	660	SIGA_0	SIGB_0
11	1140	540	SIGA_11	SIGB_11
12	1020	540	SIGA_12	SIGB_12
13	900	540	SIGA_13	SIGB_13
14	780	540	SIGA_14	SIGB_14
15	660	540	SIGA_15	SIGB_15

4.3. R2 MF Signal Meanings

There are two groups of meanings associated with each set of signals. Group I meanings and Group II meanings are associated with the 15 forward signals. Group A meanings and Group B meanings are associated with the 15 backward signals, as shown in *Figure 14*.

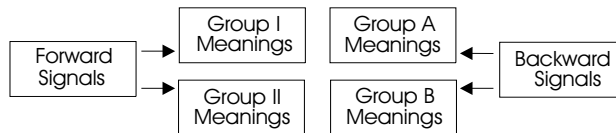


Figure 14. Multiple Meanings for R2 MF Signals

In general, Group I forward signals and Group A backward signals are used to control the call setup and to transfer address information between the outgoing register (CO) and the incoming register (CPE). The incoming register can then signal to the outgoing register to change over to the Group II and Group B meanings.

Group II forward signals provide the calling party's category, and Group B backward signals provide the condition of the called subscriber's line. For further information, see *Table 8* describing the purpose of the signal groups and the changeover in meanings.

Signaling must always begin with a Group I forward signal followed by a Group A backward signal that serves to acknowledge the signal just received and also has its own meaning. Each signal then requires a response from the other party. Each response becomes an acknowledgment of the event and an event for the other party to respond to.

Backward signals serve to indicate certain conditions encountered during call setup or to announce switch-over to changed meanings of subsequent backward signals. Changeover to Group II and Group B meanings allows information about the state of the called subscriber's line to be transferred.

Table 8. Purpose of Signal Groups and Changeover in Meaning

Signal	Purpose
Group I	Group I signals control the call set-up and provide address information.
Group A	<p>Group A signals acknowledge Group I signals (see exception under signal A-5 below) for call set-up, and can also request address and other information. Group A signals also control the changeover to Group II and Group B meanings through the following signals:</p> <p>A-3 Address Complete - Changeover to Reception of Group B Signals: Indicates the address is complete and signals a changeover to Group II/B meanings; after signal A-3 is sent, signaling cannot change back to Group I/A meanings.</p> <p>A-5 Send Calling Party's Category: Requests transmission of a single Group II signal providing the calling party's category. Signal A-5 requests a Group II signal but does not indicate changeover to Group B signals. When the Group II signal requested by A-5 is received, it is acknowledged by a Group A signal; this is an exception to the rule that Group A signals acknowledge Group I signals.</p>
Group II	Group II signals acknowledge signal A-3 or A-5 and provide the calling party category (national or international call, operator or subscriber, data transmission, maintenance or test call).
Group B	Group B signals acknowledge Group II signals and provide the condition of the called subscriber's line. Before Group B signals can be transmitted, the preceding backward signal must have been A-3. Signals cannot change back to Group I/A.

The Incoming Register Backward Signals Can Request:

- Transmission of address
 - Send next digit
 - Send digit previous to last digit sent
 - Send second digit previous to last digit sent
 - Send third digit previous to last digit sent
- Category of the call (the nature and origin)
 - National or international call
 - Operator or subscriber
 - Data transmission
 - Maintenance or test call
- Whether or not the circuit includes a satellite link
- Country code and language for international calls
- Information on use of an echo suppressor

The Incoming Register Backward Signals Can Indicate:

- Address complete - send category of call
- Address complete - put call through
- International, national, or local congestion
- Condition of subscriber's line
 - Send SIT to indicate long-term unavailability
 - Line busy
 - Unallocated number
 - Line free - charge on answer
 - Line free - no charge on answer (only for special destinations)

Voice Software Reference - Features Guide for Linux

- Line out of order

NOTE: The meaning of certain forward multifrequency combinations may also vary depending upon their position in the signaling sequence. For example, with terminal calls the first forward signal transmitted in international working is a language or discriminating digit (signals I-1 through I-10). When the same signal is sent as other than the first signal, it usually means a numerical digit.

See *Table 9* through *Table 12* for the signal meanings:

- *Table 9:* Meanings for R2 MF Group I Forward Signals
- *Table 10:* Meanings for R2 MF Group II Forward Signals
- *Table 11:* Meanings for R2 MF Group A Backward Signals
- *Table 12:* Meanings for R2 MF Group B Backward Signals

Table 9. Meanings for R2 MF Group I Forward Signals

Tone Number	Dialogic Define	(A) Primary Meaning (B) Secondary Meaning
1	SIGI_1	(A) Digit 1 (B) Language digit-French
2	SIGI_2	(A) Digit 2 (B) Language digit-English
3	SIGI_3	(A) Digit 3 (B) Language digit-German
4	SIGI_4	(A) Digit 4 (B) Language digit-Russian
5	SIGI_5	(A) Digit 5 (B) Language digit-Spanish
6	SIGI_6	(A) Digit 6 (B) Spare (language digit)
7	SIGI_7	(A) Digit 7 (B) Spare (language digit)
8	SIGI_8	(A) Digit 8 (B) Spare (language digit)
9	SIGI_9	(A) Digit 9 (B) Spare (discriminating digit)
10	SIGI_0	(A) Digit 0 (B) Discriminating digit
11	SIGI_11	(A) Access to incoming operator (Code 11) (B) Country code indicator: outgoing half-echo suppressor required
12	SIGI_12	(A) Access to delay operator (code 12); request not accepted (B) Country code indicator: no echo suppressor required
13	SIGI_13	(A) Access to test equipment (code 13); satellite link not included (B) Test call indicator
14	SIGI_14	(A) Incoming half-echo suppressor required; satellite link included (B) Country code indicator: outgoing half-echo suppressor inserted
15	SIGI_15	(A) End of pulsing (code 15); end of identification (B) Signal not used

Table 10. Meanings for R2 MF Group II Forward Signals

Tone Number	Dialogic Define	Meaning
1	SIGII_1	National: Subscriber without priority
2	SIGII_2	National: Subscriber with priority
3	SIGII_3	National: Maintenance equipment
4	SIGII_4	National: Spare
5	SIGII_5	National: Operator
6	SIGII_6	National: Data transmission
7	SIGII_7	International: Subscriber, operator, or maintenance equipment (without forward transfer)
8	SIGII_8	International: Data transmission
9	SIGII_9	International: Subscriber with priority
10	SIGII_0	International: Operator with forward transfer facility
11	SIGII_11	Spare for national use
12	SIGII_12	Spare for national use
13	SIGII_13	Spare for national use
14	SIGII_14	Spare for national use
15	SIGII_15	Spare for national use

Table 11. Meanings for R2 MF Group A Backward Signals

Tone Number	Dialogic Define	Meaning
1	SIGA_1	Send next digit (n+1)
2	SIGA_2	Send last but one digit (n-1)
3	SIGA_3	Address complete; change to Group B signals; no change back
4	SIGA_4	Congestion in the national network
5	SIGA_5	Send calling party's category; change to Group II; can change back
6	SIGA_6	Address complete; charge; set up speech conditions
7	SIGA_7	Send last but two digit (n-2)
8	SIGA_8	Send last but three digit (n-3)
9	SIGA_9	Spare for national use
10	SIGA_0	Spare for national use
11	SIGA_11	Send country code indicator
12	SIGA_12	Send language or discriminating digit
13	SIGA_13	Send nature of circuit (satellite link only)
14	SIGA_14	Request for information on use of an echo suppressor (Is an incoming half-echo suppressor required?)
15	SIGA_15	Congestion in an international exchange or at its output

Table 12. Meanings for R2 MF Group B Backward Signals

Tone Number	Dialogic Define	Meaning
1	SIGB_1	Spare for national use
2	SIGB_2	Send special information tone to indicate long-term unavailability
3	SIGB_3	Subscriber line busy
4	SIGB_4	Congestion encountered after change to Group B
5	SIGB_5	Unallocated number
6	SIGB_6	Subscriber line free; charge on answer
7	SIGB_7	Subscriber line free; no charge (only for calls to special destinations)
8	SIGB_8	Subscriber line out of order
9	SIGB_9	Spare for national use
10	SIGB_0	Spare for national use
11	SIGB_11	Spare for national use
12	SIGB_12	Spare for national use
13	SIGB_13	Spare for national use
14	SIGB_14	Spare for national use
15	SIGB_15	Spare for national use

4.4. R2 MF Compelled Signaling

R2 MF interregister signaling uses forward and backward compelled signaling. Simply put, with compelled signaling each signal is sent until it is responded to by a return signal, which in turn is sent until responded to by the other party. Each signal stays on until the other party responds, thus compelling a response from the other party.

Reliability and speed requirements for signaling systems are often in conflict: the faster the signaling, the more unreliable it is likely to be. Compelled signaling provides a balance between speed and reliability because it adapts its signaling speed to the working conditions with a minimum loss of reliability.

The R2 MF signal is composed of two significant events: tone-on and tone-off. Each tone event requires a response from the other party. Each response becomes an acknowledgment of the event and an event for the other party to respond to.

Compelled signaling must always begin with a Group I forward signal.

- The CO starts to send the first forward signal.
- As soon as the CPE recognizes the signal, it starts to send a backward signal that serves as an acknowledgment and at the same time has its own meaning.
- As soon as the CO recognizes the CPE acknowledging signal, it stops sending the forward signal.
- As soon as the CPE recognizes the end of the forward signal, it stops sending the backward signal.
- As soon as the CO recognizes the CPE end of the backward signal, it may start to send the next forward signal.

The CPE responds to a tone-on with a tone-on and to a tone-off with a tone-off. The CO responds to a tone-on with a tone-off and to a tone-off with a tone-on. Refer to *Figure 15* and *Figure 16* for more information:

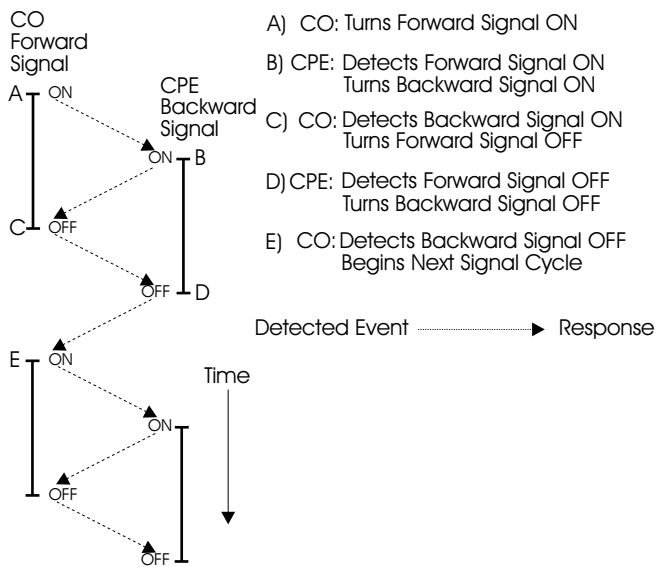


Figure 15. R2 MF Compelled Signaling Cycle

4. R2 MF Signaling

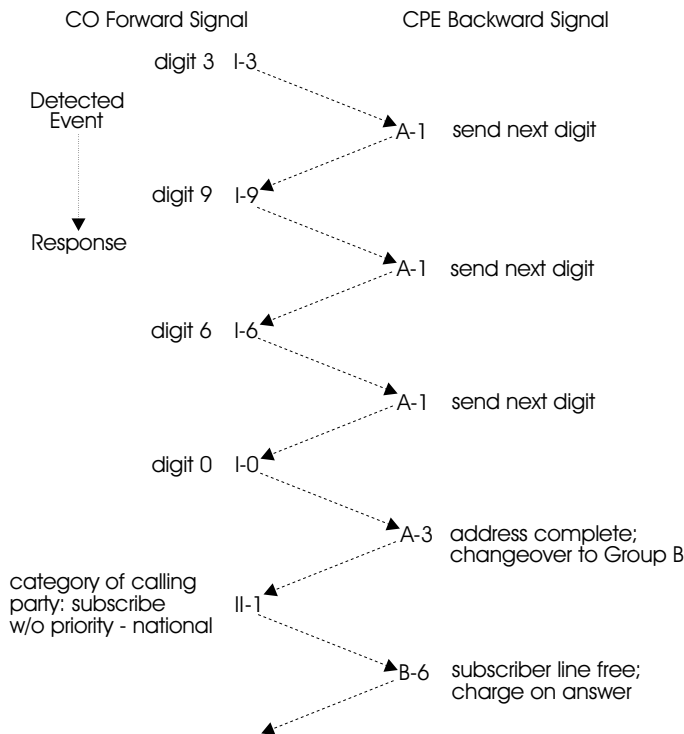


Figure 16. Example of R2 MF Signals for 4-digit DDI Application

4.5. Using R2 MF Signaling with Voice Boards

The Voice Software support for R2 MF signaling is based upon Global Tone Detection and Global Tone Generation.

The following R2 MF functions allow you to define R2 MF tones for detecting the forward signals and to play the backward signals in the correct timing sequence required by the compelled signaling procedure:

- **r2_creatfsig()**: Create R2 MF Forward Signal Tone
- **r2_playbsig()**: Play R2 MF Backward Signal

Voice Software Reference - Features Guide for Linux

See the *Voice Programmer's Guide for Linux* for a detailed description of these functions.

Four sets of defines are provided to specify the 15 Group I and 15 Group II forward signals, and the 15 Group A and 15 Group B backward signals. *Table 9, Table 10, Table 11, and Table 12* in *Section 4.3. R2 MF Signal Meanings* provide a list of these defines.

To implement R2 MF signaling, the board must have sufficient memory blocks to create the number of user-defined tones required by your application. Your application may not need to detect all 15 forward signals, especially if you do not need to support R2 MF signaling for international calls. If that is the case, your application can work with a reduced number of R2 MF tones.

DIALOG/HD boards normally contain sufficient memory to create the necessary R2 MF tones. However, you should be aware of the maximum number of user-defined tones (including non-R2 MF tones) allowed on the board. Refer to *Section 3.1.5. Memory Available for User-Defined Tone Templates* for more information.

The D/21D and D/41D boards may also be able to create all 15 R2 MF tones due to the overlap in frequencies for the R2 MF signals. If creating these tones exceeds the maximum number of tones allowed, you may be able to support R2 MF signaling through a reduced number of R2 MF user-defined tones. Refer to *Section 3.1.5. Memory Available for User-Defined Tone Templates*.

4.6. Related Publications

For more information on R2 MF signaling, you can refer to the following publications:

Specifications of Signaling Systems R1 and R2, International Telegraph and Telephone Consultative Committee (CCITT), Blue Book Volume VI, Fascicle VI.4, ISBN 92-61-03481-0

General Recommendations on Telephone Switching and Signaling, International Telegraph and Telephone Consultative Committee (CCITT), Blue Book Volume VI, Fascicle VI.1, ISBN 92-61-03451-9

5. Analog Display Services Interface

The Analog Display Services Interface (ADSI) protocol can be used to transmit data to a display-based telephone connected to an analog loop-start line. The telephone must be a true ADSI-compliant device (check with the telephone manufacturer).

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

The chapter contains a summary of the ADSI protocol followed by a description of how to implement ADSI support using Dialogic functions.

5.1. ADSI Protocol

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

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

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

After the data is transmitted, the ADSI telephone sends an acknowledgment consisting of a DTMF “d” plus a digit from 1 to 5 indicating the number of

messages in the transmission that the ADSI telephone received and understood. By obtaining this message count and comparing it with the number of messages transmitted, you can check for errors and retransmit any messages not received. (If you send 4 messages and the telephone receives 2, you must resend messages 3 and 4.)

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

The following section describes how to implement these operations.

5.2. Dialogic ADSI Support

Each time a new call is initiated on a channel, send the alert tone to alert the telephone that ADSI data will be transferred.

The ADSI alert tone can be sent and acknowledged, and ADSI data can be transferred using the **dx_setparm()** and **dx_play()** or **dx_playf()** functions. This is accomplished by setting the voice channel parameter **DXCH_DTINITSET** to **DM_A** in the **dx_setparm()** function and executing the **dx_play()** or **dx_playf()** function with the **PM_ADSIALERT** define ORed in the **mode** parameter.

If the acknowledgment digit is not received from the telephone within 500 ms following the end of the alert tone, the function will return a 0 but the termination mask returned by **ATDX_TERMMSK()** will be **TM_MAXTIME** to indicate an ADSI protocol error.

NOTE: The function will return a -1 if a failure is due to a general play error.

If the handshaking and transmission are successful, the function terminates normally with a **TM_EOD** (End of data reached on playback) termination mask returned by **ATDX_TERMMSK()** to indicate that the operation is complete.

5. Analog Display Services Interface

To transfer ADSI data without an alert tone, use the **dx_clrdigbuf()** or **dx_getdig()** function to ensure that there are no pending digits. Transfer ADSI data using the **dx_play()** or **dx_playf()** function with the PM_ADSI define ORed in the **mode** parameter.

If the transmission is successful, the function terminates normally with a TM_EOD (End of data reached on playback) termination mask returned by **ATDX_TERMMSK()** to indicate that the operation is complete.

The application is responsible for determining whether the message count acknowledgment matches the number of messages that were transmitted and for retransmitting any messages. Use the **dx_getdig()** function with DV_TPT **tp_termno** set to DX_DIGTYPE to receive the DTMF string 'adx' where 'x' is the message count acknowledgment digit (1 - 5).

The following defines are provided for use with the **mode** parameter:

- mode** defines the play mode and asynchronous/synchronous mode. One or more of the play mode parameters listed below may be selected in the bit mask for play mode combinations.
- PM_ADSIALERT: Play using the ADSI protocol with an alert tone preceding play. If ADSI protocol mode is selected, it is not necessary to select any other play mode parameters. PM_ADSIALERT should be ORed with the EV_SYNC or EV_ASYNC parameter in the **mode** parameter.
- PM_ADSI: Play using the ADSI protocol without an alert tone preceding play. If ADSI protocol mode is selected, it is not necessary to select any other play mode parameters. If ADSI data will be transferred, PM_ADSI should be ORed with the EV_SYNC or EV_ASYNC parameter in the **mode** parameter.

NOTE: If PM_ADSI play mode is selected, the ADSI protocol will be used to transfer ADSI data and it is not necessary to select any other play mode

Voice Software Reference - Features Guide for Linux

parameters. PM_ADSI should be ORed with the EV_SYNC or EV_ASYNC parameter in the mode parameter.

Example code for defining and playing an alert tone, receiving acknowledgment of the alert tone, and transferring ADSI data is shown in the accompanying example.

Example

Defining and playing an alert tone, receiving acknowledgment of alert tone, and using dx_play() to transfer ADSI data:

```
#include <stdio.h>
#include <srllib.h>
#include <fcntl.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 channe[12];
    parm = SR_POLLMODE;
    sr_setparm(SRL_DEVICE, SR_MODEID, &parm);
    /*
     * Open the channel using the command line arguments as input
     */
    sprintf(channe, "%sC%s", argv[1],argv[2]);
    if (( chfd = dx_open(channe, NULL)) == -1) {
        printf("Board open failed on device %s\n",channe);
        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_clrldigbuf( chfd );
            printf("Digit buffer cleared ..\n");
        }
    }
}
```

5. Analog Display Services Interface

```
/*
 * 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_ADSSALERT|EV_SYNC) ==-1) {
    printf("dx_play on the ADSI file failed\n");
    exit(1);
}
}

dx_close(chfd);
exit(0);
}
```

5.3. Related Publications

The ADSI data must conform to interface requirements described in *Bellcore Technical Reference TR-NWT-000030, Voiceband Data Transmission Interface Generic Requirements*.

For information about message requirements (how the data should be displayed on the Customer Premise Equipment), see *Bellcore Technical Reference TR-NWT-001273, Generic Requirements for an SPCS to Customer Premises Equipment Data Interface for Analog Display Services*.

These technical references can be obtained from Bellcore by calling 1-800-521-CORE.

6. Speed and Volume Control

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

NOTE: Speed can be controlled on playbacks using 24 kbps or 32 kbps ADPCM only. Volume can be controlled on all playbacks regardless of the encoding algorithm.

6.1. Speed and Volume Convenience Functions

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

- **dx_addspddig()** - adds a digit that will modify speed by a specified amount.
- **dx_addvoldig()** - adds a digit that will modify volume by a specified amount.

6.2. Speed and Volume Adjustment Functions

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

- **dx_setsvcond()** - sets conditions that adjust speed or volume. Use this if you want to adjust speed or volume in response to a DTMF digit, or start of play.
- **dx_adjsv()** - adjusts speed or volume explicitly. Use this if your adjustment condition is not a digit or start of play. For example, the

application could call this function after detecting a spoken word (voice recognition) or a certain key on the keyboard.

See the *Voice Programmer's Guide for Linux* for detailed information about these functions.

6.3. Speed and Volume Modification Tables

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

The table has 21 entries. Twenty of these allow for a maximum of 10 increases and decreases in speed or volume; the entry in the middle of the table is referred to as the "origin" entry that represents normal speed or volume. The normal speed or volume is how playback occurs when the Speed and Volume Control feature is not used.

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

Although the origin is typically set to normal speed/volume, changing the setting of the origin does not affect the other settings, because all values in the SVMT are based on a deviation from normal speed/volume.

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

A speed/volume adjustment stays in effect until the next adjustment on that channel or until a system reset.

The SVMT is like a 21-speed bicycle. You can select the gear ratio for each of the 21 speeds before you go for a ride (by changing the values in the SVMT).

6. Speed and Volume Control

And you can select any gear once you are on the bike, like adjusting the current speed/volume to any setting in the SVMT.

The Speed or Volume Modification Table can be set or reset using the **dx_setsvmt()** function which uses the DX_SVMT data structure. The current values of these tables can also be returned to the DX_SVMT structure using **dx_getsvmt()**. The DX_SVCB data structure uses this table when setting adjustment conditions. See the *Voice Programmer's Guide for Linux* for information about the DX_SVMT and DX_SVCB data structures.

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

- a specified level (i.e., to a specified absolute position in the speed table or volume table)
- a change in level (i.e., by a specified number of steps up or down in the speed table or volume table)

The Speed Modification Table is shown in *Table 13*. Each entry in the table is a percentage deviation from the default play speed ("origin"). For example, the decrease[6] position reduces speed by 40%. This is four steps from the origin.

Table 13. Speed Modification Table

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

6. Speed and Volume Control

NOTE: The total speed modification range is from -50% to +50%. In this table, the lowest position actually utilized is the decrease[5] position. The remaining decrease fields are set to -128 (80h). If these "nonadjustment" positions are selected, the default action is to play at the decrease[5] speed.

These fields can be reset, as long as no values lower than -50 are used (that is, you could spread the 50% speed decrease over 10 steps rather than 5). Similarly, you could spread the 50% speed increase over 10 steps rather than 5.

The Volume Modification Table is shown in *Table 14*. Each entry in the table is a deviation in dB from the starting point or volume ("origin"). By default, each entry in the Volume Modification Table is equivalent to 2 decibels from the origin. Volume can be decreased by 2 decibels by specifying position 1 in the table, or by moving 1 step down. For example, the increase[1] position increases volume by 4 dB. This is two steps from the origin.

NOTE: The total volume modification range is from -30 dB to +10 dB. In this table, the highest position utilized is the increase[4] position. The remaining increase fields are set to -128 (80h). If these "nonadjustment" positions are selected, the default action is to play at the increase[4] volume.

These fields can be reset, as long as no values higher than +10 are used (that is, you could spread the 10 dB volume increase over 10 steps rather than 5).

Table 14. Volume Modification Table

Table Entry	Default Value (dB)	Absolute Position
decrease[0]	-30	-10
decrease[1]	-18	-9
decrease[2]	-16	-8
decrease[3]	-14	-7
decrease[4]	-12	-6
decrease[5]	-10	-5
decrease[6]	-08	-4
decrease[7]	-06	-3
decrease[8]	-04	-2
decrease[9]	-02	-1
origin	0	0
increase[0]	+02	1
increase[1]	+04	2
increase[2]	+06	3
increase[3]	+08	4
increase[4]	+10	5
increase[5]	-128 (80h)	6
.	.	.
.	.	.
.	.	.
increase[9]	-128 (80h)	10

6.4. Play Adjustment Digits

The Voice Software processes play adjustment digits differently from normal digits:

- If a play adjustment digit is entered during playback, it causes a play adjustment only and has no other effect. This means that the digit is not added to the digit queue, it cannot be retrieved with the `dx_getdig()` or `dx_getdigEx()` function, and it does not affect any termination condition.
- If the digit queue contains adjustment digits when a play begins and play adjustment is set to be level sensitive, the digits will affect the speed or volume and then be removed from the queue.

6.5. Setting Play Adjustment Conditions

Adjustment conditions are set in the same way for speed or volume. The following steps describe how to set conditions upon which volume should be adjusted:

1. Set up the Volume Modification Table (if you do not want to use the defaults):
 - Set up the `DX_SVMT` structure to specify the size and number of the steps in the table.
 - Call the `dx_setsvmt()` function, which points to the `DX_SVMT` structure, to modify the Volume Modification Table (`dx_setsvmt()` can also be used to reset the table to its default values).
2. Set up the `DX_SVCB` structure to specify the condition, the size, and the type of adjustment.
3. Call `dx_setsvcond()`, which points to an array of `DX_SVCB` structures. All subsequent plays will adjust volume as required whenever one of the conditions specified in the array occurs.

See the *Voice Programmer's Guide for Linux* for more information about `dx_setsvcond()`, `dx_setsvmt()`, and the `DX_SVMT` and `DX_SVCB` structures.

6.6. Explicitly Adjusting Speed and Volume

Speed and volume adjustments are made in the same way. The following is an example of the steps you should take to adjust speed, but you can use exactly the same procedure for volume:

1. Set up the Speed Modification Table (if you do not want to use the defaults):
 - Set up the `DX_SVMT` structure to specify the size and number of the steps in the table.
 - Call the `dx_setsvmt()` function, which points to the `DX_SVMT` structure, to modify the Speed Modification Table (`dx_setsvmt()` can also be used to reset the table to its default values).
2. When required, call `dx_adjsv()` to adjust the Speed Modification Table by specifying the size and type of the adjustment.

See the *Voice Programmer's Guide for Linux* for more information about `dx_adjsv()`, `dx_setsvmt()`, and the `DX_SVMT` structure.

7. Caller ID

The Caller ID feature provides information that can include the calling party's Directory Number (DN), the date and time of the call, and the calling party's subscriber name. The functions associated with Caller ID are categorized and described in the *Voice Programmer's Guide for Linux*.

7.1. Overview

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

NOTE: If Caller ID is enabled, on-hook detection (DTMF, MF, and Global Tone Detection) will not function.

7.2. Supported Formats

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

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

7.3. Related References

Before developing an application that requires Caller ID information, contact your service provider and request the following specifications:

Bellcore Documents for CLASS SDM and MDM :

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

Information Exchange Management

Bellcore

445 South St., Room 2J-125

P. O. Box 1910

Morristown, NJ 07962-1910

Phone: 973-829-4785

7.4. Theory of Operation

Caller Identification (Caller ID) is a service provided by local telephone companies to enable the subscriber to receive the caller's phone number (Directory Number) and other information about the call. The Caller ID information is transmitted using FSK (Frequency Shift Keying) to the subscriber from the service provider (telephone company Central Office) at 1200 baud.

Caller ID information is received from the Central Office (CO) between the first and second ring. This information is supported as sent by the service provider in the format types described in *Table 15*.

Table 15. Supported Caller ID Information

Caller ID Information	CLASS	
	SDM	MDM
Frame header (indicating SDM or MDM format type)	X	X
Calling line's Directory Number (DN)	X	X
Date	X	X
Time	X	X
Calling line's subscriber name		X
Calling line's DN (digits only)		X
Dialed number		X
Reason why caller DN is not available		X
Reason why calling subscriber name is not available		X
Indicate if the call is forwarded		X
Indicate if the call is "long distance"		X

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

7.5. Accessing Caller ID Information

Applications using the Caller ID feature receive Caller ID information from the service provider between the first and second ring. The ring event in the application must be set to occur on or after the second ring. The ring event will indicate the reception of the Caller ID information sent from the CO.

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

Voice Software Reference - Features Guide for Linux

- answered calls (application channel goes offhook): Caller ID information is available until the call is disconnected (application channel goes onhook)
NOTE: If the call is answered **before** the Caller ID information has been received from the CO, Caller ID information will not be available to the application.
- unanswered calls (application channel remains onhook): Caller ID information is available until rings are no longer received from the CO (signaled by ring event, if enabled)
NOTE: If the application remains onhook 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.

Voice API functions and parameters are included in this reference to provide access to Caller ID information received from the CO. The Caller ID functions are as follows:

- **dx_gtcallid()** - returns the calling line Directory Number (DN). Issue this function for applications that require only the calling line DN.
- **dx_gtextcallid()** - returns the requested Caller ID message. Issue this function for each type of caller ID message required.
- **dx_wtcallid()** - waits for rings and returns the calling station's DN. This convenience function combines the functionality of the **dx_setevtmsk()** and **dx_getevt()** voice functions, and the **dx_gtcallid()** Caller ID function.

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

To determine if Caller ID information has been received from the CO, before issuing a **dx_gtcallid()** or **dx_gtextcallid()** Caller ID function, check the event data in the **dx_eblk** event block structure. When the ring event is received (set by the application as stated above), the event data field in the event block is bit mapped and indicates that Caller ID information is available when bit 0 (LSB) is set to 1 (see the function code examples in this document). For details on the **dx_eblk** event block structure, refer to the Data Structures and Devices Parameters chapter in the *Voice Programmer's Guide for Linux*.

7.6. Enabling Channels to Use the Caller ID Feature

During Dialogic Service startup, before the initial use of Caller ID functions, the application must enable the Caller ID feature on the channels requiring Caller ID. Caller ID is enabled by setting the following channel-based parameter using the Dialogic Voice library function `dx_setparm()`.

Caller ID parameter for `dx_setparm()`:

Parameter	Description
<code>DXCH_CALLID</code>	<p>Default: <code>DX_CALLIDDISABLE</code></p> <p>Enable/disable Caller ID for the channel as specified in <code>dx_setparm()</code>.</p> <p>Valid values:</p> <p><code>DX_CALLIDDISABLE</code></p> <p><code>DX_CALLIDENABLE</code></p>

NOTE: Refer to the *Voice Programmer's Guide for Linux* for more information on `dx_setparm()`.

7.7. Error Handling

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

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

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

Voice Software Reference - Features Guide for Linux

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

8. Global Dial Pulse Detection

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

- The Global DPD algorithm is adaptive and can train on any DPD digit it encounters, with the greatest accuracy produced from training on a digit that has five or more pulses. Global DPD does not require a leading “0” to train the Global DPD algorithm.
- Global DPD can be performed simultaneously with DTMF detection. The application can determine whether the digit detected is a DTMF or DPD digit.
- Global DPD can be performed simultaneously with Global Tone Detection (GTD). For example, the application can use GTD to monitor for disconnect tones (dial tone or busy) simultaneously with DPD.
- Global DPD supports pulse-digit cut-through during a voice playback, with the correct digit returned in the digit buffer. Global DPD uses echo cancellation, which provides more accurate reporting of digits during voice playback. Refer to *Chapter 11. Echo Cancellation* for information about the echo cancellation feature.

The following applications are supported by the Global DPD feature:

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

8.1. DPD Parameters

This implementation of Dial Pulse Detection is referred to as Global DPD because its detection algorithm supports a wide range of dial pulses - from 8 pulses-per-second (PPS) to 22 PPS telephones.

Diallogic has qualified its Dial Pulse Detection algorithm against dial pulse data collected from different parts of the world to develop customized Global DPD download parameters. In addition, a generic 10 PPS Global DPD parameter file is provided for countries that use 10 PPS phones.

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

8.2. Global DPD Demonstration Program

The GDPD demo resides in the */usr/diallogic/demos/dpd_demos* directory. This directory contains the necessary demo source code, necessary header files, and a makefile utility to demonstrate the answer features of the voice board. The demo source code can be used as an example, or can be compiled by invoking the makefile utility in the same directory. It is used in the same way as the *pansr* and *cbansr* demonstration programs, but can accept both dial pulses and DTMF tones. See 13.3. *Asynchronous Demo Programs pansr and cbansr* for instructions on running this demo.

8.3. Global DPD Application Programming Interface

Global DPD uses the **dx_setdigtyp()** function to enable DPD. Refer to the *Voice Programmer's Guide for Linux* for information on the **dx_setdigtyp()** function.

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

```
typedef struct dv_digit {
    char dg_value[DG_MAXDIGS + 1];
    char dg_type[DG_MAXDIGS + 1];
} DV_DIGIT;
```

where DG_MAXDIGS is equal to 31.

8. Global Dial Pulse Detection

You then use a pointer to the structure that contains a digit buffer. (See the Programming Procedure and Example below.) This method allows you to determine very quickly whether a pulse or DTMF telephone is being used.

If your application has been designed to work with a DPD/120 board, you may need to modify the application to work with the DPD enabled voice boards and the improved capabilities of Global DPD (see *Section 8.3.1. Programming Procedure and Example*).

8.3.1. Programming Procedure and Example

Retrieving Digits Using `dx_getdig()`

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

1. Define a data structure of type `DV_DIGIT` (the `DV_DIGIT` structure is defined by including the `dxxxlib.h` header file).
2. Since the supported voice boards come with channels capable of Global DPD, you must enable DPD on the desired channels using the `dx_setdigtyp()` function. Refer to the *Voice Programmer's Guide for Linux*.
3. For each new connection, use `dx_setdigtyp()` with the `DM_DPDZ` mask, which initializes the DPD algorithm. After collecting the first DPD digit string, the mask can be set to `DM_DPD` for the remainder of that connection. Each subsequent invocation of `dx_setdigtyp()` must use the `DM_DPD` mask.
4. Execute the `dx_getdig()` function to collect and transfer the digits to the user's digit buffer. The digits are stored in the `dg_value` field of the `DV_DIGIT` structure. The corresponding digit types are stored in the `dg_type` field of the `DV_DIGIT` structure. The following equates identify the digit types returned in the `dg_type` field:

<u>Digit Type</u>	<u>Equate</u>
Audio Pulse	DG_APD_ASCII
Dial Pulse	DG_DPD_ASCII

Digit Type	Equate
DTMF	DG_DTMF_ASCII
Loop Pulse	DG_LPD_ASCII
MF	DG_MF_ASCII
User digit type	DG_USER1
User digit type	DG_USER2
User digit type	DG_USER3
User digit type	DG_USER4
User digit type	DG_USER5

Retrieving Digits as Events

To get the digits as events, use the following asynchronous programming model using the **dx_setevtmask()**, **sr_waitevt()**, and **sr_getevtdatap()** functions and the **DX_CST** data structure. Refer to the *Voice Programmer's Guide for Linux*.

1. Since the supported voice boards come with channels capable of Global DPD, you must enable DPD on the desired channels using the **dx_setdigtyp()** function. Refer to the *Voice Programmer's Guide for Linux*.
2. For each new connection, use **dx_setdigtype()** with the **DM_DPDZ** mask, which initializes the DPD algorithm. After collecting the first DPD digit string, the mask can be set to **DM_DPD** for the remainder of that connection. Each subsequent invocation of **dx_setdigtype()** must use the **DM_DPD** mask.
3. Use **dx_setevtmask()** to enable digit detection.
4. Use **sr_waitevt()** to wait for events.
5. When a CST event occurs, use **sr_getevtdatap()** to retrieve the pointer to the **DX_CST** structure. The typedef for the **DX_CST** is:

```
typedef struct dx_cst {
    unsigned short cst_event;
    unsigned short cst_data;
} DX_CST;
```

8. Global Dial Pulse Detection

- The `cst_data` field for a `DE_DIGITS` event contains an ASCII digit (low byte) and the digit type (high byte). The following equates distinguish the digit types that are returned in the high byte of the `cst_data` field of the `DX_CST` structure:

Digit Type	Equate
Audio Pulse	DG_APD_ASCII
Dial Pulse	DG_DPD_ASCII
DTMF	DG_DTMF_ASCII
Loop Pulse	DG_LPD_ASCII
MF	DG_MF_ASCII
User digit type	DG_USER1
User digit type	DG_USER2
User digit type	DG_USER3
User digit type	DG_USER4
User digit type	DG_USER5

Programming Example (Synchronous Model)

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

typedef enum {
    False,
    True,
} Boolean;

DV_TPT
    TerminationParameterTable[2];

DV_DIGIT
    DigitBuffer;

Boolean
    GetDigits (int, int);

char
    DeviceName[] = "dxB1C1";
```

Voice Software Reference - Features Guide for Linux

```
void main (void) {
    int
        DeviceHandle;

    /* Start the Dialogic system */

    if ((DeviceHandle = dx_open(DeviceName, NULL)) == -1) {
        printf("ERROR: dx_open() failed for device \"%s\", errno=%d\n",
            DeviceName, errno);
        return;
    }

    /* Wait for 2 rings and go off hook. */
    if ((ReturnCode = dx_wtring(DeviceHandle, 2, DX_OFFHOOK, -1)) == -1) {
        printf("ERROR: dx_wtring() failed on channel %s, %s\n",
            ATDV_NAMEP(DeviceHandle), ATDV_ERRMSGP(DeviceHandle));
        return;
    }

    /* Get 8 digits on channel 1 */
    if (GetDigits(8, DeviceHandle)) {
        /*
         * Got the digits, do something with them.
         * Note:
         * In this simple example, the 1st digit
         * is not returned, it is used for training
         * and determining digit type.
         */
    }

    /* Stop the Dialogic system */
    dx_close(DeviceHandle);
}

/*
 * This is an example function which illustrates the DPD and digit type
 * features.
 *
 * 1) Set up to detect 1 digit (DPD or DTMF).
 * 2) If digit is DPD, set up to detect rest of digits using DPD detection.
 * 3) If digit is DTMF, set up to detect rest of digits using DTMF detection.
 */
Boolean GetDigits (int NumberOfDigits, int DeviceHandle) {
    int
        DigitsDetected = 0;
    long
        TerminationMask = 0;

    /*
     * Set up mask to detect DTMF or DPD (train on initial digit) digits.
     * Clear digit buffer.
     */
    if (dx_setdigtyp(DeviceHandle, DM_DPDZ | DM_DTMF) == -1) {
        printf("ERROR: dx_setdigtyp() failed on channel %s, %s\n",
            ATDV_NAMEP(DeviceHandle), ATDV_ERRMSGP(DeviceHandle));
        return(False);
    }
}
```

8. Global Dial Pulse Detection

```
/*
1) Set up the TPT to terminate on
   A) 1 digit detected
   or
   B) a timeout after 10 seconds
2) Use the DV_DIGIT data structure which
   provides a digit array as well as a digit type
   array.
*/

dx_clrtp (TerminationParameterTable, 2);
TerminationParameterTable[0].tp_type = IO_CONT;
TerminationParameterTable[0].tp_termno = DX_MAXDTMF;
TerminationParameterTable[0].t_length = 1;
TerminationParameterTable[0].tp_flags = TF_MAXDTMF;

TerminationParameterTable[1].tp_type = IO_EOT;
TerminationParameterTable[1].tp_termno = DX_MAXTIME;
TerminationParameterTable[1].t_length = 100;
TerminationParameterTable[1].tp_flags = TF_MAXTIME;

if (dx_clrdigbuf(DeviceHandle) == -1) {
    printf("ERROR: dx_clrdigbuf() failed on channel %s, %s\n",
           ATDV_NAMEP(DeviceHandle), ATDV_ERRMSGP(DeviceHandle));
    return(False);
}

if ((DigitsDetected = dx_getdig(DeviceHandle, TerminationParameterTable,
                                &DigitBuffer, EV_SYNC)) == -1) {
    printf("ERROR: dx_getdig() failed on channel %s, %s\n",
           ATDV_NAMEP(DeviceHandle), ATDV_ERRMSGP(DeviceHandle));
    return(False);
}

/*
   Check if the dx_getdig() function terminated on Max. Digits
   or if we timed out
*/
if ((TerminationMask = ATDX_TERMMSK(DeviceHandle)) == AT_FAILURE) {
    printf("ERROR: ATDX_TERMMSK() failed on channel %s, %s\n",
           ATDV_NAMEP(DeviceHandle), ATDV_ERRMSGP(DeviceHandle));
    return(False);
}

if (TerminationMask & TM_MAXTIME) {
    /* We timed out waiting for Digit */
    return(False);
}
```

Voice Software Reference - Features Guide for Linux

```
/*
    Determine the type of the digit detected.
    If it is DPD (DG_DPD_ASCII), set up to detect only DPD digits.
    If it is DTMF (DG_DTMF_ASCII), set up to detect only DTMF digits.
*/

switch (DigitBuffer.dg_type[0]) {
    case DG_DPD_ASCII:
        /* Set up mask to detect only DPD digits */
        if (dx_setdigtyp(DeviceHandle, DM_DPD) == -1) {
            printf("ERROR: dx_setdigtyp() failed on channel %s, %s\n",
                ATDV_NAMEP(DeviceHandle), ATDV_ERRMSGP(DeviceHandle));
            return(False);
        }
        break;
    case DG_DTMF_ASCII:
        /* Set up mask to detect only DTMF digits */
        if (dx_setdigtyp(DeviceHandle, DM_DTMF) == -1) {
            printf("ERROR: dx_setdigtyp() failed on channel %s, %s\n",
                ATDV_NAMEP(DeviceHandle), ATDV_ERRMSGP(DeviceHandle));
            return(False);
        }
        break;
    default:
        /* Unexpected or invalid digit type */
        return(False);
}

/*
    1) Set up the TPT to terminate on
        A) 'NumberOfDigits - 1' digits detected
        or
        B) Timeout after 30 seconds
    2) Use the DV_DIGIT data structure which
        provides a digit array as well as a digit type
        array.
*/
dx_clrtpt(TerminationParameterTable, 2);
TerminationParameterTable[0].tp_type = IO_CONT;
TerminationParameterTable[0].tp_termno = DX_MAXDTMF;
TerminationParameterTable[0].t_length = NumberOfDigits - 1;
TerminationParameterTable[0].tp_flags = TF_MAXDTMF;

TerminationParameterTable[1].tp_type = IO_EOT;
TerminationParameterTable[1].tp_termno = DX_MAXTIME;
TerminationParameterTable[1].t_length = 300;
TerminationParameterTable[1].tp_flags = TF_MAXTIME;

if (dx_clrdigbuf(DeviceHandle) == -1) {
    printf("ERROR: dx_clrdigbuf() failed on channel %s, %s\n",
        ATDV_NAMEP(DeviceHandle), ATDV_ERRMSGP(DeviceHandle));
    return(False);
}

if ((DigitsDetected = dx_getdig(DeviceHandle, TerminationParameterTable,
    &DigitBuffer, EV_SYNC)) == -1) {
    printf("ERROR: dx_getdig() failed on channel %s, %s\n",
        ATDV_NAMEP(DeviceHandle), ATDV_ERRMSGP(DeviceHandle));
    return(False);
}
}
```

8. Global Dial Pulse Detection

```
/*
   Check if we received all of the expected digits. The dx_getdig()
   function returns the total number of digits received + 1 for the
   '\0' character terminating the digit string.
*/
if (DigitsDetected == NumberOfDigits) {
    return(True);
} else {
    return(False);
}
}
```

8.3.2. Programming Considerations

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

- Talk-off rejection (the ability of the algorithm to distinguish between dial pulses and the human voice) will improve after the first digit is detected.
- Digit detection will be slightly more accurate (about 2%) after detecting a digit of 5 or greater. It is not necessary to dial a special training digit to do this. The application may simply restrict the first menu to digits 5, 6, 7, 8, 9, and 0, and the training will be complete. Subsequent menus may be unrestricted.
- In general, detection accuracy is greater for higher digits than for lower. While detection accuracy is very high, it may be further improved by restricting menu selections, whenever convenient, to digits greater than 3.

9. Transaction Record

Transaction Record enables the recording of a two-party conversation by allowing two SCbus time slots from a single channel to be recorded. This feature is useful for Call Center applications where it is necessary to archive a verbal transaction or record a live conversation. A live conversation requires two time slots on the SCbus, but Dialogic voice boards today can only record one time slot at a time. No loss of channel density is realized with this feature. For example, a D/160SC-LS can still record 16 simultaneous conversations. Voice activity on two channels can be summed and stored in a single file, or in a combination of files, devices, and/or memory.

NOTE: Transaction Record is not supported on D/41ESC, VFX/40ESC, and VFX/40ESCplus boards.

The following functions are used for the Transaction Record feature:

- **dx_recmm()** - records voice data from two channels to a data file, memory, or custom device
- **dx_recmmf()** - records voice data from two channels to a single file

See the *Voice Programmer's Guide for Linux* for descriptions of these functions.

10. Silence Compressed Record

The Silence Compressed Record feature (SCR) enables recording with silent pauses eliminated. This results in smaller recorded files with no loss of intelligibility. When the audio level is at or falls below the silence threshold for a minimum duration of time, SCR begins. When a short burst of noise (glitch) is detected, the compression does not end unless the glitch is longer than a specified period of time.

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

The SCR parameters specify the silence threshold, the duration of silence at the end of speech before silence compression begins, the duration of a glitch in the line which does not stop silence compression, and more. *Figure 17* illustrates how these parameters work. Refer to the *Software Installation Reference for Linux* for details of the parameters and information on how to enable and configure this feature.

The following encoding algorithms and sampling rates are supported in SCR:

- 6 KHz and 8 KHz OKI ADPCM
- 8 KHz and 11 KHz linear PCM
- 8 KHz and 11 KHz A-law PCM
- 8 KHz and 11 KHz Mu-law PCM

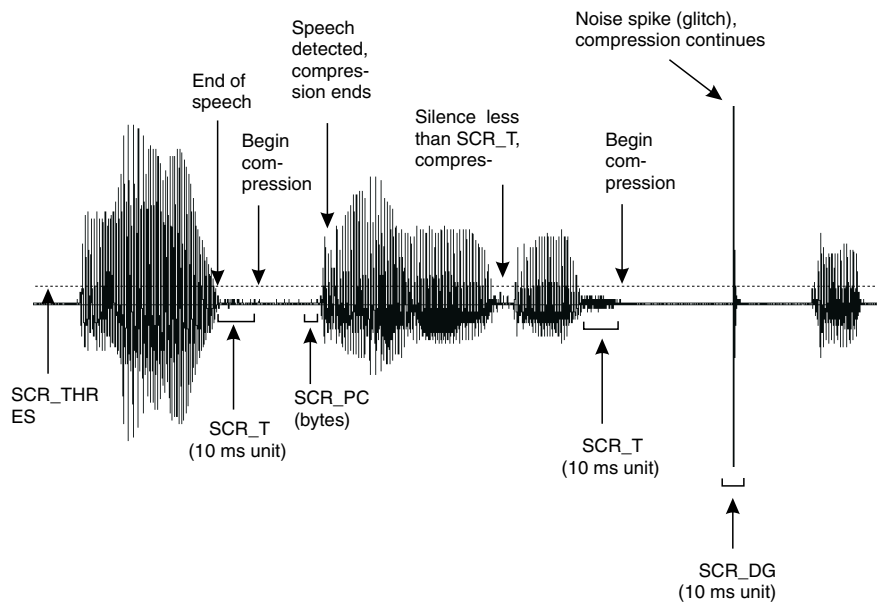


Figure 17. SCR Parameters

11. Echo Cancellation

The Dialogic Echo Cancellation Resource (ECR) feature is a functional component of a Dialogic voice channel. In ECR mode, the voice channel can dynamically perform echo cancellation on any external SCbus time slot signal.

The Dialogic ECR feature lets you use echo cancellation on signals external to the voice channel. The echo cancellation capability becomes a system-wide resource that may be applied to any SCbus PCM stream. The addition of the ECR feature allows the application to dynamically configure a voice channel as either an echo cancellation device (ECR mode) or as a standard voice processing channel (SVP mode). In ECR mode, a portion of the standard voice functionality remains available while another portion of it becomes unavailable.

NOTE: The ECR feature is supported on high-density boards (boards having 8 or more channels) only.

11.1. Overview

The Dialogic echo canceller accepts two SCbus input data streams. One stream contains data that is identical to that which was transmitted to the echo-producing circuit (Transmit Signal in *Figure 18*). The second stream, referred to as the echo-carrying stream, contains received data from this circuit. The received data typically contains a signal with two time-varying signals superimposed upon one another. One signal consists of a filtered version of the transmitted data (referred to as echo) and the other signal originates at the far end (referred to as “far-end speech”).

The purpose of the echo canceller is to sufficiently reduce the magnitude of the echo component, such that it does not interfere with further processing or analysis of the echo-canceled data stream. The echo canceller performs this function by computing a model of the impulse response of the echo path using information in the echo-carrying signal. Then, given the impulse response model and access to the echo reference signal, the echo canceller forms an estimate of the echo. This estimate is then subtracted from the echo-carrying signal, forming a third, echo-canceled signal.

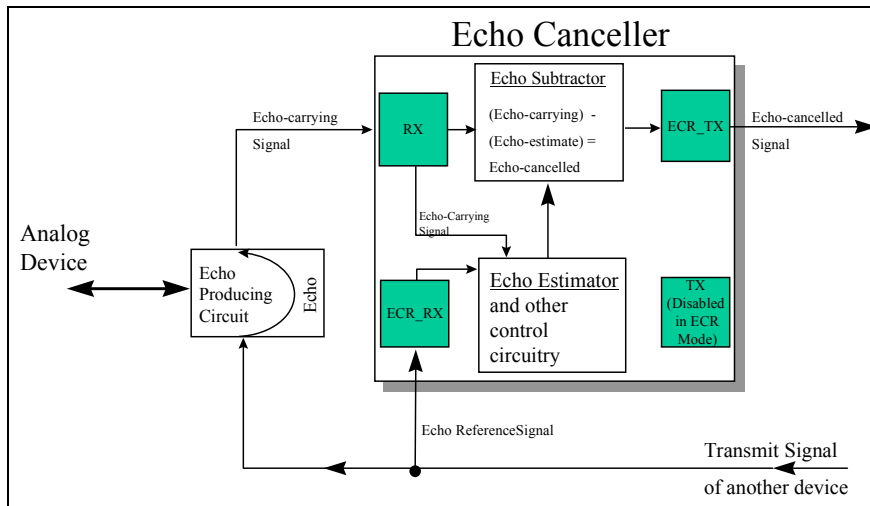


Figure 18. Echo Canceller with Relevant Input and Output Signals

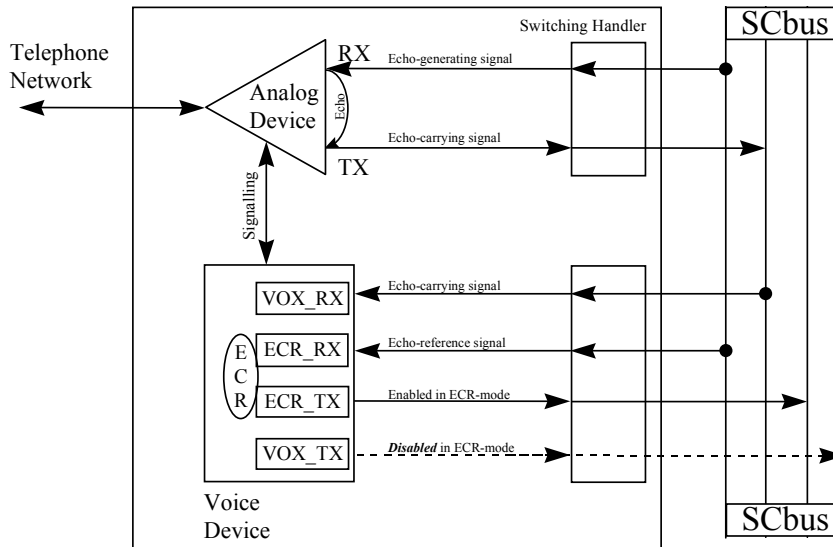


Figure 19. Echo Canceller Operating over an SCbus

11. Echo Cancellation

Figure 19 illustrates the signals used in the echo canceller. For echo cancellation, an extra SCbus time slot is assigned to each voice device for use by the ECR feature. To activate ECR mode, the application must route two receive time slots to the voice channel.

Once the ECR feature is enabled on a board, each voice channel is also permanently assigned two SCbus **transmit** time slots. These time slots are referred to as the voice-transmit time slot and the echo-cancellation transmit time slot. You can retrieve the time slot numbers for each via the **dx_getxmitslot()** and **dx_getxmitslotecr()** functions, respectively. If the ECR feature is not enabled, the channels are not assigned the echo cancellation SCbus transmit time slots, and ECR mode is not possible on any voice channel of that board.

The function **dx_listen()** routes the echo-carrying signal to the voice device. A call to **dx_listenecr()** or **dx_listenecrex()** routes the echo reference signal to the voice channel and simultaneously activates ECR mode. The resulting echo canceller uses the echo reference signal to estimate the echo component in the echo-carrying signal, and subtracts that estimate from the echo-carrying signal. This process results in an echo-canceled signal with a greatly reduced echo component.

For another device to receive the echo-canceled signal output by the echo canceller, it calls **dx_getxmitslotecr()** to retrieve the echo canceller's transmit time-slot number, and calls **xx_listen()** (where **xx_** is **ag_**, **dt_**, **dx_**, or **ms_**) to connect its receive channel to the echo-canceled signal.

To return the voice channel to Standard Voice Processing (SVP) mode, the application calls **dx_unlistenecr()** on the voice channel to stop the echo canceller, disable ECR mode, and disconnect the echo canceller's receive channel.

11.1.1. Modes of Operation

When the ECR feature is enabled via the Dialogic Configuration Manager (DCM) on a supported board, there are two possible modes of operation: SVP and ECR. Until ECR mode is activated, the board operates in the Standard Voice Processing (SVP) mode, which offers default echo cancellation. The ECR mode, which provides high-performance echo cancellation, can be dynamically activated or deactivated on any voice channel of the enabled board. Details about the two echo cancellation modes are provided below.

SVP Mode

All voice channels are initially in the SVP mode with the default echo cancellation for ECR feature-enabled boards. The SVP mode utilizes a 48 tap (6 ms) echo canceller. In SVP mode, all Dialogic voice functions operate as usual, with one exception. If a channel in SVP mode is playing a file and listening (via a **dx_listen()** function), then playback transmits data on both the standard voice-transmit time slot and the echo-cancellation transmit time slot. The standard voice-transmit time slot carries the play signal. The echo cancellation time slot carries an echo-canceled version of the signal from the receive time slot. This echo-canceled signal is derived from the original play signal (the echo reference) and the signal from the receive time slot specified in the **dx_listen()** function (the echo carrying signal).

ECR Mode

Any voice channel can be placed into ECR mode via the **dx_listenecr()** or **dx_listenecrex()** function on an ECR feature-enabled board. When a voice channel is placed in ECR mode, the echo reference SCbus time slot is specified and the high performance echo canceller is activated. The ECR mode supplies 128 tap (16 ms) echo cancellation.

When an echo carrying signal is provided as an input to the ECR by an associated **dx_listen()** function, an echo-canceled version of that signal is produced on the echo cancellation SCbus time slot. If no echo carrying signal is defined, the contents of the echo-cancellation transmit time slot are undefined and unpredictable. Other characteristics of the echo canceller can be set if the ECR mode is activated using the **dx_listenecrex()** function. See the *Voice Programmer's Guide for Linux* for more information on this function.

NOTE: **dx_listen()** may precede or follow the **dx_listenecr()** or **dx_listenecrex()** function. If multiple **dx_listen()** and **dx_listenecr()** or **dx_listenecrex()** function calls are issued against a single channel, the echo cancellation operates on the last two issued. Successive **dx_listenecr()** or **dx_listenecrex()** functions can be issued without requiring any **dx_unlistenecr()** between them.

While a channel is in ECR mode, a number of standard voice operations are not available. The unavailable operations include the following:

11. Echo Cancellation

- Play
- Record

NOTE: 8kHz PCM record is the only supported record encoding when a channel is in the ECR mode. Any such 8kHz PCM record is a recording of the echo-canceled signal.

- Dial
- Tone Generation
- R2 MF
- Transaction Record

If a channel is actively performing any of the above operations, a **dx_listenecr()** or **dx_listenecrex()** function is not performed, and the function returns an error to the application. Conversely, if a channel is in ECR mode, a request for any of these operations is not honored, except for the record noted. A channel may be returned to SVP mode dynamically via the **dx_unlistenecr()** function.

11.1.2. Buffer Size Adjustments for Internet Telephony

Previously, data buffers on the Dialogic voice board (called firmware buffers) had a fixed size of 512 bytes. To reduce voice latency in Internet telephony applications, the firmware buffer size is now programmable from 128 to 512 bytes.

Dialogic Voice API functions stream voice data between the Dialogic voice board firmware buffers and the telephony application via driver buffers. The size of these driver buffers can be set to any value between 256 bytes and 16 KB. Overall signal delay can be reduced by adjusting both the firmware and driver buffers.

Refer to the *Release 2 Software Installation Reference for Linux* for more information.

11.2. ECR Application Models

Two application models are provided in this section to illustrate building an echo-canceled connection via the SCbus.

11.2.1. How to Set Up the ECR Bridge

This application model uses two Modular Station Interface (MSI/SC) station devices connected via the SCbus to two voice channel devices. The voice channel devices are operating in ECR mode. Two telephones (*Figure 20*) are connected to the MSI/SC stations for providing input and for listening to the echo-canceled output of each voice device.

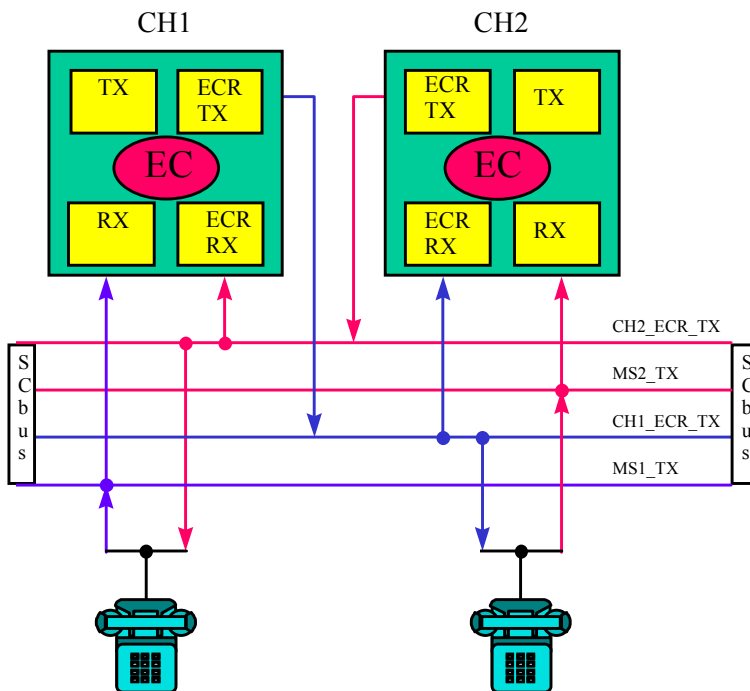


Figure 20. ECR Bridge Example Diagram

11. Echo Cancellation

1. Get SCbus transmit time slots of both MSI/SC devices and the ECR transmit time slots of the two voice channel devices.

```
ms_getxmitslot (MS1, &MS1_TX);
```

```
ms_getxmitslot (MS2, &MS2_TX);
```

```
dx_getxmitslotecr (CH1, &CH1_ECR_TX);
```

```
dx_getxmitslotecr (CH2, &CH2_ECR_TX);
```

2. Have both MSI/SC stations listen to the ECR transmit of the opposite voice channel.

```
ms_listen (MS1, &CH2_ECR_TX);
```

```
ms_listen (MS2, &CH1_ECR_TX);
```

3. Have both voice channel devices listen to their corresponding MSI/SC station device.

```
dx_listen (CH1, &MS1_TX);
```

```
dx_listen (CH2, &MS2_TX);
```

4. Have each voice channel connect its echo canceller's receive time slot to the opposite echo canceller's ECR transmit. These signals are used as echo reference signals.

```
dx_listenecr (CH1, & CH2_ECR_TX);
```

```
dx_listenecr (CH2, & CH1_ECR_TX);
```

■ Example

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

main()
{
    int chdev1, chdev2;          /* Voice channel device handles */
    int msdev1, msdev2;        /* MSI/SC station device handles */
    SC_TSINFO sc_tsinfo;      /* SCbus time slot information structure */
    long scts;                /* Pointer to SCbus time slot */
    long ms1txts, ms2txts,    /* Transmit time slots of stations 1 & 2 */
    ch1ecrxtxs, ch2ecrxtxs; /* Transmit time slots of echo-cancellers on voice channels 1
                             & 2 */

    /* Open voice board 1 channel 1 device */
    if ((chdev1 = dx_open("dxxxB1C1", 0)) == -1) {
        printf("Cannot open channel dxxxB1C1.  errno = %d", errno);
        exit(1);
    }
    /* Open voice board 1 channel 2 device */
    if ((chdev2 = dx_open("dxxxB1C2", 0)) == -1) {
        printf("Cannot open channel dxxxB1C2.  errno = %d", errno);
        exit(1);
    }
    /* Open MSI/SC board 1 station 1 device */
    if ((msdev1 = ms_open("msiB1C1", 0)) == -1) {
        printf("Cannot open station msiB1C1.  errno = %d", errno);
        exit(1);
    }
    /* Open MSI/SC board 1 station 2 device */
    if ((msdev2 = ms_open("msiB1C2", 0)) == -1) {
        printf("Cannot open station msiB1C2.  errno = %d", errno);
        exit(1);
    }

    /* Initialize an SCbus time slot information */
    sc_tsinfo.sc_numts = 1;
    sc_tsinfo.sc_tsarrayp = &scts;

    /* Get SCbus time slot connected to transmit of MSI/SC station 1 on board 1 */
    if (ms_getxmitslot(msdev1, &sc_tsinfo) == -1) {
        printf("Error message = %s, on %s", ATDV_ERRMSGP(msdev1), ATDV_NAMEP(msdev1));
        exit(1);
    }
    ms1txts = scts;

    /* Get SCbus time slot connected to transmit of MSI/SC station 2 on board 1 */
    if (ms_getxmitslot(msdev2, &sc_tsinfo) == -1) {
        printf("Error message = %s, on %s", ATDV_ERRMSGP(msdev2), ATDV_NAMEP(msdev2));
        exit(1);
    }
    ms2txts = scts;
}
```

11. Echo Cancellation

```
/* Get SCbus time slot connected to transmit of voice channel 1 on board 1 */
if (dx_getxmitslotecr(chdev1, &sc_tsinfo) == -1) {
    printf("Error message = %s, on %s", ATDV_ERRMSGP(chdev1), ATDV_NAMEP(chdev1));
    exit(1);
}
chlecrtxts = scts;

/* Get SCbus time slot connected to transmit of voice channel 2 on board 1 */
if (dx_getxmitslotecr(chdev2, &sc_tsinfo) == -1) {
    printf("Error message = %s, on %s", ATDV_ERRMSGP(chdev2), ATDV_NAMEP(chdev2));
    exit(1);
}
ch2ecrtxts = scts;

/* Have MSI/SC station 1 listen to channel 2's echo-cancelled transmit */
if (ms_listen(msdev1, &sc_tsinfo) == -1) {
    printf("Error message = %s, on %s", ATDV_ERRMSGP(msdev1), ATDV_NAMEP(msdev1));
    exit(1);
}

scts = chlecrtxts;

/* Have MSI/SC station 2 listen to channel 1's echo-cancelled transmit */
if (ms_listen(msdev2, &sc_tsinfo) == -1) {
    printf("Error message = %s, on %s", ATDV_ERRMSGP(msdev2), ATDV_NAMEP(msdev2));
    exit(1);
}

scts = ms1txts;

/* Have channel 1 listen to station 1's transmit */
if (dx_listen(chdev1, &sc_tsinfo) == -1) {
    printf("Error message = %s, on %s", ATDV_ERRMSGP(chdev1), ATDV_NAMEP(chdev1));
    exit(1);
}

scts = ms2txts;

/* Have channel 2 listen to station 2's transmit */
if (dx_listen(chdev2, &sc_tsinfo) == -1) {
    printf("Error message = %s, on %s", ATDV_ERRMSGP(chdev2), ATDV_NAMEP(chdev2));
    exit(1);
}

scts = ch2ecrtxts;

/* Have channel 1's echo-canceller listen to channel 2's echo-cancelled transmit */
if (dx_listenecr(chdev1, &sc_tsinfo) == -1) {
    printf("Error message = %s, on %s", ATDV_ERRMSGP(chdev1), ATDV_NAMEP(chdev1));
    exit(1);
}

scts = chlecrtxts;

/* Have channel 2's echo-canceller listen to channel 1's echo-cancelled transmit */
if (dx_listenecr(chdev2, &sc_tsinfo) == -1) {
    printf("Error message = %s, on %s", ATDV_ERRMSGP(chdev2), ATDV_NAMEP(chdev2));
    exit(1);
}
}
/* Bridge connected, both stations receive echo-cancelled signal */
```

Voice Software Reference - Features Guide for Linux

```
/*
 *
 * Continue
 *
 */

/* Then perform xx_unlisten() and dx_unlistenecr(), plus all necessary xx_close()s */

if (ms_unlisten(msdev2) == -1) {
    printf("Error message = %s, on %s", ATDV_ERRMSGP(msdev2), ATDV_NAMEP(msdev2));
    exit(1);
}
if (ms_unlisten(msdev1) == -1) {
    printf("Error message = %s, on %s", ATDV_ERRMSGP(msdev1), ATDV_NAMEP(msdev1));
    exit(1);
}
if (dx_unlistenecr(chdev2) == -1) {
    printf("Error message = %s, on %s", ATDV_ERRMSGP(chdev2), ATDV_NAMEP(chdev2));
    exit(1);
}
if (dx_unlistenecr(chdev1) == -1) {
    printf("Error message = %s, on %s", ATDV_ERRMSGP(chdev1), ATDV_NAMEP(chdev1));
    exit(1);
}
if (dx_unlisten(chdev2) == -1) {
    printf("Error message = %s, on %s", ATDV_ERRMSGP(chdev2), ATDV_NAMEP(chdev2));
    exit(1);
}
if (dx_unlisten(chdev1) == -1) {
    printf("Error message = %s, on %s", ATDV_ERRMSGP(chdev1), ATDV_NAMEP(chdev1));
    exit(1);
}
if (dx_close(chdev1) == -1) {
    printf("Error message = %s, on %s", ATDV_ERRMSGP(chdev1), ATDV_NAMEP(chdev1));
    exit(1);
}
if (dx_close(chdev2) == -1) {
    printf("Error message = %s, on %s", ATDV_ERRMSGP(chdev2), ATDV_NAMEP(chdev2));
    exit(1);
}
if (ms_close(msdev1) == -1) {
    printf("Error message = %s, on %s", ATDV_ERRMSGP(chdev1), ATDV_NAMEP(chdev1));
    exit(1);
}
if (ms_close(msdev2) == -1) {
    printf("Error message = %s, on %s", ATDV_ERRMSGP(msdev2), ATDV_NAMEP(msdev2));
    exit(1);
}
return(0);
}
```

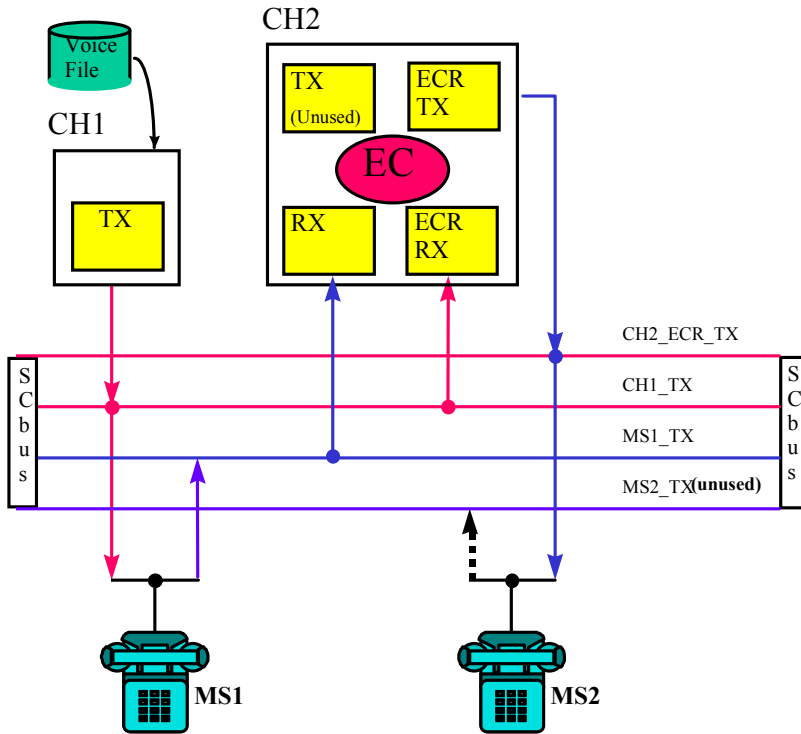


Figure 21. ECR Play Over the SCbus

11.2.2. How to Set Up an ECR Play Over the SCbus

In this model, two MSI/SC station devices (*Figure 21*) are connected via the SCbus to two voice channel devices. The second voice channel device is operating in ECR mode. Two telephones are connected to the MSI/SC stations for providing input and listening to the echo-cancelled output of the second voice channel device, and to the non-echo-cancelled output of the first voice channel device.

1. Get SCbus transmit time slots of both MSI/SC devices and the ECR transmit time slots of the two voice channel devices.

```
ms_getxmitslot (MS1, &MS1_TX);
dx_getxmitslot (CH1, &CH1_TX);
dx_getxmitslotecr (CH2, &CH2_ECR_TX);
```

Voice Software Reference - Features Guide for Linux

2. Have the MSI/SC station 1 listen to the transmit (TX) of channel 1.

```
ms_listen (MS1, & CH1_TX);
```

3. Have MSI/SC station 2 listen to the ECR transmit of channel 2.

```
ms_listen (MS2, & CH2_ECR_TX);
```

4. Have voice channel 2 listen to MSI/SC station 1's transmit.

```
dx_listen (CH2, & MS1_TX);
```

5. Have voice channel 2 connect its echo canceller's receive time slot to transmit of channel 1. This signal is used as the echo reference signal.

```
dx_listenecr (CH2, & CH1_TX);
```

■ Example

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

main()
{
    int chdev1, chdev2;          /* Voice channel device handles */
    int msdev1, msdev2;         /* MSI/SC station device handles */
    SC_TSINFO sc_tsinfo;       /* SCbus time slot information structure */
    long scts;                  /* Pointer to SCbus time slot */
    long ms1txts,               /* Transmit time slots of stations 1 & 2 */
        ch1txts, ch2ecrtxts;   /* Transmit time slots of echo-cancellers on
                                voice channels 1 & 2 */

    /* Open voice board 1 channel 1 device */
    if ((chdev1 = dx_open("dxxB1C1", 0)) == -1) {
        printf("Cannot open channel dxxB1C1.  errno = %d", errno);
        exit(1);
    }
    /* Open voice board 1 channel 2 device */
    if ((chdev2 = dx_open("dxxB1C2", 0)) == -1) {
        printf("Cannot open channel dxxB1C2.  errno = %d", errno);
        exit(1);
    }
    /* Open MSI/SC board 1 station 1 device */
    if ((msdev1 = ms_open("msiB1C1", 0)) == -1) {
        printf("Cannot open station msiB1C1.  errno = %d", errno);
        exit(1);
    }
    /* Open MSI/SC board 1 station 2 device */
    if ((msdev2 = ms_open("msiB1C2", 0)) == -1) {
        printf("Cannot open station msiB1C2.  errno = %d", errno);
        exit(1);
    }
}
```

11. Echo Cancellation

```
/* Initialize an SBus time slot information */
sc_tsinfo.sc_numts = 1;
sc_tsinfo.sc_tsarray = &scts;

/* Get SBus time slot connected to transmit of voice channel 1 on board 1 */
if (ms_getxmitslot(msdev1, &sc_tsinfo) == -1) {
printf("Error message = %s, on %s", ATDV_ERRMSGP(msdev1), ATDV_NAMEP(msdev1));
exit(1);
}
msltxts = scts;

/* Get SBus time slot connected to transmit of voice channel 1 on board 1 */
if (dx_getxmitslot(chdev1, &sc_tsinfo) == -1) {
printf("Error message = %s, on %s", ATDV_ERRMSGP(chdev1), ATDV_NAMEP(chdev1));
exit(1);
}
chltxts = scts;

/* Get SBus time slot connected to transmit of voice channel 1 on board 1 */
if (dx_getxmitslotecr(chdev2, &sc_tsinfo) == -1) {
printf("Error message = %s, on %s", ATDV_ERRMSGP(chdev2), ATDV_NAMEP(chdev2));
exit(1);
}
ch2ecrxts = scts;

/* Have station 1 listen to file played by voice channel 1 */
scts = chltxts;
if (ms_listen(msdev1, &sc_tsinfo) == -1) {
printf("Error message = %s, on %s", ATDV_ERRMSGP(msdev1), ATDV_NAMEP(msdev1));
exit(1);
}

/* Have station 2 listen to echo-cancelled output of voice channel 2 */
scts = ch2ecrxts;
if (ms_listen(msdev2, &sc_tsinfo) == -1) {
printf("Error message = %s, on %s", ATDV_ERRMSGP(msdev2), ATDV_NAMEP(msdev2));
exit(1);
}

/* Have voice channel 2 listen to echo-carrying signal from station 1 */
scts = msltxts;
if (dx_listen(chdev2, &sc_tsinfo) == -1) {
printf("Error message = %s, on %s", ATDV_ERRMSGP(chdev1), ATDV_NAMEP(chdev1));
exit(1);
}

/* And activate the ECR feature on voice channel 2, with the echo-reference signal
coming from voice channel 1 */
scts = chltxts;
if (dx_listencr(chdev2, &sc_tsinfo) == -1) {
printf("Error message = %s, on %s", ATDV_ERRMSGP(chdev2), ATDV_NAMEP(chdev2));
exit(1);
}

/* Setup completed, any signal transmitted from channel device 1,
* will a) be received by station 1,
* b) contribute echo to the transmit of station 1,
* c) will be heard AFTER echo-cancellation (on channel 2) by
* station 2.*/
```

Voice Software Reference - Features Guide for Linux

```
/*
.
. Continue
.
*/

/* Then perform xx_unlisten() and dx_unlistenecr(), plus all necessary xx_close()s */
if (ms_unlisten(msdev2) == -1) {
    printf("Error message = %s, on %s", ATDV_ERRMSGP(msdev2), ATDV_NAMEP(msdev2));
    exit(1);
}
if (ms_unlisten(msdev1) == -1) {
    printf("Error message = %s, on %s", ATDV_ERRMSGP(msdev1), ATDV_NAMEP(msdev1));
    exit(1);
}
if (dx_unlistenecr(chdev2) == -1) {
    printf("Error message = %s, on %s", ATDV_ERRMSGP(chdev2), TDV_NAMEP(chdev2));
    exit(1);
}
if (dx_unlisten(chdev2) == -1) {
    printf("Error message = %s, on %s", ATDV_ERRMSGP(chdev2), ATDV_NAMEP(chdev2));
    exit(1);
}
if (dx_close(chdev1) == -1) {
    printf("Error message = %s, on %s", ATDV_ERRMSGP(chdev1), ATDV_NAMEP(chdev1));
    exit(1);
}
if (dx_close(chdev2) == -1) {
    printf("Error message = %s, on %s", ATDV_ERRMSGP(chdev2), ATDV_NAMEP(chdev2));
    exit(1);
}
if (ms_close(msdev1) == -1) {
    printf("Error message = %s, on %s", ATDV_ERRMSGP(msdev1), ATDV_NAMEP(msdev1));
    exit(1);
}
if (ms_close(msdev2) == -1) {
    printf("Error message = %s, on %s", ATDV_ERRMSGP(msdev2), ATDV_NAMEP(msdev2));
    exit(1);
}
return(0);
}
```

12. G.726 ADPCM Voice Coder

G.726 is an ITU-T recommendation that specifies an adaptive differential pulse code modulation (ADPCM) technique for recording and playing back audio files. It is useful for applications that require speech compression, encoding for noise immunity, and uniformity in transmitting voice and data signals.

12.1. Dialogic Support for G.726

Dialogic provides a G.726 bit exact voice coder that is compliant with the ITU-T G.726 recommendation. Messages are stored at 32 Kbps using G.726 ADPCM.

Audio encoded in the G.726 bit-exact format complies with the Voice Profile for Internet Messaging (VPIM), a communications protocol that makes it possible to send and receive messages from disparate messaging systems over the Internet. G.726 bit exact is the audio encoding and decoding standard supported by VPIM. See *Appendix A* for a list of ITU-T G.726 documents.

NOTES: 1. The G.726 voice coder is currently supported only on the D/300PSC-E1 board.

NOTES: 2. The G.726 voice coder is enabled on a per-board basis.

12.2. Enabling and Using the G.726 Voice Coder

Before using the G.726 voice coder, turn on the feature in the *dialogic.cfg* file:

```
g726_std = ON
```

Select the G.726 voice coder for use on a board by specifying the **DATA_FORMAT_G726** parameter in the DX_XPB data structure.

Following is sample of the values for G.726 in the DX_XPB structure:

```
wFileFormat:          FILE_FORMAT_VOX
wDataFormat:          DATA_FORMAT_G726
nSamplesPerSec:       DRT_8KHZ
nBitsPerSample:       4
```

Voice Software Reference - Features Guide for Linux

Use the `DX_XPB` structure in conjunction with the `dx_playiottdata()` and `dx_reciottdata()` functions. See the function reference and the `DX_XPB` data structure sections of the *Voice Programmer's Guide for Linux* for more information.

To determine the voice resource handles to be used with `dx_playiottdata()` and `dx_reciottdata()`, call functions such as `sr_getboardcnt()`, `ATDV_SUBDEVS()`, and `ATDX_CHNAMES()` to return the number of boards of a particular type, the number of subdevices for the device, and the channel device names.

13. Voice Library Demo Programs

There are five Voice Library demo programs included with the voice software for Linux.

- d40demo
- custserv
- horoscope
- cbansr
- pansr

The purpose of the demo programs is to show how to use the Voice Library functions in a voice application. They provide a working skeleton on which to base your applications. The Voice Library demos are located in the */usr/dialogic/demos* directory. A directory structure for the demos directory is shown in *Figure 22*.

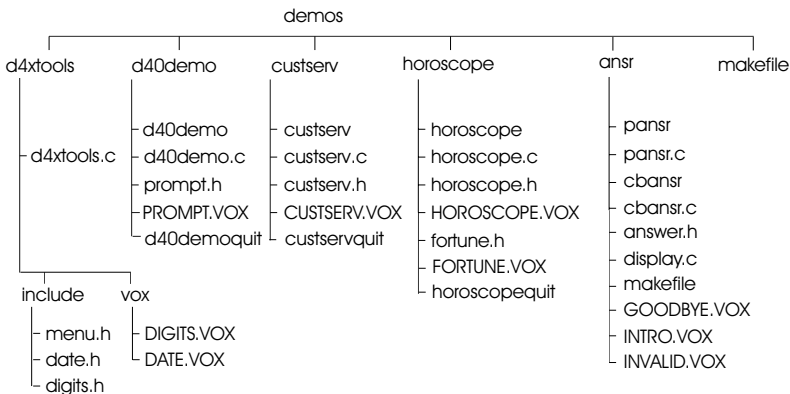


Figure 22. Voice Library Demo Directory Structure

Voice Software Reference - Features Guide for Linux

The distribution media include the source and executable version of the demo programs, and a makefile to compile the source. The source for the demos is written in C and is instructional to those with a C and Linux programming background.

A toolkit of general-purpose C routines is also included with the demo programs. This toolkit aids in the development of voice applications. The toolkit source is contained in the file *d4xtools.c*.

The main demo program, *d40demo*, is an implementation of a simple order entry application. This chapter explains the details of the *d40demo* source code and the routines contained in the toolkit. The other demos provided are not explained in detail, but each demo is described briefly in *Sections 13.2. Other Synchronous Demos* and *13.3. Asynchronous Demo Programs pansr and cbansr*.

13.1. D/40demo - Synchronous Demo

13.1.1. Boards Supported

The *D/40demo* program can be used with the following Dialogic voice boards: *D/21D*, *D/21H*, *D/41D*, *D/41ESC*, *D/41H*, and *D/160SC-LS*, assuming the Dialogic System Release for the operating system you are using also supports a given board.

13.1.2. Physical Connections

The demo can be run by configuring the system using one or more supported Dialogic voice boards connected to a phone via a central office (CO), CO simulator, or PBX. *Figure 23* shows how to connect a *D/41ESC* board to a phone.

13. Voice Library Demo Programs

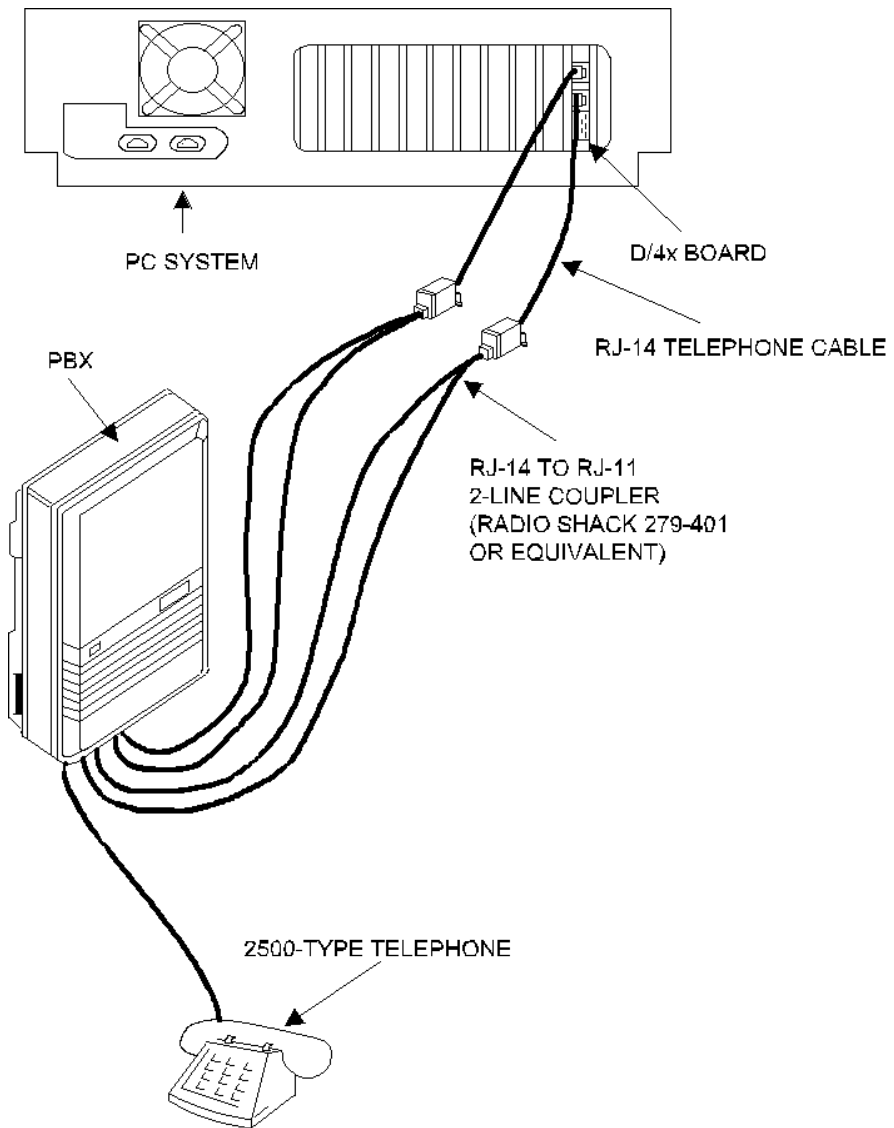


Figure 23. Connecting D/41ESC Board to Phone

Figure 24 shows connecting a D/41ESC board to a phone and Figure 25 shows connecting a D/160SC-LS board.

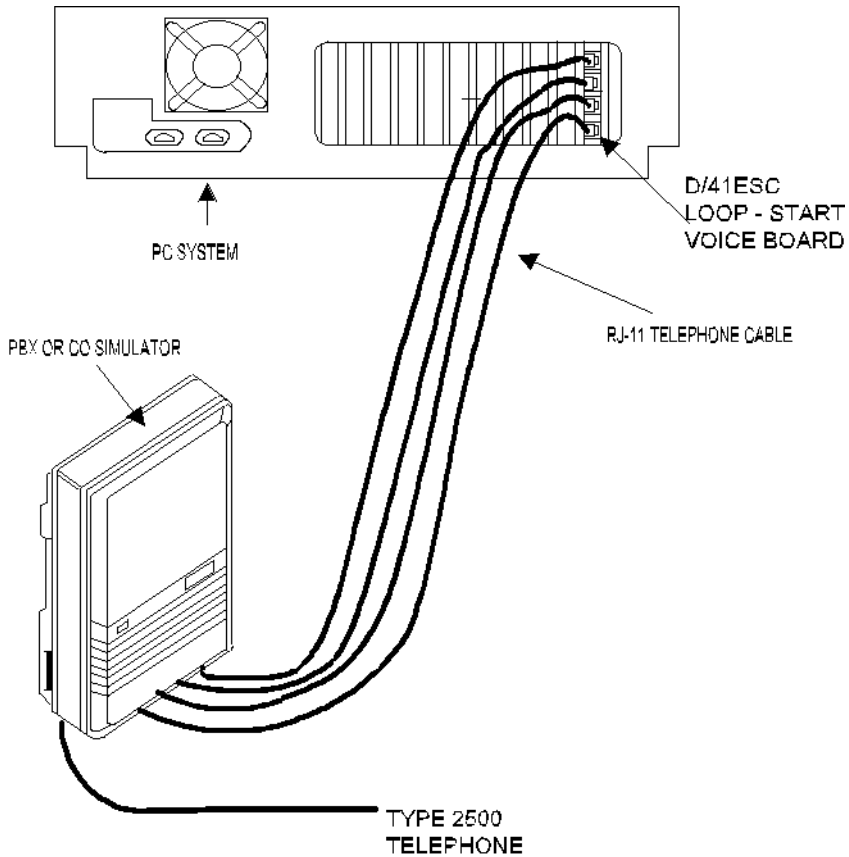


Figure 24. Connecting D/41ESC Board to Phone

13. Voice Library Demo Programs

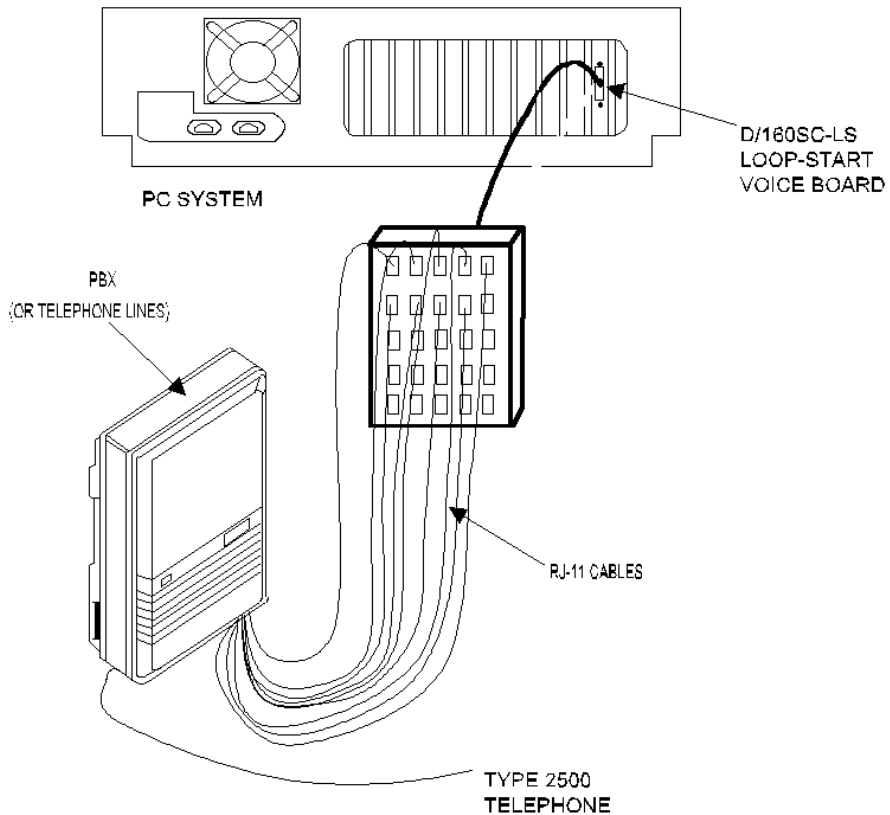


Figure 25. Connecting D/160SC-LS Board to Phone

13.1.3. Running the d40demo Program

To use the d40demo, change to the directory containing the executable version of the demo program. To run the demo, type the demo executable name at the system prompt and use the channel name you wish to run the demo on as the argument, as detailed below.

To run the d40demo program:

1. Change to the directory containing the executable version of the program:

Voice Software Reference - Features Guide for Linux

```
$ cd /usr/dialogic/demos/d40demo
```

2. Type the following at the command line:

```
$ d40demo [device name list]
```

3. The device name list is a list of one or more channels that the demo will use; for example:

```
$ d40demo dxoxB1C1 dxoxB1C2
```

In the above example, d40demo uses channels 1 and 2 on board 1.

13.1.4. d40demo Program Overview

The d40demo program creates a background process for each channel that was specified as an argument. The main process terminates and you are left with one background process per channel.

Each background process waits to receive a call on its channel. When a call is received, the phone is taken off-hook after one ring, and an introduction message is played. The caller is presented with a menu of four choices:

- place an order by pressing "1"
- check an order by pressing "2"
- cancel an order by pressing "3"
- end the call by pressing "*"

The caller is prompted for a response. An invalid response prompts the caller to try again. A valid response prompts the caller to complete the action he chose. It is suggested that you run d40demo and become familiar with it before reading further about how it works.

13.1.5. Source Code Overview

The following section contains an overview of the source code.

Outline

The source code to d40demo is located in two modules, *d40demo.c* and *d4xtools.c*. As an introduction to the program, a simplified outline of d40demo is shown:

```
INITIALIZATION
  Spawn subprocess for each channel
  End parent process
  Initialize interrupt handlers
  Open channel and files
  Initialize process variables

MAIN PROGRAM BODY
  Place channel onhook
  Wait for one ring
  Go offhook
  Play introduction message

START MENU ROUTINE LOOP
  Play menu prompt
  Get caller response
  Process caller response
END LOOP
```

The outline provides a general approach to any voice application, not only d40demo. For example, all voice applications need some type of initialization routine to open the channel, initialize variables, and open files. A voice menu system is also a universal feature found in most voice applications because a caller needs a vocal prompt to instruct him on his choice of actions. These routines are called system routines and are in the *d4xtools.c* module.

Since d40demo is also a working order entry system, the source code contains nonsystem routines that are specific to the order entry application. The *d40demo.c* module contains the routines that manage a simple database, which are called the application specific routines.

The d40demo program uses three system routines that will be discussed in this chapter:

- Initialization

Voice Software Reference - Features Guide for Linux

- Menu System Routine
- Messaging System Routine

These routines are discussed in *Sections 13.1.6. Initialization through 13.1.8. Messaging System Routine*. Emphasis will be placed on the menu system and the initialization routines.

Global Variables

There are global variables used by the functions in *d4xtools.c*. In the following sections of this chapter, the examples refer to these variables. These variables are defined in *d4xtools.c*:

Variable	Definition
cur_menu	Current Menu - A pointer to the current active menu.
pre_menu	Previous Menu - A pointer to the previous menu.
idlestate	Idle State Reentry Environment - This is the reentry point to the process. Using the longjmp() and setjmp() Linux functions, idlestate is set to be the point where the process sets the hookstate to onhook and waits for the phone to ring.
devhandle	Device handle - The variable that holds the descriptor for the channel.

There are two default DV_TPT arrays that are defined in *d40demo.c* and are used by the *d4xtools.c* system routines:

Variable	Definition
def_rp_tpt	Default Record and Play - The default DV_TPT array for recording and playing using dx_rec() and dx_play() .
def_dg_tpt	Default Get Digit - The default DV_TPT array for getting digit responses using the dx_getdig() function.

13.1.6. Initialization

When `d40demo` is started, the application is initialized by a call in `main()` to `init_sys()`. The `init_sys()` function is located in the `d4xtools.c` module. It performs system initialization by spawning a child process for each channel, configuring the signal handler, opening the channel, initializing the message table, and ending the parent process. The `init_sys()` code is:

```
int init_sys(argc,argv)
int  argc;
char  *argv[] ;

{
    int rc;
    register DL_MSGTBL * msgtblp;
    char str[80];

    /*
    * Loop through each argument spawning a child for each and every
    * device name in the argument list. After spawning all child
    * processes the parent will terminate. Every child process will
    * return a unique index number which is the device name argument
    * number used for that child.
    */
    while (--argc) {
        /* Fork a copy of the application to handle this channel */
        if ((rc = fork()) == -1) {
            (void) fprintf(stderr, "%s: Error forking child process.", prgnamep);
            perror("");
            exit(1);
        }

        /* If this is the child then return the child's unique index number */
        if (rc == 0) {
            (void) close(0);

            /*
            * Set up a signal handler to handle shutdown
            */
            (void) signal(SIGINT, sigcatch);
            (void) signal(SIGQUIT, sigcatch);
            (void) signal(SIGTERM, sigcatch);
            (void) signal(SIGHUP, sigcatch);

            /* Give global access to the device name */
            (void) strcpy(devname, argv[argc]);

            /* Open the channel */
            if ((devhandle=dx_open(devname,O_RDWR)) == -1){
                disperr("failed to open device.");
            }
        }
    }
}
```

Voice Software Reference - Features Guide for Linux

```
/*
 * Open all message files listed in the message table and assign
 * each returned descriptor to the appropriate field in the same
 * message table entry.
 */
for (msgtblp = dx_msgtbl; msgtblp->mt_fn; msgtblp++) {
    if ((msgtblp->mt_fd = open(msgtblp->mt_fn, O_RDONLY)) == -1) {
        (void) sprintf(str, "failed to open %s", msgtblp->mt_fn);
        disperr(str);
    }
}
return(argc);
}
/*
 * The parent process has spawned all its children and is ready to die.
 */
exit(0);

/* Never gets here but makes lint happy */
return (0);
}
```

For every argument supplied at the command line, the **init_sys()** function uses the Linux system call **fork()** to spawn a child copy of the *d40demo* process. Any code following the call to **fork()** is executing in both the child and the parent process. The following code fragment checks if the process is a child process, and if it is, closes file descriptor 0, the standard input, if it is:

```
if (rc == 0) {
    close(0);
    :
}
```

The code that follows the **close()** function call within the if statement is only executed in the child process. The parent process fails the (`rc == 0`) comparison and loops back to the while statement. When all the arguments are exhausted, the parent process eliminates itself by a call to the **exit()** function.

The **signal()** function calls redirect the signals from the outside world to call the **sigcatch()** function. The **sigcatch()** function, located in *d4xtools.c*, is the signal processing routine for each channel. It stops I/O on the channel, sets the hook state to onhook, and exits the process. By intercepting the signals and using **sigcatch()**, the process exits cleanly if it receives an interrupt, quit, terminate, or hang-up signal from the outside world.

13. Voice Library Demo Programs

The **strcpy()** function copies the channel device name to a global variable to give all functions access to the device name.

The channel specified on the command line is opened by the **dx_open()** function with read and write privileges. The descriptor for the channel is stored in the global variable `devhandle`.

The message structure is initialized in the for-loop. It scans every entry in the message structure, opens the specified file with read-only privileges, and stores the file descriptor for the open file in the message structure. This code allows an arbitrary number of playback files or devices to be opened. You only need to specify a `DL_MSGTBL` structure for each playback file or device, and the **init_sys()** function will open all the files and store the file descriptors.

The final action of a child process is to return the argument number as the return value. *d40demo.c* uses this value when it initializes the order entry database.

13.1.7. Menu System Routine

d40demo is built around a menu system that controls the flow of the program. The menu system is a general concept that is useful for any voice application. Understanding how the menu system works is essential to understanding how *d40demo* works.

Model of the Menu System

In general, when a call is received by a voice application, all callers are presented with the same initial voice menu. This menu is referred to as the root menu. For example, in a customer service application, the initial prompt might introduce the company and ask the caller to press 1 for sales or 2 for support. The caller listens to the voice prompt and presses 1. The application prompts with another menu, asking the caller to press 1 for Jane, 2 for John, or 3 to hang up. The caller presses 2 and is prompted to record a message to John. When the caller completes recording he is prompted with the same menu asking him to press 1 for Jane, 2 for John, or 3 to hang-up. The caller presses 3 and is disconnected.

Voice Software Reference - Features Guide for Linux

From the example above, it can be seen that all menus have the following characteristics. They:

- play voice prompts
- require caller input
- transfer control based on caller input

When input is received from the caller, four things can happen:

- control is passed to another menu
- an action is performed
- an action is performed and control is passed to another menu
- an error message is played (invalid input)

All menus provide the caller with options. In the example above, the first caller response transferred the caller to another menu. The second response performed an action (recording a message) and prompted the caller with the same menu. In another case, the caller might have been transferred to a different menu.

A menu must obey the following rules:

- A menu must always transfer control to a menu. This can be a different menu or the same menu.
- A menu is not required to perform an action. If the menu does perform an action, it still must pass control to some menu.

Implementation

The menu system is implemented by using three data structures: `DL_MENU`, `DL_MNUOPTS`, and `DL_DATA`, and three functions: `menu_engine()`, `gt_data()`, and `val_menu()`. These functions are defined in `d4xtools.c`. By defining the structures and using the functions, a complete menu system for any voice application can be implemented.

To see how a menu system can be implemented, examine the code that defines `d40demo`'s main menu as examples. The definition for `d40demo`'s only menu is located in the variables `menu_1`, `menu1_opts`, and `data_1`, which are declared in `d40demo.c`.

DL_MENU

The DL_MENU structure defines the menu. It uses a DL_MNUOPTS structure to define the response options and a DL_DATA structure to define the termination conditions for a response.

```
typedef struct DL_MENU {
    DL_MNUOPTS * me_opttbl;          /* Pointer to menu's option table */
    DL_MENU * me_defmenu;           /* Default next menu for this menu */
    DL_MNUOPTS * (*me_validfnc) (); /* Pointer to menu's validation function */
    char * me_prompt;               /* Pointer to menu's prompt message name */
    char * me_retry;                /* Pointer to menu's retry message name */
    void (*me_error) ();            /* Pointer to menu's error message name */
    DL_DATA * me_data;              /* Pointer to Data table for this menu */
} DL_MENU;
```

Field	Definition
me_opttbl	Option Table - Pointer to a DL_MNUOPTS structure that defines the options for this menu.
me_defmenu	Default Menu - Pointer to the default next-menu. The menu system will transfer control to the menu specified in this field after an action has taken place, but only if the mn_nxtmenu structure specified in the option table is NULL. This field can contain a pointer to any valid DL_MENU structure. In addition, the <i>menu.h</i> header file defines two DL_MENU structures that can be used, DL_PREV and DL_CURR. DL_PREV returns the caller to the previous menu, and DL_CURR returns the caller to the current menu.
me_validfnc	Validation Function - Pointer to the function that validates the caller's response. This function returns a DL_MNUOPTS structure containing the response that was chosen.
me_prompt	Prompt Message - Pointer to the voice prompt message for this menu.
me_retry	Retry Message - Pointer to the retry prompt message for this menu. This message is played if the caller response is not found in the option table.

Voice Software Reference - Features Guide for Linux

me_error	Error Function - Pointer to the function that will handle a fatal error occurring during this menu. Fatal errors occur when a valid response is not given within the maximum number of retries. The maximum number of retries is set by the DL_DATA structure associated with this menu.
me_data	Data Table - Pointer to a DL_DATA structure that defines the termination conditions for getting data during this menu.

The DL_MENU structure implements the functions a menu must perform. A menu needs to:

- provide options to the caller
- prompt the caller to input an option
- validate the caller input
- perform an action based on input
- provide a prompt if the caller makes a mistake
- provide a way out if a fatal error occurs
- transfer control to some other menu or itself when everything is done

The fields of a DL_MENU structure provide the means to specify all these functions. The root menu for d40demo is defined in a DL_MENU structure with the variable name menu_1. The comments explain what each value indicates.

```
DL_MENU menu_1 = {
    menu_opts,      /* The option table for menu 1 is in menu_opts */
    DM_CURR,       /* Return to current menu after completing action */
    val_menu,      /* val_menu() is the input validation function */
    "mm_menu",     /* mm_menu is the main menu voice prompt */
    "mm_retry",    /* mm_retry is the main menu retry prompt */
    goodbye,       /* goodbye() is the error function */
    &data_1        /* The termination condition are in data_1 */
};
```

DL_MNUOPTS

The DL_MNUOPTS structure defines the caller's response options for a menu. It specifies the digits a caller can enter and whether the digits indicate an action or another menu. The options for a menu are implemented as a NULL-terminated

13. Voice Library Demo Programs

array of DL_MNUOPTS structures. An array of DL_MNUOPTS structures allows a menu to have an arbitrary amount of options.

```
typedef struct DL_MNUOPTS {  
    char      * mo_rspstr;           /* Response string to this option      */  
    void      (*mo_fcnt) ();        /* Next function for this option      */  
    DL_MENU   * mo_nxtmenu;        /* Next menu for this option          */  
} DL_MNUOPTS;
```

The fields are defined as follows:

Field	Definition
mo_rspstr	Response String - This is the digit string that the caller must press to initiate this action.
mo_fcnt	Next Function - This is the name of the void function that will be called if the digits in mn_rspstr were received.
mo_nxtmenu	Next Menu - This is a pointer to the next menu. If the digit specified by mn_rspstr transfers control to another menu, this field points to that menu. If this field is NULL the control is transferred to the menu pointed to by me_defmenu in the DL_MSG structure that contains the option table.

The options available in a menu are:

- transfer control to another menu
- execute a function
- do both
- do neither

All menus must provide a pointer to the next menu at one of two levels. The next menu can be tied to the response and be specified in the mn_nxtmnu field of the DL_MNUOPTS structure. If this field is NULL, the next menu defaults to a higher level. The default next menu for all options in a menu is defined in the me_defmenu field of the DL_MENU structure. The action specified in the mn_fcnt field of the DL_MNUOPTS structure isn't required for every menu option.

Voice Software Reference - Features Guide for Linux

NOTE: The next menu for a response must be specified as either:

- a next menu (`mo_nxtmnu`) in the `DL_MNUOPTS` structure
- or the default menu (`mo_fcmt`) in the `DL_MENU` structure for all the options not specifying a next menu.

The `menu1_opts` variable is the NULL-terminated array for the root menu in `d40demo.c`. It is initialized as:

```
DL_MNUOPTS menu1_opts[] = {
    {"1", enterord, NULL},
    {"2", chekord, NULL},
    {"3", canord, NULL},
    {"*", goodby, NULL},
    {NULL, NULL, NULL}
};
```

All the options for `d40demo` use the default next menu. The four possible inputs for `d40demo` (1,2,3, or *) are defined in the `menu1_opts` array. For example, the first table entry in `menu1_opts` defines the following: If the caller presses the digit 1, call the void function `enterord()`.

DL_DATA

The `DL_DATA` structure defines the termination conditions for getting the response to the menu options. It holds the values for several fields that map to corresponding `DV_TPT` elements used when getting a caller's response. These are used with the global termination conditions located in `def_dg_tpt`.

```
typedef struct DL_DATA {
    unsigned short da_recvdig; /* Number of digits to receive */
    unsigned short da_time; /* Maximum time limit */
    char * da_digit; /* Terminating digit string */
    unsigned short da_numretry; /* Number of retries before giving up */
}DL_DATA;
```

13. Voice Library Demo Programs

Field	Definition
da_recvdig	Receive Digits - The maximum number of digits to receive.
da_time	Time Limit - The amount of time a caller has to respond to the voice prompt. (100 ms units)
da_digit	Digit String - A string of specific digits, any one of which will terminate the I/O when received.
da_numretry	Number of Retries - The number of retries the caller has to press a correct digit.

In *d40demo.c*, the variable `data_1` defines the termination conditions for the root menu. The comments explain what each value indicates.

```
DL_DATA data_1 ={\n    1, /* Read only 1 digit */\n    100, /* Allow 10 seconds for entering it */\n    NULL, /* No termination digits needed */\n    3 /* Allow 3 retries for entry */\n};
```

menu_engine()

The `menu_engine()` function is defined in the *d4xtools.c* module. Its purpose is to control flow of the program by using the menu tables.

```
void menu_engine( )\n{\n    DL_MNUOPTS * rspentry;\n    DL_MENU * tmp;\n    char digbuf[2];\n\n    /* The following is the main menuing system engine. Its purpose\n    * is to drive the application as defined by the menu pointed to by\n    * cur_menu.\n    */\n    while(1) {\n\n        /*\n        * Execute the response parser\n        */\n        rspentry=(DL_MNUOPTS *)gt_data(digbuf, cur_menu->me_validfnc, cur_menu->me_prompt,\n            cur_menu->me_retry, cur_menu->me_error, cur_menu->me_data);
```

Voice Software Reference - Features Guide for Linux

```
/*
 * Execute the appropriate actions for the last response
 */

/* If a function is required execute it */
if(rsprentry->mo_fcnt != NULL) {
    (void) (*rsprentry->mo_fcnt)();
}

/*
 * If a new menu has been specified then switch to next menu
 * else switch to the default menu
 */

if(rsprentry->mo_nxtmenu != NULL) {
    /*
     * Make the current menu previous
     * and the previous menu current
     */
    pre_menu = cur_menu;
    cur_menu = rsprentry->mo_nxtmenu;
} else {

    /* Switch to the default menu */
    switch((int)cur_menu->me_defmenu) {
    case NULL:
        fprintf(stderr, "Missing default menu. Menu system error");
        exit(1);

    case (int)DM_PREV:
        /* Swap the current menu with the previous menu */
        tmp = pre_menu;
        pre_menu = cur_menu;
        cur_menu = tmp;
        break;

    case (int)DM_CURR:
        break;

    default:
        pre_menu = cur_menu;
        cur_menu = cur_menu->me_defmenu;
        break;
    }
}
}
```

In the while-loop, the first function call is to the system routine **gt_data()** (see the next section). The **gt_data()** function plays the voice prompt of the current menu, gets the caller's response, and returns the response as a **DL_MENUOPTS** structure. The **gt_data()** function validates the response and plays any error messages if the response is wrong. The value returned is guaranteed to be a valid response.

13. Voice Library Demo Programs

The `menu_engine()` function checks the `mo_fcnt` field of the returned structure for a value and calls the function if one exists.

`menu_engine()` then transfers control to the next menu indicated in the `mo_nxtmenu` field. If the value of the field is `NULL`, control is passed to the default menu, specified in the `me_defmenu` field. A value of `DM_PREV` in `me_defmenu` sets the current menu to be the previous menu, and loops back to the `gt_data()` call at the beginning of the loop. The value of `DM_CURR` loops back, leaving the current menu the same.

The `menu_engine()` function relies on other system routines defined in `d4xtools.c` for playing the prompts, handling errors, and validating responses.

`gt_data()`

The `gt_data()` function gets the caller's response to a voice prompt.

```
void      * gt_data(digbufp,validfnc,prompt,retry,error,datap)
char      * digbufp;
void      * (*validfnc)();
char      * prompt;
char      * retry;
void      (*error)();
DL_DATA  * datap;

{
  unsigned short  numretry;          /* Number of allowed retries */
  void            * rp;              /* general return ptr from valid fn */

  DV_DIGIT        digit;
  DV_TPT          dattpt[MAXTERMS];

  (void)memcpy(dattpt,def_dg_tpt,(MAXTERMS*sizeof(DV_TPT)));

  dattpt[DX_MAXDIME-1].tp_length=datap->da_recvdig;
  dattpt[DX_MAXTIME-1].tp_length=datap->da_time;
  dattpt[DX_DIGMASK-1].tp_length=blt_tdig_msk(datap->da_digit);

  numretry = datap->da_numretry;      /* Set the retry count */

  /*
   * Continue to prompt for this input until an acceptable response is
   * entered or the retry count has been exhausted
   */
  do {
```

Voice Software Reference - Features Guide for Linux

```
/*
 * Play the prompt message
 */
playmsg(prompt);
/*
 * Get the user's response (DTMF digits)
 */
if(dx_getdig(devhandle, (DV_TPT *)dattpt,&digit,EV_SYNC) == -1) {
    disperr("Failed on get digits");
}

/* Copy the digits to our buffer */
strcpy(digbufp,digit.dg_value);

check_term(); /* check for fatal errors)
              /* If a response was given, validate it */

/* NOSTRICT */
if ((rp= (*validfnc) (digbufp)) != NULL) {
    return (rp);
}

/* Not a valid response */
playmsg(retry);
} while (--numretry);

/*
 * No valid response within max. # of retries, so call fatal error
 * handler for this menu, if any provided, and then hang up and go idle.
 */
if (error)
    (*error) ();

longjmp(idlestate,1);

/* never gets here but makes lint happy */
return (NULL);
}
```

The **memcpy()** function call copies the default get digit DV_TPT array, **def_dg_tpt**, into the local structure, **dattpt**. The local DV_TPT array is redefined by using the values of the current menu's DL_DATA structure, **datap**, in the following code fragment:

```
dattpt [DX_MAXDTMF-1].tp_length=datap->da_recvdig;
dattpt [DX_MAXTIME-1].tp_length=datap->da_time;
dattpt [DX_DIGMASK-1].tp_length=bld_tdig_msk(datap->da_digit);
```

The caller's response to the voice prompt is received by the library call to **dx_getdig()** or **dx_getdigEx()**. The reason for termination is checked in the call to the system function **check_term()**.

13. Voice Library Demo Programs

The call to the function pointed to by `validfnc` validates the caller's response. If the response is invalid, the validation function returns a failure, the error message is played, and the caller is prompted to input another response. If the caller does not give a correct response within the maximum number of retries, the error message is played, and the process jumps back to the `idlestate`. The `idlestate` is defined in `main()` by the code:

```
main(argc,argv)
{
    .
    .
    (void)setjmp(idlestate);    /* where idlestate is defined      */
}
```

When the `longjmp(idlestate,1)` call is made in `gt_data()`, the control jumps to the statement directly after the `setjmp(idlestate)` call in `main()`.

check_term()

When an I/O function returns, the `check_term()` function is used to find out the reason for termination.

```
void check_term( )
{
    if (ATDX_TERMMSK(devhandle) & (TM_MAXNOSIL|TM_LCOFF|TM_PATTERN|TM_USRSTOP)) {
        longjmp(idlestat,1);    /*hang up&ready for new call */
    }
    return;
}
```

`check_term()` uses `ATDX_TERMMSK()` attribute to check the termination type. If the I/O function was terminated for any of the cases, the function calls a `longjmp()` and returns to `idlestate`. If none of these terminations occurred, the caller is still on the line, and the response must be validated.

13.1.8. Messaging System Routine

Concept

The messaging system provides a way to play voice prompts or digits by specifying the name. It implements the `dx_play()` function at a higher level. For example, to play the greeting message, a single argument to the `playmsg()` function is required:

```
playmsg("intro");
```

Implementation

The message playback is implemented by using two data structures, DL_MSGS and DL_MSGTBL, and by the following functions defined in *d4xtools.c*:

- **playmsg()**
- **plaympmsg()**
- **buildmsg()**
- **buildmsg_v()**
- **builddate()**
- **gt_groupiott()**
- **gt_iott()**
- **gt_numiott()**
- **gt_pairiott()**

The DL_MSGTBL structure provides the physical file name, the descriptor, and the device type for a file of messages. The initialization routine, **init_sys()**, will open the file and provide a file descriptor for each filename provided in the *mt_fn* field. The message system requires a NULL-terminated array of DL_MSGTBL structures.

```
typedef struct DL_MSGTBL {
    DL_MSGS    mt_msgsp; /* Pointer to a message structure*/
    char       mt_fn;   /* Pointer to file name */
    int        mt_fd;   /* File descriptor*/
    int        mt_typ  /* Type of the storage media*/
} DL_MSGS;
```

The DL_MSGS structure makes the location of the message transparent to the application by using the structure to assign an ASCII name to the message's offset and length. To signify the end of the messages, the message system requires a NULL-terminated DL_MSGS array for each DL_MSGTBL structure.

```
typedef struct dx_msgs {
    char * ms_msgname; /* Pointer to message name*/
    long  ms_offset;  /* Offset of this message - used in a DX_IOTT*/
    unsigned ms_lngth; /* Length of this message - used in a DX_IOTT*/
} DL_MSGS;
```

The message system uses the DL_MSGS and DL_MSGTBL structures to store the information needed for a DX_IOTT structure. The **buildmsg()**, **buildmsg_v()**, **builddate()**, **gt_iott()**, **gt_numiott()**, **gt_groupiott()**, and **gt_pairiott()** functions use the message structures to build the DX_IOTT structure. The

13. Voice Library Demo Programs

playmsg() and **playpmsg()** functions use the **DX_IOTT** structure to play the message. The code for the **playmsg()** function is:

```
void playmsg(msgp)
char * msgp;
{
    DX_IOTT iott[25];
    DX_IOTT * iottp;

    if (msgp[0] < ' '){          /* if it's an IOTT list          */
        iottp = (DX_IOTT *)msgp; /* just play it              */
    }else{                      /* else                      */
        buildmsg(iott,msgp,NULL); /* build IOTT list for      */
        iottp = iott;          /* named prompt of nmbr    */
    }

    if(dx_play(devhandle, (DX_IOTT *)iottp, (DV_TPT *)def_rp_tpt,NULL) == -1) {
        disperr("Error playing message");
    }
    check_term(&csb);          /* abort if hang-up condition */
}
```

The first if statement checks to see if the **msgp** parameter is either a **DX_IOTT** structure or an ASCII name. If it is an ASCII name, a call to **buildmsg()** is used to create a **DX_IOTT** structure from the **DL_MSG** structure named by **msgp**.

The **dx_play()** function plays the message using the **DX_IOTT** structure and the default record-play **DV_TPT** structure located in **def_rp_tcb**.

13.2. Other Synchronous Demos

The distribution media contains two other demos that provide examples of different synchronous voice applications. It is instructive to examine how each of the demos uses the system routines provided in *d4xtools.c*. These two demos, **custserv** and **horoscope**, use the same command line interface.

13.2.1. Custserv Demo

The **custserv** (customer service) demo transfers calls to the proper customer service representative based on the caller's input.

The **custserv** demo demonstrates the use of nested menus in a voice application using the menu system in *d4xtools.c*.

Running the custserv Demo Program

To use the custserv demo, first change to the directory containing the executable version of the demo program. To run the demo, type the demo executable name at the system prompt and use the channel name you wish to run the demo on as the argument. Detailed instructions are provided below.

To run the custserv demo:

1. Change to the directory containing the executable version of the program:

```
$ cd /usr/dialogic/demos/custserv
```

2. Type the following at the command line:

```
$ custserv [device name list]
```

The device name list is a list of one or more channels that the demo will use; for example:

```
$ custserv dxxxB1C1 dxxxB1C2
```

In the above example, the custserv demo uses channels 1 and 2 on board 1.

After the command line arguments are executed, the custserv demo presents you with the initial menu:

- press "1" for hardware support
- press "2" for software support
- press "3" for service and maintenance

If you press "1" from the main menu, the following submenu is played:

- press "1" for PC support
- press "2" for minicomputer support
- press "3" for mainframe support

If you press "2" from the main menu, the following submenu is played:

- press "1" for database support
- press "2" for spreadsheet support
- press "3" for word processing support

13. Voice Library Demo Programs

If you press a "3" from the main menu, or any number from the submenus, the demo plays a message to the caller. The message simulates a customer service representative and explains that, if this had been a real application, a live representative would have answered.

NOTE: If a demo is not terminated, the driver for the demos does not unload after quitting the program, necessitating a reboot before another demo can be loaded.

To terminate the custserv demo:

1. Run the ps command to determine the process ID.
2. Kill process_id.

13.2.2. Horoscope Demo

The horoscope program demonstrates methods of caller input. One of the important features it demonstrates is the chaining of separate messages together into a single playback message. This is implemented by using a DX_IOTT structure.

The horoscope demo asks you for your date of birth and plays a corresponding horoscope message.

Running the horoscope Demo Program

To use the horoscope demo, first change to the directory containing the executable version of the demo program. To run the demo, type the demo executable name at the system prompt and use the channel name you wish to run the demo on as the argument. Detailed instructions are provided below.

To run the horoscope demo:

1. Change to the directory containing the executable version of the program:

```
$ cd /usr/dialogic/demos/horoscop
```

2. Type the following at the command line:

```
$ horoscope [device name list]
```

Voice Software Reference - Features Guide for Linux

The device name list is a list of one or more channels that the demo will use; for example:

```
$ horoscope dxxxB1C1 dxxxB1C2
```

In the above example, horoscope demo uses channels 1 and 2 on board 1.

After the above command line arguments have been executed, the horoscope demo does the following:

- plays the welcome greeting
- asks you to enter the month they were born and to press "#" when finished
- asks you to enter the day they were born and to press "#" when finished
- asks you to enter the year they were born and to press "#" when finished
- plays a message saying the day of week, date, and year the caller was born
- plays a horoscope message

NOTE: If a demo is not terminated, the driver for the demos does not unload after quitting the program, necessitating a reboot before another demo can be loaded.

To terminate the horoscope demo:

1. Run the ps command to determine the process ID.
2. Kill process_id.

13.3. Asynchronous Demo Programs *pansr* and *cbansr*

The Voice Library asynchronous demo programs (*cbansr.c* and *pansr.c*) can be found in the `/usr/dialogic/demos/ansr` directory included with the voice software for Linux. These asynchronous demo programs work with all Dialogic SCbus-based products (including digital network interface boards). These demos incorporate examples of SCbus time slot routing.

The inbound asynchronous callback answer (*cbansr*) and inbound asynchronous polled mode answer demo (*pansr*) demos simulate an answering machine. They support all voice cards in SCbus and standalone mode, and digital T-1 robbed-bit or E-1.

13. Voice Library Demo Programs

The following is an example of a hardware setup for use with either asynchronous demo program:

- One VFX/40ESC, which should be downloaded as the front end (D41E_Resource = OFF)
- One Central Office Simulator (COS)
- One Analog phone.

Connect the phone to Line 1A of COS and connect Line 2 of the COS to the desired channel on the back of the board.

All of the command line arguments to this demos can be found by typing:

```
cbansr -?
```

or

```
pansr -?
```

Here are some switches for use with the cbansr and pansr demos:

- d d4xnum Number of the first D/4x board to use
- t dtinum Number of the first DTI board to use
- n N Number of D/4x channels to use, max: 12
- f frontend Analog, T1 or E1

To run the selected configuration:

1. Go to `/usr/dialogic/demos/ansr` and type the following arguments at the command line:

```
./cbansr -d1 -n4 -fAnalog
```

or

```
./pansr -d1 -n4 -fAnalog
```

2. The demos display a screen with the status of the channels and calls.
3. After the application is running, dial 5 to ring the configured channel.
4. After the voice prompt, leave a message and hang up.

5. Now Dial 5 again and enter the 4-digit extension number during voice prompt.
6. Listen for your message, ensuring it was recorded correctly and completely.
7. Following a successful demo run, recompile and rerun, taking note of any errors or warnings.
NOTE: Note: If a demo is not terminated, the driver for the demos does not unload after quitting the program, necessitating a reboot before another demo can be loaded.
8. Terminate the `cbansr` and `pansr` programs terminated by executing a “Ctrl C.”

13.4. Caveats

The demo programs are not fully implemented applications. They all contain limitations. For example, in `d40demo`, one of the limitations is that the database is not shared by all the channels. If a caller places an order on one channel, any attempts to cancel it on another channel will fail.

The purpose of the demo programs is to provide an example of coding a voice application. They are not intended to be used as real voice applications. The demos contain all the elements needed to write a voice application. However, a fully implemented voice application **can** be written based on techniques used in the `d40demo` source code and by using the code found in `d4xtools.c`.

Appendix A

Related Publications

For more information on related hardware and software products see the following Dialogic publications:

- For information about the voice programming libraries, see *Voice Software Reference: Programmer's Guide for Linux*
- For information on fax software development, see the *Fax Software Reference for Linux*
- For descriptions of the network programming libraries, see
Digital Network Interface Software Reference
Digital Audio Conferencing Software Reference
MSI/SC Software Reference
- For information on GlobalCall software development, see:
GlobalCall API Software Reference
GlobalCall ISDN Technology User's Guide
GlobalCall E-1/T-1 Technology User's Guide
GlobalCall Country-Dependent Parameters (CDP) Reference
GlobalCall Country-Dependent Parameters Reference for PDK Protocols
GlobalCall DPNSS ISDN Protocol Reference
GlobalCall Analog Technology User's Guide
- For information on ISDN software development, see the *ISDN Software Reference*
- For information on SCbus routing, see:
SCbus Routing Guide
SCbus Routing Software Reference for Linux
- For information on Dialogic boards, see also the respective *Quick Installation Card*.
- For information on the ITU-T G.726 Recommendation, see the following ITU-T publications available from the International Telecommunication Union (ITU), Place des Nations, CH-1211 Geneva 20, Switzerland. Tel: +41 22 730 5111. Fax: +41 22 733 7256. E-mail: itumail@itu.int.

Voice Software Reference - Features Guide for Linux

G.726 - 40, 32, 24, 16 kbit/s Adaptive Differential Pulse Code Modulation (ADPCM).

G.726A - Extensions of Recommendation G.726 on 40, 32, 24, 16 kbit/s ADPCM for use with uniform-quantized input and output.

G.726 III/G.727 II - Appendix III/II to Recommendation G.726/G.727 - Comparison of ADPCM algorithms.

Glossary

A-law: Pulse Code Modulation (PCM) algorithm used in digitizing telephone audio signals in E-1 areas. See also **Mu-law**.

Adaptive Differential Pulse Code Modulation: See **ADPCM**.

ADPCM (Adaptive Differential Pulse Code Modulation): A 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 64K bits/second to as low as 24K bits/second.

ADSI (Analog Display Services Interface): A Bellcore standard defining a protocol for the flow of information between a switch, a server, a voice mail system, a service bureau, or a similar device and a subscriber's telephone, PC, data terminal, or other communicating device with a screen. The idea of ADSI is to add words to a system that usually only uses touch tones. In a typical voice mail system, you call up and hear choices: "to listen to new messages, press 1, to hear saved messages, press 2," etc. ADSI is designed to display the choices you're hearing on a screen attached to your phone. ADSI's signaling is DTMF and standard Bell 202 modem signals from the service to your 202-modem-equipped phone. From the phone to the service it's only touch tone. ADSI works on every phone line in the world.

AGC (Automatic Gain Control): An electronic circuit used to maintain the audio signal volume at a constant level.

AMIS (Audio Messaging Interchange Specification): A series of standards aimed at addressing the problem of how voice messaging systems produced by different vendors can network or inter-network. It deals specifically with the interaction of the systems and does not affect the systems themselves. There are two specifications: 1. AMIS-digital: All the control information and the voice messages are ported between systems digitally. 2. AMIS-analog: Control information and messages are transferred in analog form. For AMIS specifications, call Hartfield Associates (Boulder, CO) at (303) 442-5395.

analog: 1. A method of telephony transmission in which the signals from the source (for example, speech in a human conversation) are converted into an electrical signal that varies continuously over a range of amplitude values analogous to the original signals (as opposed to digital signaling). 2. Used to refer to applications that use loop start signaling.

ANI: Automatic Number Identification.

API: See **Application Programming Interface**.

Application Programming Interface: A set of standard software interrupts, calls, and data formats that application programs use to initiate contact with network services, mainframe communications programs, or other program-to-program communications.

ASCIIZ string: A null-terminated string of ASCII characters.

asynchronous function: A function that allows program execution to continue without waiting for a task to complete. To implement an asynchronous function, an application-defined event handler must be enabled to trap and process the completed event. See **synchronous function**.

AT: An IBM or IBM-compatible Personal Computer (PC) containing an 80286 or higher microprocessor, a 16-bit bus architecture, and a compatible BIOS.

AT bus: The common communication channel in a PC AT. The channel uses a 16-bit data path architecture, which allows up to 16 bits of data transfer. This bus architecture includes the standard PC bus plus a set of 36 lines for additional data transmission, addressing, and interrupt request handling.

Automatic Gain Control: See **AGC**.

base memory address: A starting memory location (address) from which other addresses are referenced.

bit mask: A pattern which selects or ignores specific bits in a bit mapped control or status field.

bitmap: An entity of data (byte or word) in which individual bits contain independent control or status information.

board device: A board-level object that can be manipulated by a physical library. Board devices can be real physical boards, such as a D/4x board, or virtual devices, such as one of the D/4x boards that is emulated by a D/xxxSC board.

Board Locator Technology (BLT): Operates in conjunction with a rotary switch to determine and set nonconflicting slot and IRQ interrupt-level parameters, thus eliminating the need to set confusing jumpers or DIP switches.

buffer: A block of memory or temporary storage device that holds data until it can be processed. It is used to compensate for the difference in the rate of the flow of information (or time occurrence of events) when transmitting data from one device to another.

bus: An electronic path which allows communication between multiple points or devices in a system.

busy device: A device that is stopped, being configured, has a multitasking or nonmultitasking function, or I/O function active on it.

cadence: A rhythmic sequence or pattern. Once established, it can be classified as a single ring, a double ring, or a busy signal by comparing the periods of sound and silence to establish parameters.

cadence detection: A voice driver feature that analyzes the audio signal on the line to detect a repeating pattern of sound and silence.

Call Progress Analysis: The process used to automatically determine what happened after an outgoing call is dialed. Also referred to as call analysis or call progress.

Call Status Transition Event Functions: Functions that set and monitor events on devices.

CCITT (Comite Consultatif Internationale de Telegraphique et Telephonique): One of the four permanent parts of the International Telecommunications Union, a United Nations agency based in Geneva. The CCITT is divided into three sections: 1. Study Groups set up standards for telecommunications equipment, systems, networks, and services. 2. Plan Committees develop general plans for the evolution of networks and services. 3. Specialized Autonomous Groups produce handbooks, strategies, and case studies to support developing countries.

Central Office: See **CO**.

channel device: A channel-level object that can be manipulated by a physical library, such as an individual telephone line connection. A channel is also a subdevice of a board. See **subdevice**.

channel: 1. When used in reference to a Dialogic 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 a 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.

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 that lets you access or transmit voice, fax, and e-mail messages from a single point; voice mail and voice messaging; fax systems including fax broadcasting, fax mailboxes, fax-on-demand, and fax gateways; transaction processing such as Audiotex and Pay-Per-Call information systems; 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.

Configuration Functions: Functions that alter the configuration of devices.

Convenience Functions: Functions that simplify application writing.

data structure: Programming term for a data element consisting of fields, where each field may have a different type definition and length. A group of data structure elements usually share a common purpose or functionality.

debouncing: Eliminating false signal detection by filtering out rapid signal changes. Any detected signal change must last for the minimum duration as specified by the debounce parameters before the signal is considered valid. Also known as deglitching.

deglitching: See **debouncing**.

device: A computer peripheral or component controlled through a software device driver. A Dialogic voice and/or network interface expansion board is a board containing one or more logical board devices. Each channel or time slot on the board is considered a device.

device channel: A Dialogic voice data path that processes one incoming or outgoing call at a time (equivalent to the terminal equipment terminating a phone line). There are 4 device channels on a D/4x, 8 on a D/80SC, 16 on a D/160SC, 24 on a D/240SC, 30 on a D/300SC, 32 on a D/320SC, 48 on a D/480SC-2T1, and 60 on a D/600SC-2E1 board. See also **time slot**.

device driver: Software that acts as an interface between an application and hardware devices.

device handle: Numerical reference to a device, obtained when a device is opened using `xx_open()`, where `xx` is the prefix defining the device to be opened. The device handle is used for all operations on that device.

Device Management Functions: Functions that open and close devices.

Device name: Literal reference to a device, used to gain access to the device via an `xx_open()` function, where `xx` is the prefix defining the device to be opened.

DIALOG/HD (Dialogic High Density) boards: Dialogic voice and telephone network interface resource boards having 8 or more ports. All DIALOG/HD boards use Board Locator Technology and can communicate via the SCbus.

Digital signal processor: see **DSP**.

digitize: The process of converting an analog waveform into a digital data set.

download: The process where board level program instructions and routines are loaded during board initialization to a reserved section of shared RAM.

downloadable SpringWare firmware: Software features loaded to Dialogic voice hardware. Features include voice recording and playback, enhanced voice coding, tone detection, tone generation, dialing, call progress analysis, voice detection, answering machine detection, speed control, volume control, ADSI support, automatic gain control, and silence detection.

driver: A software module which provides a defined interface between an application program and the firmware interface.

DSP: 1. Digital signal processor. A specialized microprocessor designed to perform speedy and complex operations upon digital signals. 2. Digital signal processing.

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. The tone frequencies are, in Hz:

1: 697, 1209	2: 697, 1336	3: 697, 1477
4: 770, 1209	5: 770, 1336	6: 770, 1477
7: 852, 1209	8: 852, 1336	9: 852, 1477
0: 941, 1336	*: 941, 1209	#: 941, 1477

DualSpan board: A Dialogic voice board that has connections for two E-1 or T-1 lines.

E-1: A CEPT digital telephony format devised by the CCITT. A digital transmission channel that carries data at the rate of 2.048 Mbps (DS-1 level).

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.

echo-cancelled signal: The output signal of an echo canceller after echo has been removed from the echo-carrying signal.

echo canceller: The software component responsible for performing echo cancellation.

echo-carrying signal: In regard to the **echo canceller**, a signal containing incoming speech data plus an echo component.

echo-producing circuit: Typically the interface between 4-wire (typically digital) and 2-wire (typically analog) circuits, which, due to impedance mismatches, reflects part of the receive signal into the transmit signal.

echo reference signal: The signal that initially introduced echo into the echo-carrying signal. This signal is used by the **echo canceller** to estimate the echo component in the echo carrying signal.

ECR: Dialogic Echo Cancellation Resource, consisting of three Voice library function APIs for implementing echo cancellation on a Dialogic voice channel device.

ECR mode: The operational mode for a Dialogic voice channel utilizing the Dialogic ECR feature at its highest level.

ECR_RX: The receive signal of the voice channel device's **echo canceller** containing the echo reference signal.

ECR_TX: The transmit signal produced by the voice channel device's **echo canceller** containing the echo-cancelled 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 a Dialogic application program designed to trap and control processing of device-specific events. The rules for creating a DTI/2xx event handler, for example, are the same as those for creating a Linux signal handler.

Event Management functions: A class of device-independent functions (contained in the Standard Runtime Library) that connect events to application-specified event handlers, allowing users to retrieve and handle events that occur on the device. See **Standard Runtime Library**.

Extended Attribute functions: 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 **Standard Runtime Library**.

firmware load file: The firmware file that is downloaded to a Dialogic voice board. This file has an *.fwl* extension.

flash: A signal generated by a momentary on-hook condition. This signal is used by the voice hardware to alert a telephone switch that special instructions will follow. It usually initiates a call transfer. See also **hook state**.

frequency detection: A voice driver feature that detects the tri-tone Special Information Tone (SIT) sequences and other single-frequency tones for call progress analysis.

G.726: An ITU-T audio coding and decoding recommendation that defines the characteristics for conversion of a 64 Kbps A-law or μ -law PCM channel at 8000 samples per second to and from a 40, 32, 24, or 16 Kbps channel. The conversion is applied to the PCM stream using an adaptive differential pulse code modulation (ADPCM) transcoding technique. G.726 is useful for applications that require speech compression, encoding for noise immunity, and uniformity in transmitting voice and data signals.

Global Dial Pulse Detection (Global DPD): A Dialogic SpringWare firmware enhancement that enables computer telephony (CT) applications to respond to caller input originating from rotary or pulse dialing telephones.

Global Tone Detection: A feature that allows the creation and detection of user-defined tone descriptions on a channel-by-channel basis.

hook state: The current line status of the channel: either on-hook or off-hook. A telephone station is said to be on-hook when the conductor loop between the station and the switch is open and no current is flowing. When the loop is closed and current is flowing, the station is off-hook. These terms are derived from the position of the old fashioned telephone set receiver in relation to the mounting hook provided for it.

hook switch: The circuitry that controls the on-hook and off-hook state of the voice device telephone interface.

hybrid: See **echo-producing circuit**.

I/O Functions: Functions that transfer data to and from devices.

I/O: Input-Output.

idle device: A device that has no functions active on it.

In-band: The use of robbed-bit signaling (T-1 systems only) on the network. The signaling for a particular channel or time slot is carried within the voice samples for that time slot, thus within the 64 kbps (kilobits per second) voice bandwidth.

in-band signaling: (1) In an analog telephony circuit, in-band refers to signaling that occupies the same transmission path and frequency band used to transmit voice tones. (2) In digital telephony, "in-band" means signaling transmitted within an 8-bit voice sample or time slot, as in T-1 "robbed-bit" signaling.

Interrupt Request Level (IRL): Same as IRQ (See **Interrupt Request**).

Interrupt Request (IRQ): A signal sent to the central processing unit to temporarily suspend normal processing and transfer control to an interrupt handling routine. Interrupts may be generated by conditions such as completion of an I/O process, detection of hardware failure, and power failures.

kernel: A set of programs in an operating system that implement the system's basic functions.

loop: The physical circuit between the telephone switch and the voice processing board.

loop current: The current that flows through the circuit from the telephone switch when the voice device is off-hook.

loop current detection: A voice driver feature that returns a connect after detecting a loop current drop.

loop start: An electrical circuit consisting of two wires (or leads) called tip and ring, which are the two conductors of a telephone cable pair in an analog environment. The CO provides voltage (called "talk battery" or just "battery") to power the line. When the circuit is complete, this voltage produces a current called loop current. The circuit provides a method of starting (seizing) a telephone line or trunk by sending a supervisory signal (going off-hook) to the CO.

loop-start interfaces: Devices, such as an analog telephones, that receive an analog electric current. For example, taking the receiver off-hook closes the current loop and initiates the calling process.

LSI board: a Dialogic loop-start interface expansion board.

MSI/SC board: a Dialogic Modular Station Interface board for connecting devices to station devices (phones), with conferencing support.

Mu-law: (1) Pulse Code Modulation (PCM) algorithm used in digitizing telephone audio signals in T-1 areas. (2) The PCM coding and companding standard used in Japan and North America. See also **A-law**.

NLP: Non Linear Processor. Operates on the output of the **echo canceller** to provide improved echo suppression as long as the echo reference signal contains a speech signals and the echo-carrying signal does **not**.

off-hook: The state of a telephone station when the conductor loop between the station and the switch is closed and current is flowing. When a telephone handset is lifted from its cradle (or an equivalent condition occurs), the telephone line state is said to be off-hook. See also **hook state**.

on-hook: Condition or state of a telephone line when a handset on the line is returned to its cradle (or an equivalent condition occurs). See also **hook state**.

PC: Personal Computer. Refers to an IBM Personal Computer or compatible machine.

PCM: See **Pulse Code Modulation**.

polling: The process of repeatedly checking the status of a resource to determine when state changes occur.

polling functions: Voice Library functions used to check the current status of a voice device. Polling functions are also used to examine the number and configuration of devices in the system and to detect when events occur on a device.

Pulse Code Modulation (PCM): A 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.

resource: Functionality (e.g. 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.

robbed-bit signaling: The type of signaling protocol implemented in areas using the T-1 telephony standard. In robbed-bit signaling, signaling information is carried in-band, within the 8-bit voice samples. These bits are later stripped away, or "robbed," to produce the signaling information for each of the 24 time slots.

routing functions: Functions that assign analog and digital channels to specific SCbus time slots; these SCbus time slots can then be connected to transmit or listen to other SCbus time slots.

sampling rate: Frequency with which a digitizer takes measurements of the analog voice signal.

SCbus (Signal Computing Bus): A TDM (Time Division Multiplexed) bus that connects SCSA (Signal Computing System Architecture) resources. It allows audio, signaling, and control information to be transmitted and received among these resources. Also, a hardwired connection between Switch Handlers (SC2000 chips) on SCbus-based products for transmitting information over 1024 time slots to all devices connected to the SCbus.

SCbus routing functions: Functions that enable an application to connect or disconnect (make or break) the receive (listen) channel of a device to or from an SCbus time slot.

SCR: See **Silence Compressed Record**.

SCSA: See **Signal Computing System Architecture**.

Signal Computing System Architecture (SCSA): A Dialogic standard open development platform. An open hardware and software standard that incorporates virtually every other standard in PC-based switching. All signaling is out of band. In addition, SCSA offers time slot bundling and allows for scalability.

signaling insertion: The signaling information (on hook/off hook) associated with each channel is digitized, inserted into the bit stream of each time slot by the device driver, and transmitted across the bus to another resource device. The network interface device generates the outgoing signaling information.

Silence Compressed Record (SCR): A recording that eliminates or limits the amount of silence in the recording without dropping the beginning of words that activate recording.

silence threshold: The level that sets whether incoming data to the voice board is recognized as silence or nonsilence.

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. For example, for `dx_play()`, the solicited event is "play complete."

Special Information Tones: See **SIT**.

speed and volume control: Voice software that contains functions and data structures to control the speed and volume of play on a channel. The end user controls the speed or volume of a message by entering a DTMF tone.

speed and volume modification table: Each channel on a voice board has a table with 20 entries that allow for a maximum of 10 increases and decreases in speed or volume, and 1 "origin" entry that represents regular speed or volume.

SpringWare: Software algorithms build into the downloadable firmware that provides the voice processing features available on all Dialogic voice boards.

SRL: See **Standard Runtime Library**.

Standard Attribute functions: Class of functions that take one input parameter (a valid Dialogic device handle) and return generic information about the device. For instance, Standard Attribute functions return IRQ and error information for all device types. Standard Attribute function names are case-sensitive and must be in capital letters. Standard Attribute functions for all Dialogic devices are contained in the Dialogic SRL. See **Standard Runtime Library**.

Standard Information Tones: See **SIT**.

Standard Runtime Library (SRL): A Dialogic software resource containing Event-Management and Standard Attribute functions and data structures used by all Dialogic devices, but which return data unique to the device. See the *Standard Runtime Library Programmer's Guide for Linux*.

station device: Any analog telephone or telephony device (such as a telephone or headset) that uses a loop-start interface and connects to an MSI/SC 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.

SVP mode: The standard voice processing mode for a Dialogic voice channel, where all standard Dialogic voice channel features are enabled and the default echo cancellation is provided.

synchronous function: Blocks program execution until a value is returned by the device. Also called a blocking function. See **asynchronous function**.

System Release Development Package: The software and user documentation provided by Dialogic that is required to develop applications.

T-1: A digital telephony format used in North America and Japan. In T-1, 24 voice conversations are time-division multiplexed into a single digital data stream containing 24 time slots. Signaling data are carried "in-band"; as all available time slots are used for conversations, signaling bits are substituted for voice bits in certain frames. Hardware at the receiving end must use the "robbed-bit" technique for extracting signaling information. T-1 carries data at the rate of 1.544 Mbps (DS-1 level).

tap: The echo tail length that the echo canceller handles in the ECR mode is 16 ms, also referred to as 128 tap. **SVP mode** echo cancellation utilizes a 48 tap (6 ms) **echo canceller**.

TDM: See **Time Division Multiplexing**.

termination condition: An event or condition which, when present, causes a process to stop.

termination event: An event that is generated when an asynchronous function terminates. See **asynchronous function**.

Time Division Multiplexing (TDM): 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 **time slot**.

time slot: In a digital telephony environment, a normally continuous and individual communication (for example, someone speaking on a telephone) is (1) digitized, (2) broken up into pieces consisting of a fixed number of bits, (3) combined with pieces of other individual communications in a regularly repeating, timed sequence (multiplexed), and (4) transmitted serially over a single telephone line. Each individual pieced-together communication is called a time slot.

time slot assignment: The ability to route the digital information contained in a time slot to a specific analog or digital channel on an expansion board. See **device channel**.

training period: When used in reference to echo cancellation, the period after enabling echo cancellation during which the **echo canceller** develops an estimate of the echo component contained in the echo-carrying signal.

Transaction Record feature: Permits recording two SCbus time slots from a single channel.

Universal Dialogic Diagnostic program: Software diagnostic routines for testing board-level functions of Dialogic hardware.

VFX: Any Dialogic combined voice, fax, and network interface board, such as the or VFX/40ESC.

Voice hardware and software: Expansion boards and associated software that support voice processing.

voice processing: Converting human voice into data that can be reconstructed and played back at a later time.

Voice system: A combination of expansion boards and software that lets you develop and run voice processing applications.

wink: In T-1 or E-1 systems, a signaling bit transition from on to off, or off to on, and back again to the original state. In T-1 systems, the wink signal can be transmitted on either the A or B signaling bit. In E-1 systems, the wink signal can be transmitted on either the A, B, C, or D signaling bit. Using either system, the choice of signaling bit and wink polarity (on-off-on or off-on-off hook) is configurable through DTI/xxx board download parameters.

Index

A

Adaptive Differential Pulse Code Modulation, G.726, 10, 131

Address field, Generic Configuration File, 5

address signals, 67

adjusting speed and volume. See speed and volume control

ADPCM, G.726, 10, 131

ADSI. *See* Analog Display Services Interface (ADSI)

Analog Display Services Interface (ADSI), 8, 83, 85

answering machine detection, 46

Antares, 2

ATDX_ANSRSIZ(), 22, 40, 42, 43

ATDX_CONNTYPE()
cadence detection, 43
Call Analysis results, 22
connection types returned, 45, 46, 49
Loop Current Detection, 49

ATDX_CPERROR(), 22

ATDX_CPTERM(), 18, 22

ATDX_CRTNID(), 22, 46

ATDX_DTNFAIL(), 22, 44

ATDX_FRQDUR(), 22, 29

ATDX_FRQDUR2(), 22, 29

ATDX_FRQDUR3(), 22, 29

ATDX_FRQHZ(), 22, 28

ATDX_FRQHZ2(), 22, 29

ATDX_FRQHZ3(), 22, 29

ATDX_FRQOUT(), 22, 31

ATDX_LONGLOW(), 43

ATDX_SHORTLOW(), 22, 36, 42

ATDX_SIZEHI(), 22, 42

ATDX_TERMMSK(), 64

audio cadences, 32

B

backward signals (CCITT signaling system tones)
definition, 67
Group A and B, 71, 73, 74
overview, 67
R2 MF tones, 68

board
Name fields, 5

boards
emulated, 3
naming conventions, 1
virtual, 3

buffer size adjustments for Internet telephony, 121

Busy, 36, 46

C

ca_ansrdgl, 41, 42, 49

ca_cnosig, 46

ca_dtn_deboff, 44, 45

Voice Software Reference - Features Guide for Linux

- ca_dtn_pres, 44
- ca_hedge, 41, 42, 49
- ca_hightch, 38
- ca_intflag, 23
- ca_intflg, 24
- ca_lcdly, 48
- ca_lcdly1, 49
- ca_logltch, 38
- ca_lower2frq, 27
- ca_lower3frq, 28
- ca_lowerfrq, 27, 30
- ca_maxansr, 42, 49
- ca_mvertime3frq, 28
- ca_mvertimefrq, 27
- ca_nbrdna, 39
- ca_noanswer, 46
- ca_nsbusy, 40
- CA_PAMD Speed Value, 47
- CA_PAMD_fail time, 47
- ca_pamd_failtime, 47
- CA_PAMD_minimum allowable ring, 48
- ca_pamd_minring, 48
- ca_pamd_qtemp, 47
- CA_PAMD_QUALITMP, 47
- ca_pamd_spdval, 47
- ca_stdely
 - Call Analysis cadence detection, 26, 30, 37
 - Call Analysis positive voice detection, 26, 30
 - Call Analysis SIT tone detection, 26, 30
 - definition, 45
 - settings, 26, 30, 37
- ca_time2frq, 27
- ca_time3frq, 28
- ca_timefrq, 27, 30, 31
- ca_upper2frq, 27
- ca_upper3frq, 28
- ca_upperfrq, 27, 30
- cadence, 33, 34
- cadence detection
 - ATDX_ANSRSIZ(), 43
 - ATDX_CONNTYPE(), 43
 - ATDX_LONGLOW(), 34, 43
 - ATDX_SHORTLOW(), 34, 42
 - ATDX_SIZEHI(), 34, 42
 - Busy, 36
 - ca_ansrdgl, 41
 - ca_cnosig, 39
 - ca_cnosil, 39
 - ca_hightch, 38
 - ca_logltch, 38
 - ca_maxansr, 42
 - ca_nbrdna, 39
 - ca_nsbusy, 40
 - ca_stdely, 37
 - Call Analysis, 14, 31, 43, 49
 - Connect, 36
 - DX_CAP parameters, 26, 30, 37
 - Extended Attributes, 34
 - high glitch, 38
 - low glitch, 38
 - No Answer, 36
 - No Ringback, 36
 - nonsilence, 34
 - outcomes, 34

- parameters, 32, 36, 37
- parameters affecting Busy, 39
- parameters affecting Connect, 40
- parameters affecting No Answer, 39
- parameters affecting No Ringback, 38, 39
- salutation processing, 42
- start delay, 37
- cadence patterns
 - double ring (figure), 33
 - standard busy signal, 33
 - standard single ring, 33
 - typical, 33
- Call Analysis. *See also* Cadence
 - Detection; PerfectCall Call Analysis
 - ATDX_CPEERROR(), 49
 - ATDX_CPTERM(), 18, 49
 - Basic, 13
 - call outcomes, 19, 20
 - call outcomes (busy, connect, no answer, no ringback, operator intercept), 13
 - components, 15
 - description, 13, 14
 - dial tone, 44
 - DX_CAP, 17, 18
 - dx_dial(), 17, 19
 - Enhanced (PerfectCall), 7, 13
 - errors, 49
 - Extended Attribute functions, 22
 - frequency detection, 14, 24
 - DX_CAP parameters, 25, 26
 - errors, 30
 - range, 24
 - SIT tones, 24, 25, 26, 27, 28, 29
 - how to use, 17
 - initiating, 19
 - introduction, 7
 - Loop Current Detection, 14, 23
 - obtaining additional information, 22
 - parameter structure, 17
 - PerfectCall (Enhanced), 13, 15, 43
 - positive voice detection, 14, 23, 26, 30
 - results
 - answer duration, 22
 - connection type, 22
 - conntype, 45
 - dial tone failure, 22
 - error, 22
 - first frequency duration, 22
 - frequency detection, 22
 - frequency out of bounds, 22
 - last termination, 22
 - non-silence, 22
 - second frequency duration, 22
 - shorter silence, 22
 - third frequency duration, 22
 - tone identifier, 22, 97
 - tone_id, 46
 - SIT tone detection, DX_CAP
 - parameters, 25, 26
 - termination results, 19
- call outcome, determining, 19
- Call Progress Characterization Utility, 37
- Caller ID
 - accessing information, 99
 - enabling, 100, 101
 - error handling, 101
 - introduction, 9
 - overview, 97
 - related references, 98
 - supported formats, 97
- CCITT Signaling System R2 MF tones, 68
- channel names, 5
- Channel Parameter Block, ca_dtn_pres, 44
- check_term(), 153
- CLASS, 97, 98

Voice Software Reference - Features Guide for Linux

compelled signaling, 79

CON_CAD, 45

CON_PAMD, 46

connect, 36

Continuous No Signal, 46

controlling speed and volume. See speed and volume control

Convenience Functions

- dx_wtcallid(), 100
- Speed and Volume, 89

CPC Utility, 37

CR_BUSY, 19

CR_CEPT, 19

CR_CNCT, 19

CR_ERROR, 19

CR_FAXTONE, 19

CR_LGTUERR, 50

CR_MEMERR, 49

CR_MXFRQERR, 50

CR_NOANS, 19

CR_NODIALTONE, 19

CR_NORB, 19

CR_OVRLPERR, 50

CR_STOPD, 19

CR_TMOUTOFF, 50

CR_TMOUTON, 49

CR_UNEXPTN, 50

CR_UPFRQERR, 50

Custom Local Area Signaling Services, 97, 98

custserv, 133

D

D/2x, naming, 5, 6

D/4x, naming, 5, 6

D/8x, naming, 5, 6

d40demo

- global variables, 140

- included with the voice software for Linux, 133

- initialization, 140

- menu system routine, 143

 - check_term(), 153

 - DL_DATA, 148

 - DL_MENU, 144

 - DL_MENUPTS, 146

 - gt_data(), 151

 - implementation, 144

 - menu_engine(), 149

 - model, 143

- messaging system routine, 153

- outline, 139

- program overview, 134, 138, 139, 155

DDI (Direct Dialing-In) service, 68

demo programs, 11. See also d40demo

- asynchronous callback answer, 157

- asynchronous polled mode answer, 157

- caveats, 159

- cbansr, 157

- custserv, 155, 157

- d40demo, 138, 155

- directory structure, 133

- Global DPD, 104

- horoscope, 156

- pansr, 157

- physical connection, 134

- running, 137, 155, 156

- synchronous, 133, 159

- Detection
 - answering machine, 46
 - dial tone, 44
 - Fax machine or modem, 48
- device names, 5
 - board, 5
 - channel, 5
 - D/2x, 5
 - D/41ESC, 5
 - D/4x, 5
 - DIALOG/HD, 5
- Dial Pulse Detection. *See* Global Dial Pulse Detection (Global DPD)
- Dial tone
 - detection, 44
 - international, 44
 - local, 44
 - special, 44
- Dial Tone Debounce, 44, 45
- Dial Tone Not Present, 44
- Dial Tone Present, 44
- Dialed Number Identification Service, 68
- DIALOG/HD boards, 2, 5, 7
- digits for adjusting play, 95
- Direct Dialing-In (DDI) service, 68
- Disconnect Supervision, 62
- DL_DATA. *See* d40demo
- DL_MENU. *See* d40demo
- DL_MNUOPTS. *See* d40demo
- DNIS (Dialed Number Identification Service), 68
- DTMF detection, 103
- dtnfail, 44
- dx_addspddig(), 89
- dx_addtone(), 53, 56, 63
- dx_addvoldig(), 89
- dx_adjsv(), 89, 96
- dx_blddt(), 54, 55
- dx_blddtcad(), 54
- dx_bldst(), 54, 55
- dx_bldstcad(), 54, 55
- dx_bldtngen(), 64
- DX_CAP. *See* also Call Analysis
 - ca_cnosig, 46
 - ca_dtn_deboff, 44, 45
 - ca_dtn_npres, 44
 - ca_maxintering, 46
 - ca_pamd_failtime, 47
 - ca_pamd_minring, 48
 - ca_pamd_qtemp, 47
 - ca_pamd_spdval, 47
 - ca_stdely, 45
 - Continuous No Signal, 46
 - Dial Tone Debounce, 44, 45
 - Dial Tone Not Present, 44
 - Dial Tone Present, 44
 - Maximum Inter-ring, 46
 - No Answer, 46
 - PAMD fail time, 47
 - PAMD minimum allowable ring, 48
 - PAMD Qualification Template, 47
 - PAMD Speed Value, 47
 - parameters used by Call Analysis, 18
 - Start Delay, 45
- dx_chgdur(), 16
- dx_chgfreq(), 16
- dx_chgrepcnt(), 16
- dx_clrcap(), 18

Voice Software Reference - Features Guide for Linux

`dx_deltone()`, 16, 17, 56
`dx_dial()`, 17, 18, 19
`dx_distone()`, 56
`dx_enbtone()`, 57
`dx_getdig()`, 53, 95
`dx_getdigEx()`, 95
`dx_getsvmt()`, 90
`dx_getxmitslot()`, 119
`dx_getxmitslotecr()`, 119
`dx_gtextcallid()`, 100
`dx_initcallp()`, 16
`dx_listen()`, 119
`DX_OPTDIS`, 24
`DX_OPTEN`, 24
`DX_OPTNOCON`, 24
`DX_PAMDENABLE`, 24, 46
`DX_PAMDOPTEN`, 24, 46
`dx_playiottdata()`, 131
`dx_playtone()`, 64
`DX_PVDENABLE`, 24
`DX_PVDOPTEN`, 24
`DX_PVDOPTNOCON`, 24
`dx_rec()`, 115
`dx_reciottdata()`, 131
`dx_recm()`, 113
`dx_recmf()`, 113
`dx_setevtmsk()`, 57
`dx_setgtdamp()`, 54, 55

`dx_setparm()` for enabling Caller ID,
101
`dx_setsvcond()`, 89, 95
`dx_setsvmt()`, 90, 95, 96
`DX_SVCB`, 95
`DX_SVMT`, 95, 96
`dx_unlistenecr()`, 119
`dx_wtcallid()`, 100
`DX_XPB`, 131

E

Echo Cancellation Resource (ECR)
bridge, 122
buffer size adjustments for Internet
telephony, 121
echo component, 117
echo reference signal, 117
echo-carrying signal, 117
introduction, 10
mode, 120
overview, 117
Echo canceller, 117
Echo component, 117
Echo reference signal, 117
Echo-carrying signal, 117
ECR. *See* Echo Cancellation Resource
(ECR)
ECR bridge, 122
enabling Caller ID, 101
encoding algorithm, G.726, 131
encoding algorithms, support in SCR,
115
error handling in Caller ID, 101

F

- Fax machine detection, 48
- forward signals (CCITT signaling system tones), 67, 68, 71, 73
- forward signals (CCITT signaling system tones), 74
- frequency detection using Call Analysis, 14, 24. *See also* Call Analysis
- frequency, lower, 30
- frequency, minimum time, 30, 31
- frequency, upper, 30

G

- G.726, 10, 131
- GDPD. *See* Global Dial Pulse Detection (Global DPD)
- Generic Configuration File
 - Address field, 5
 - file format, 4
 - Ioport field, 6, 7
 - Name field, 5, 6
 - Nchn field, 6
- Global Dial Pulse Detection (Global DPD)
 - API, 104
 - demonstration, 104
 - features, 103
 - improving detection, 111
 - overview, 9, 103
 - parameters, 103
 - programming, 105
 - supported applications, 103
- Global DPD. *See* Global Dial Pulse Detection (Global DPD)
- Global Tone Detection
 - applications, 62

- building tone templates, 54
- Call Analysis memory usage, 29
- define dual frequency cadence tone, 55
- define dual frequency tone, 55
- define single frequency cadence tone, 55
- define single frequency tone, 55
- defining tones, 53
- definition, 53
- disconnect supervision, 62
- functions with which GTD cannot work, 63
- introduction, 7
- leading edge detection, 63
- maximum number of tones, 62
- overview, 53
- R2 MF, 81
- retrieving tone events, 57
- set amplitude, 55
- use with Global DPD, 103
- using with PBX, 63
- with Caller ID, 97

Global Tone Generation

- data structures, 65
- definition, 64
- introduction, 7
- overview, 53
- R2 MF, 81
- template, 64
- TN_GEN, 64
- with Caller ID, 97

Group A and B signals, 71, 73, 74

Group I and II signals, 71, 73, 74

gt_data(), 151

GTD tones, defining. *See* Global Tone Detection**H**

High Density boards, terminology, 2

Voice Software Reference - Features Guide for Linux

horoscope, 133

I

incoming register, 67, 73

incoming signals, indicating, 73

Internet telephony, buffer size
adjustments, 121

interregister signals, 67

Ioport field, Generic Configuration File,
6, 7

L

leading edge detection using debounce
time, 63

line signals, 67

Loop Current Detection. *See also* Call
Analysis
ca_lcdly, 48
ca_lcdly1, 49
parameters affecting a connect, 48
parameters ignored, 49
use in Call Analysis, 14
use in PerfectCall Call Analysis, 48,
49

M

Maximum Inter-ring, 46

maxintering, 46

memory requirements, R2 MF, 82

menu_engine(). *See* d40demo

mode parameter, 84

Modem detection, 48

N

Name field in configuration files, 5, 6

Nchn field, in Generic Configuration
File, 6

No Answer, 36, 46

No Ringback, 36

nonsilence, 32

O

operator intercept SIT tones, 24, 25, 26,
27, 28

outgoing register, 67

P

PAMD, 46

PAMD fail time, 47

PAMD minimum allowable ring, 48

PAMD Qualification Template, 47

PAMD_FULL, 47

PAMD_QUICK, 47

parameter files, voice.prm, 115

PerfectCall Call Analysis
activating, 15
ATDX_CONNTYPE, 46
Busy, 46
ca_dtn_npres, ca_dtn_npres, 44
components, 15
dial tone detection, 44
disabling, 16
fax machine detection, 48
modem detection, 48
outcomes, 20
overview, 13
Positive Answering Machine
Detection, 46
ringback, 45
tone definitions, 16
tone detection, 43
tone types, 43

play, specifying mode, 84, 85
 PM_ADSI, 85
 Positive Answering Machine Detection,
 14, 46
 positive voice detection using Call
 Analysis, 14, 49. *See also* Call
 Analysis
 protocol, VPIM, 10, 131

R

R2 MF signaling
 address signals, 67
 and Global Tone
 Detection/Generation, 81
 backward signals (CCITT signaling
 system tones), 67, 68, 71,
 73, 74
 compelled signaling, 79
 definition, 8
 Dialed Number Identification
 Service, 68
 Dialogic support, 81
 direct dialing-in service, 68
 forward signals (CCITT signaling
 system tones), 67, 68, 71,
 73, 74
 Group A and B signals, 74
 Group I and II signals, 71, 73, 74
 incoming register, 67
 interregister signals, 67
 line signals, 67
 maximum number of tones, 82
 multifrequency combinations, 68
 outgoing register, 67
 outgoing signals, 67
 overview, 67
 related publications, 82
 signal meanings, 71
 Voice board support, 81
 r2_creatfsig(), 81

r2_playbsig(), 81
 recording with silence compression, 9,
 115
 related publications, 161
 Ringback detection, 45

S

SCR, 9. *See* silence compressed record
 silence compressed record, 9, 115
 SIT tones, 27, 28, 29. *See also* operator
 intercept SIT tones
 detection
 affect on GTD tones, 29
 lower frequency (tone 1), 27, 30
 lower frequency (tone 2), 27
 lower frequency (tone 3), 28
 memory usage, 29
 start delay, 26, 30
 time (tone 1), 30, 31
 time of frequency (tone 1), 27
 time of frequency (tone 2), 27
 time of frequency (tone 3), 28
 upper frequency (tone 1), 27, 30
 upper frequency (tone 2), 27
 upper frequency (tone 3), 28
 using Call Analysis, 24, 25, 26,
 27, 28
 using Extended Attribute
 functions, 28
 frequency information, 31
 Special Information Tones. *See* SIT
 tones
 speed. *See* speed and volume control
 speed and volume control
 adjustment digits, 95
 adjustment functions, 89
 Convenience functions, 89
 explicitly adjusting, 95
 introduction, 8

Voice Software Reference - Features Guide for Linux

- modification tables, 90, 91
- overview, 89
- setting adjustment conditions, 95

Standard Voice Processing (SVP)
mode, 119

Start Delay, 45

synchronous demo program, 133, 159

T

talk-off rejection, 111

TID_BUSY1, 43, 46

TID_BUSY2, 43, 46

TID_DIAL_INTL, 43

TID_DIAL_LCL, 43

TID_DIAL_XTRA, 43

TID_FAX1, 43

TID_FAX2, 44

TID_RNGBK1, 43

TN_GEN, 64

Tone definitions, 16

Tone events, retrieving, 57

Tone Templates, 54, 56

Tones

- maximum number for Global
Detection, 62
- maximum number for R2 MF, 82

Transaction Record, 9, 113

V

virtual boards, 3

voice board, R2 MF, 81

voice coder, G.726, 10, 131

Voice Profile for Internet Messaging
(VPIM), 10, 131

voice.prm, 115

volume. *See* speed and volume control

VPIM, 10, 131

X

xx_listen(), 10, 119

