Compatibility Guide for the Dialogic R4 API on DM3 Products

for Linux and Windows

05-1237-011

COPYRIGHT NOTICE

Publication Date: January 2002

Part Number: 05-1237-011

Intel Corporation
Telecommunications and Embedded Group
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

List of Tables

List of Figures

1. Introduction to this Publication

This document describes the R4 API support for DM3 devices. See *Section 2.1. Introduction to R4 for DM3* for the meaning of the terms “R4” and “R4 for DM3.”

1.1. How This Guide is Organized

This guide is organized as follows:

Chapter 1. Introduction to this Publication describes how this document is organized and lists related Dialogic publications.

Chapter 2. The Dialogic R4 API for DM3 provides a short summary of the technologies and basic features that are supported. It also presents specific details for system-related information, such as software configuration, initialization, and operation.

Chapter 3. R4 SCbus Routing API for DM3 describes the SCbus routing support, along with any compatibility issues, restrictions, and limitations.

Chapter 4. R4 Voice API for DM3 describes in detail the Voice Library support, along with any compatibility issues, restrictions, and limitations.

Chapter 5. R4 GlobalCall API for DM3 describes in detail the GlobalCall Call Control Library support, along with any compatibility issues, restrictions, and limitations.

Chapter 6. R4 DTI API for DM3 describes in detail the DTI Network Interface Library support, along with any compatibility issues, restrictions, and limitations.

Chapter 7. R4 Fax API for DM3 describes the Fax Library support including any compatibility issues, restrictions and limitations.

Chapter 8. R4 DCB API for DM3 describes the DCB Audio Conferencing Library support, including any compatibility issues, restrictions, and limitations.

Chapter 9. R4 MSI API for DM3 describes the MSI API support including any compatibility issues, restrictions, and limitations.

Chapter 10. R4 Audio Input API for DM3 describes the AI API support including any compatibility issues, restrictions, and limitations.

Chapter 11. Implementing Applications for R4 on DM3 provides programming information that may be helpful when creating an application program that uses R4 for DM3.

Chapter 12. R4 on DM3 Resource Routing and Cluster Configuration provides general information on the R4 on DM3 fixed and flexible routing configurations, including an overview of DM3 clusters as well as all information specific to using a fixed routing configuration.

Appendix A - CPA Default Tone Templates provides detailed characteristics on all supported call progress analysis tone templates.

Index contains an alphabetical list of features and topics.

1.2. Related Publications and Information

The following software references correspond to the API features discussed in this publication:

- *Voice Software Reference for Windows:*
  Features Guide (05-1457-001 or later)
  Programmer's Guide (05-1456-003 or later)
  Standard Run-time Library (05-1458-001 or later)
- *Voice Software Reference for Linux*
  Features Guide (05-1454-003 or later)
  Programmer's Guide (05-1453-002 or later)
  Standard Run-time Library (05-1455-003 or later)
- *GlobalCall API Software Reference* (05-0387-009 or later)
- *GlobalCall E-1/T-1 Technology User's Guide* (05-0615-006 or later)
- *GlobalCall ISDN Technology User's Guide* (05-0653-003 or later)
- *GlobalCall IP Technology User's Guide* (05-1512-001 or later)

- *GlobalCall Application Developer's Guide* (05-1526-002 or later)
- *Digital Network Interface Software Reference* (05-1313-004 or later)
- *SCbus Routing Software Reference for Windows* (05-0439-003 or later)
- *SCbus Routing Function Reference for Linux* (05-0313-002 or later)
- *SCbus Routing Guide* (05-0289-005 or later)
- *Fax Software Reference for Windows* (05-0172-010 or later)
- *Fax Software Reference for Linux* (05-0036-010 or later)
- *Dialogic Audio Conferencing Software Reference* (05-0512-002 or later)
- *MSI/SC Software Reference* (05-1218-003 or later)
- *Continuous Speech Processing API Programming Guide* (05-1699-001 or later)
- *Continuous Speech Processing API Library Reference* (05-1700-001 or later)
- *Continuous Speech Processing Demo Guide* (05-1701-001 or later)

The following references may also be of use:

- *DM3 GlobalCall Resource T-1 Protocol User's Guide* (05-1225-001 or later)
- *DM3 GlobalCall Resource ISDN Protocol User's Guide* (05-1226-001 or later)
- *DM3 Mediastream Architecture Overview* (05-0813-002 or later)
- *ISDN Software Reference* (05-0867-002 or later)
- *System Release Installation and Configuration Guide for Windows* (05-1194-005 or later)
- *System Release Software Installation and Configuration Guide for Linux* (05-1420-002 or later)
- *DM3 Configuration File Reference* (05-1272-002 or later)
- *IP Media Users Guide* (05-1594-001-01 or later)

See also important information in the *Release Guide* and *Release Updates* for the software system release in which R4 for DM3 is provided.

2. The Dialogic R4 API for DM3

2.1. Introduction to R4 for DM3

The term “**Dialogic R4 API**” (“Dialogic System Software Release 4 Application Programming Interface”) is used to identify the Dialogic well-established and time-proven direct interface used for creating computer telephony application programs. The R4 API is actually a rich set of proprietary APIs for building computer-telephony applications tailored to Dialogic hardware products. These APIs encompass technologies that include voice, network interface, fax, and speech.

The R4 API now supports a new generation of Dialogic hardware products, which are based on the Dialogic DM3 mediastream product architecture.

When this documentation uses the name “**R4 for DM3**” and “**R4 on DM3**,” it refers to specific aspects of the R4 interface that relate to support for DM3 products.

NOTE: The terms SpringWare boards (or SpringWare devices) and earlier-generation boards (or earlier-generation devices) are used interchangeably in this document. These are boards whose device family is not DM3, such as the Dialog/HD series of boards.

2.2. Feature Overview of R4 for DM3

IMPORTANT

The R4 API for DM3 is provided in a Dialogic software release that may contain additional restrictions than those listed here. A feature listed here as “supported” in R4 for DM3 may be listed in the Dialogic software *Release Guide* or *Release Updates* as “not supported” on DM3 boards. See these documents for details on additional restrictions, compatibility issues, and known problems.

The following is a summary of the features supported in R4 for DM3. The rest of this publication describes the differences between the R4 and the R4 for DM3 implementations.

NOTE

This release provides support for two types of routing configuration: fixed and flexible. Basically, **fixed routing**, which was the only configuration available in the first two System Releases containing R4 on DM3 (DNA 3.3 and SR 5.0), is a restricted routing, whereas **flexible routing** is compatible with the routing capabilities in the R4 API. These routing configurations are fully described in *Section 12.1. Fixed and Flexible Routing Configurations*.

Since the routing configuration affects the routing capability and resource management features that are supported, both routing configurations are documented in this *Compatibility Guide*. The routing information in this *Compatibility Guide* is presented under the assumption that flexible routing will be used. However, because fixed routing is an option, information on fixed routing is provided, although it is gathered entirely into *Chapter 12. R4 on DM3 Resource Routing and Cluster Configuration*.

The availability of flexible routing for a specific Dialogic product depends upon the Dialogic software release in which the product is supported. Fixed and flexible routing are supported on Dialogic products as indicated in the Dialogic software *Release Guide* and *Release Updates*, so refer to those documents for information on specific product support.

2.2.1. Basic System Features Supported

The following is a list of the system features that are supported in **Windows**:

- Full support for the Standard Runtime Library and all standard programming models, which include the following:
    - Synchronous

- Asynchronous: Window Callback, Polled, Callback
- Extended Asynchronous
- Mixed Models that combine the other models

The following is a brief list of the system features that are supported in **Linux**:

- Support for the Standard Runtime Library (SRL) and the following standard programming models:
  - Synchronous
  - Asynchronous: Polled, Non-Signal Callback
  - Extended Asynchronous
  - Mixed Models that combine the other models

NOTE

The Standard Runtime Library
Signal Callback Programming Model is **not** supported.

- Compatibility and Interoperability:
  - API device query capability can return new data to identify and distinguish DM3 boards from other boards
  - A single application program can use DM3 and earlier-generation devices (e.g., Dialog/HD) together in one system subject to the restricted conditions as noted in this publication
  - R4 for DM3 will not affect existing R4 applications

2.2.2. Basic SCbus Routing Features Supported

The following is a list of the SCbus routing software features that are supported. The SCbus protocol is used and the SCbus routing API applies to all the boards regardless of whether they use an SCbus or CT Bus physical interboard connection.

SCbus routing features are documented in the *SCbus Routing Function Reference* and the *SCbus Routing Guide*, which describe the features and associated technical information such as parameters, functions and data structures.

- SCbus voice resource, fax resource, GlobalCall network interface timeslot, and DTI network interface timeslot routing
- SCbus device information for DM3 boards (can distinguish DM3 boards from other boards)

2.2.3. Basic Voice Features Supported

IMPORTANT

The R4 API for DM3 is provided in a Dialogic software release that may contain additional restrictions than those listed here. A feature listed here as "supported" in R4 for DM3 may be listed in the Dialogic software *Release Guide* or *Release Updates* as "not supported" on DM3 boards. See these documents for details on additional restrictions, compatibility issues, and known problems.

The following is a list of the key voice software features that are supported. Voice features are documented in the *Voice Software Reference*, which describes the features and their technical information such as parameters, functions and data structures.

- Continuous Speech Processing (CSP): The CSP features and any related compatibility issues are documented in the *Continuous Speech Processing API Library Reference* and *Continuous Speech Processing API Programming Guide*. These documents discuss the **ec_reciottdata( )**, **ec_stream( )**, and other CSP functions, along with related CSP voice parameters. Although CSP is related to the Voice API, it is provided as a separate product, so CSP voice parameters are only covered in CSP documentation and not in the *Voice Software Reference*.
- Signaling: Tone Detection/Generation:
    - Dual Tone Multi Frequency (DTMF)
    - Multi Frequency (MF) (Tone Generation only)

    - Global Tone Detection (GTD) user-defined tones
    - Global Tone Generation (GTG) user-defined tones, including Cadenced Tone Generation
- Play and Record:
    - OKI ADPCM at 6 kHz with 4-bit samples (24 Kb/s) and 8 kHz with 4-bit samples (32 Kb/s), VOX and WAVE file formats
    - G.711 PCM at 6 kHz with 8-bit samples (48 Kb/s) and 8 kHz with 8-bit samples (64 Kb/s) using A-law or μ-law coding, VOX and WAVE file formats
    - G.721 at 8 kHz with 4-bit samples (32 Kb/s), VOX and WAVE file formats
    - PCM at 11 kHz with 8-bit samples (88 Kb/s) and 11 kHz with 16-bit samples (176 Kb/s) using linear coding, VOX and WAVE file formats. (see the DX_XPB data structure in the *Voice Software Reference*; for 176 Kb/s specify 16 in **wBitsPerSample**.),
    - TrueSpeech at 8 kHz with 16-bit samples (8.5 Kb/s), VOX and WAVE file formats
    - GSM 6.10 full-rate voice coder at 13 Kb/s using Microsoft format (VOX and WAVE file formats) and TIPHON format (VOX file format only), (see the DX_XPB data structure in the *Voice Software Reference* for details)
    - G.726 bit-exact voice coder at 8 kHz with 2-, 3-, 4-, or 5-bit samples (16, 24, 32, 40 Kb/s), VOX and WAVE file formats (see the DX_XPB data structure in the *Voice Software Reference* for details)
    - Offset playback of WAVE files with the **dx_playiottdata( )** function and the IO_USEOFFSET parameter in the DX_IOTT data structure
    - PerfectPitch Speed Control
    - PerfectLevel Volume Control
- Transmit/Receive Analog Display Services Interface (ADSI) data. See *Section 4.7. Support for ADSI 2-Way FSK*.

- Call Analysis through the **dx_dial( )** function. See section *4.6. Voice API Call Progress Analysis Support*.
- Transaction Record
- Bulk data buffer sizing through the **dx_setchxfercount( )** function.
- Silence Compression Record supports the following coders:
  - OKI ADPCM at 6 kHz with 4-bit samples (24 Kb/s) and 8 kHz with 4-bit samples (32 Kb/s), VOX and WAVE file formats
  - G.711 PCM at 6 kHz with 8-bit samples (48 Kb/s) and 8 kHz with 8-bit samples (64 Kb/s) using A-law or µ-law coding, VOX and WAVE file formats
  - G.721 at 8 kHz with 4-bit samples (32 Kb/s), VOX and WAVE file formats
  - PCM at 11 kHz with 8-bit samples (88 Kb/s) and 11 kHz with 16-bit samples (176 Kb/s) using linear coding, VOX and WAVE file formats (see the DX_XPB data structure in the *Voice Software Reference*; for 176 Kb/s specify 16 in **wBitsPerSample**)
  - G.726 bit-exact voice coder at 8 kHz with 2-, 3-, 4-, or 5-bit samples (16, 24, 32, 40 Kb/s), VOX and WAVE file formats (see the DX_XPB data structure in the *Voice Software Reference* for details)
- Other Features:
  - API device query capability can return new data to identify and distinguish DM3 boards from other boards

2.2.4. Basic GlobalCall Call Control Features Supported

IMPORTANT

The R4 API for DM3 is provided in a Dialogic software release that may contain additional restrictions than those listed here. A feature listed here as “supported” in R4 for DM3 may be listed in the Dialogic software *Release Guide* or *Release Updates* as “not supported” on DM3 boards. See these documents for details on additional restrictions, compatibility issues, and known problems.

The following is a list of the key call control software features that are supported:

- Basic call control, including the ability to make a call, answer a call, release a call, etc.
- Partial support for ISDN supplementary services
- Ability to dynamically get and set protocol parameters at run time

GlobalCall features are documented in the *GlobalCall API Software Reference*, which describes the features and their technical information such as parameters, functions and data structures, and the *GlobalCall E-1/T-1 Technology User's Guide* and *GlobalCall ISDN Technology User's Guide*, which describe technology-specific information.

The following protocols are supported:

- T-1 Robbed Bit protocols:
  - E&M
  - Ground Start FXS
  - Loop Start FXS
- E-1 CAS R2MF protocols
- ISDN protocols: Including basic call control and partial supplementary services support (see *Section 5.9.3. gc_SndMsg( ) Restrictions*).
- Analog protocols (on DI products only, where DI refers to **D**ialogic brand **I**ntegrated (DI) Series of multifunctional telephony boards)
- IP protocols (on IPLink products only)

See the *Release Guide* for a list of the currently supported protocols.

2.2.5. Basic DTI Network Interface Features Supported

Most DTI Network Interface software features are **not** supported. GlobalCall call control replaces the basic DTI network interface functionality. The following is a list of the DTI features that can be used in conjunction with GlobalCall features.

- Support for layer 1 alarms, that is, physical alarms
- SCbus DTI network interface timeslot routing

- API device query capability can return new data to identify and distinguish DM3 boards from other boards
- Resource management (open and close devices)

2.2.6. Basic Fax Features Supported

IMPORTANT

The R4 API for DM3 is provided in a Dialogic software release that may contain additional restrictions than those listed here. A feature listed here as “supported” in R4 for DM3 may be listed in the Dialogic software *Release Guide* or *Release Updates* as “not supported” on DM3 boards. See these documents for details on additional restrictions, compatibility issues, and known problems.

The following is a list of key fax software features that are supported. Fax features are documented in the *Fax Software Reference*.

- API device query capability can return new data to identify and distinguish DM3 boards from other boards
- Data rate
  - up to 14.4 kbps transmission
  - up to 14.4 kbps reception
  - selectable preferred data transmission and reception rates
- File storage format
  - raw MH, MR and MMR encoded data
  - TIFF/F MH, MR and MMR encoded data
  - ASCII for transmit only
- Data transmission encoding scheme (over the phone line)
  - MH, MR, and MMR
  - selectable data transmission encoding scheme

- Data reception encoding scheme (over the phone line)
  - MH, MR, and MMR
  - selectable data reception encoding scheme
- T.30 Error Correction Mode (ECM)
- T.30 Phase B event notification
- T.30 Phase D event notification

2.2.7. Basic DCB Audio Conferencing Features Supported

IMPORTANT

The R4 API for DM3 is provided in a Dialogic software release that may contain additional restrictions than those listed here. A feature listed here as “supported” in R4 for DM3 may be listed in the Dialogic software *Release Guide* or *Release Updates* as “not supported” on DM3 boards. See these documents for details on additional restrictions, compatibility issues, and known problems.

The following is a list of key DCB audio conferencing software features that are supported. The DCB features are documented in the *Dialogic Audio Conferencing Software Reference*.

- The maximum number of conferees that can be conferenced is not limited to 32, as with earlier-generation boards. The maximum number of conferees depends upon the hardware (refer to the *Release Guide* for hardware-specific information).
- Coach/pupil feature allowing a private subconference between two conferees.
- Tone clamping to reduce the amount of DTMF tones heard in a conference.
- Automatic gain control for all conferees.
- Active talker indication identifies which conferees are currently talking.
- Echo cancellation is supported by the DCB resource for each talker (DM3 only)

In addition, the DCB API contains several enhancements specifically for DM3 boards (see *8.5. DCB Audio Conferencing API Enhancements for DM3 Boards*).

2.2.8. Basic MSI Features Supported

IMPORTANT

The R4 API for DM3 is provided in a Dialogic software release that may contain additional restrictions than those listed here. A feature listed here as "supported" in R4 for DM3 may be listed in the Dialogic software *Release Guide* or *Release Updates* as "not supported" on DM3 boards. See these documents for details on additional restrictions, compatibility issues, and known problems.

The following is a list of the key MSI software features that are supported. MSI features are documented in the *MSI/SC Software Reference*, which describes the features and their technical information such as parameters, functions and data structures.

- Station Interface support
  - Capability to ring telephones with a Ringer Equivalence Number (REN) of two, per station
  - Capability to ring telephones with FSK Caller ID
  - Programmable ring cadence options
  - Capability to transmit FSK Message Waiting Indicator while on-hook
  - Tone generation and DTMF detection support via dedicated tone-only voice devices
  - Ability to detect on-hook, off-hook and hook-flash
- Station Management support
  - API device query capability can return new data to identify and distinguish DM3 boards from other boards
  - Ability to determine a station's dedicated tone-only voice device via enhancements to **ms_getctinfo( )**

2.2.9. Basic Audio Input Features Supported

IMPORTANT

The R4 API for DM3 is provided in a Dialogic software release that may contain additional restrictions than those listed here. A feature listed here as "supported" in R4 for DM3 may be listed in the Dialogic software *Release Guide* or *Release Updates* as "not supported" on DM3 boards. See these documents for details on additional restrictions, compatibility issues, and known problems.

The Audio Input (AI) API is used to provide music or other information on-hold. A list of the Audio Input API software features supported is provided in Chapter *10. R4 Audio Input API for DM3*.

2.3. Configuration, Initialization, and Operation

This section describes important information related to the configuration, start-up, and operation of R4 for DM3.

2.3.1. Device Initialization Hint

The **xx_open( )** functions for the Voice (dx), GlobalCall (gc), Network (dt), and Fax (fx) APIs are asynchronous in this release of R4 on DM3, unlike the standard R4 versions, which are synchronous. This should usually have no impact on an application, except in cases where a subsequent function calls on a device that is still initializing, that is, is in the process of opening. In such cases, the initialization must be finished before the follow-up function can work. The function won't return an error, but it is blocked until the device is initialized.

For instance, if your application called the following two functions:

```
dx_open()

dx_getfeaturelist()
```

R4 API
for DM3

The **dx_getfeaturelist( )** is blocked until the initialization of the device is completed internally, even though **dx_open( )** has already returned success. In other words, the initialization (**dx_open**) may appear to be complete, but, in truth, it is still going on in parallel.

With some applications, this may cause slow device-initialization performance. Fortunately, you can avoid this particular problem quite simply by reorganizing the way the application opens and then configures devices. The recommendation is to do all **xx_open( )** functions for all channels before proceeding with the next function. For example, you would have one loop through the system devices to do all the **xx_open( )** functions first, and then start a second loop through the devices to configure them, instead of doing one single loop where a **xx_open( )** is immediately followed by other API functions on the same device. With this method, by the time all **xx_open( )** commands are completed, the first channel will be initialized, so you won't experience problems.

This change is not necessary for all applications, but if you experience poor initialization performance, you can gain back speed by using this hint.

2.3.2. Component Interoperation

Media Loads

Different configurations for DM3 products are supported in the form of media loads. For instance, a specific media load is available for users who need to implement CSP and conferencing in their applications. See the *Release Guide* for specific media loads available with this release.

R4 for DM3 Interoperability with R4

- R4 for DM3 can be installed and operated in a computer with existing R4 applications without affecting those applications; however, when the computer contains both DM3 and earlier-generation devices (such as Dialog/HD), the **sr_getboardcnt( )** function will return the count for both types of devices in Windows. In Linux, use SRL device mapper functions to return information about the structure of the system. These functions are described in the *Voice Software Reference - Standard Run-time Library* .
- A single R4 application program can be written or modified to use both DM3 and earlier-generation devices (such as Dialog/HD) together in one system.

- The DM3 clustering method of voice resource and network interface timeslots implemented in the R4 API for DM3 is consistent with the clustering operation used in the DM3 embedded platform interface known as the DM3 Direct Interface (MDI). See Chapter *12. R4 on DM3 Resource Routing and Cluster Configuration* for a description of DM3 clusters.

R4 for DM3 Interoperability with Other APIs/Interfaces

- R4 for DM3 can coexist (can be installed and can reside in a single PC) along with other software APIs or interfaces, such as the DM3 Application Foundation Code (AFC) and the DM3 embedded platform interface known as the DM3 Direct Interface (MDI).
- As with the R4 API, application programs designed with these APIs **cannot** run simultaneously unless each application uses its own set of unique DM3 components. Resource sharing between different applications is not supported due to the lack of a central resource manager to control access to the components.

2.3.3. Board Device Names and Numbers

As usual, board device names are assigned during the Dialogic Service initialization process. The Service uses the standard R4 naming conventions and numbering sequence for DM3 devices that are in the system configuration. (See the *System Release Installation and Configuration Guide* for full details on the R4 device-naming conventions.)

The following conventions apply to DM3 board naming and numbering for R4:

- All DM3 board devices are assigned standard device names for R4; for example, dxxxB1, dxxxB2, dtiB1, dtiB2.
- All DM3 channel and timeslot devices are assigned standard device names for R4; for example, dxxxB1C1, dxxxB1C2, dtiB1T1, dtiB1T2.
- A single physical DM3 board device can contain multiple “virtual boards” that are each numbered in sequential order; for example, a DM/V960-4T1 QuadSpan board with four digital network interfaces contains four virtual network interface boards that would follow a sequential numbering pattern such as dtiB1, dtiB2, dtiB3, dtiB4.

- In Windows, all DM3 board devices are numbered in sequential order **after** the earlier-generation devices (e.g., Dialog/HD boards) are numbered.

  Windows Naming Example:
  DM3: dtiB2 - dtiB5 / dxxxB7 - dxxxB30
  Springware: dtiB1 / dxxxB1 - dxxxB6

- In Linux, if all boards download properly, DM3 boards are named first and earlier-generation boards are named second. This is the reverse of Windows. To handle this naming convention, ensure your configuration files are set up so that your device names point to the proper boards. For example, take a system with a SpringWare board and a DM3 board.

  Linux Naming Example:
  DM3: dtiB1 - dtiB4 / dxxxB1 - dxxxB24
  Springware: dtiB5 / dxxxB25 - dxxxB30

- All DM3 board devices are numbered in sequential order based on the logical Board ID assigned by the DM3 driver (the board having the lowest logical Board ID will be assigned the next board number, and so on).
- In Windows, when R4 for DM3 is installed on a computer containing both DM3 and earlier-generation devices (such as Dialog/HD), the **sr_getboardcnt( )** function when called with DEV_CLASS_VOICE returns a count of all voice board devices including DM3 boards. When called with DEV_CLASS_DTI, it returns a count of all DTI board devices including DM3 boards.
- In Linux, when R4 for DM3 is installed on a computer containing both DM3 and earlier-generation devices (such as Dialog/HD), use SRL device mapper functions to return information about the structure of the system. These functions are described in the *Voice Software Reference - Standard Run-time Library*.
- For an illustration of these device numbering conventions, see *Table 1. Example of Assigned Device Names and Numbers* on page 24.

2.3.4. Compatibility of Device Names/Numbers

Although there is consistency among different types of compatible Dialogic hardware in how devices are numbered, device mapping (device naming or device numbering) is hardware dependent. If a programmer "hard-codes" an application to use device names based on specific Dialogic boards, some of those device names may need to be changed if a different model board is used as a replacement. You can achieve the greatest degree of backward compatibility among Dialogic boards by making the device mapping in the application program hardware independent. The method for achieving this, along with sample application code, is provided in the Technical Note entitled "Identifying the number and type of Dialogic boards in a Windows NT system from within an application," which is found at *http://support.dialogic.com/tnotes/tnbyos/winnt/tn193.htm* . It can also be accessed through the Dialogic technical support web site at *http://support.dialogic.com* by selecting Answers, then Tech Notes, and then the operating system, “Win NT.”

NOTE: DM3 E1 dxxx channels are allocated contiguously. On earlier-generation products , they are allocated with a 2-channel gap. For example, with earlier-generation products, the names dxxxB1C1-dxxxB8C2 would be assigned for the first card and dxxxB9C1-dxxxB16C2 for the second card in a system. Notice that device names dxxxB8C3 and dxxxB8C4 do not exist. By contrast, for a DM3 DualSpan E1, the naming convention is contiguous: dxxxB1C1-dxxxB15C4.

2.3.5. Determining Channel Capabilities in Flexible Routing Configurations

There are three different types of voice devices: the **ISDN compatible** (all), the **T1 CAS compatible**, and the **E1 CAS compatible**. The T1 CAS compatible is a superset of ISDN compatible, and E1 CAS compatible is a superset of T1 CAS compatible.

When using GlobalCall, only certain DM3 network interfaces can be attached to certain other DM3 voice devices using **gc_Attach( )**, **gc_Open( )** or **gc_OpenEx( )**. Attaching DM3 devices together depends on the network protocol used and voice device capabilities. Specifically:

- A DM3 ISDN network device can be attached to any DM3 voice device.
- A DM3 T1 CAS network device must be attached to a T1 CAS compatible DM3 voice device.
- A DM3 E1 CAS network device must be attached to an E1 CAS compatible DM3 voice device.

When using GlobalCall, if a voice device is not CAS or R2MF capable, it cannot be attached (in the attach call or in the open call) to a network interface device that has CAS or R2MF loaded. Likewise, if a voice device is not routable, it cannot be used in a **gc_Attach( )** call.

While a network interface protocol cannot be determined programmatically, the function **dx_getfeaturelist( )** provides a programmatic way of determining voice capability so that the application can make decisions. **dx_getfeaturelist( )** returns bitmask information about the type of front end it can support, meaning ISDN, CAS (T1 CAS), R2MF (E1 CAS), or a combination thereof. Since the information is returned as a bitmask, each is independent of the other, so you can expect to get one and/or the other bitmask.

ft_front_end Feature Defines

The following have been added to the *dxxxlib.h* under the FrontEnd (ft_front_end) feature defines:

```
FT_ROUTEABLE
FT_ISDN
FT_CAS
FT_R2MF
```

The ft_front_end feature defines in the *dxxxlib.h* header may contain the above four possible values. Note that for fixed routing, the FT_ROUTEABLE is not set, so none of the other bits is set. For flexible routing, the FT_ROUTEABLE bit is set, and the other three bits are set based on cluster contents.

For instance, if the ft_front_end bitmask is FT_ROUTEABLE | FT_ISDN | FT_CAS, this means that the channel is capable of flexible routing and can also work with an ISDN or a CAS (T1) frontend. In this example, R2MF is missing, so the channel cannot work with a front-end that is R2MF (E1 CAS) capable. As another example, FT_ROUTEABLE | FT_ISDN | FT_CAS | FT_R2MF indicates support for flexible routing plus all three front-end capabilities, including R2MF.

2.3.6. Getting, Using, and Closing Device Handles in a DM3 Flexible Routing Configuration

In the DM3 flexible routing configuration, R4 on DM3 works the same way as the R4 API. You can use the same GlobalCall device initialization, handling, and routing procedures for DM3 devices as you use for earlier-generation devices.

DM3 devices have similar characteristics to earlier-generation devices. The device must first be opened in order to obtain its handle, which can then be used to access the device functionality. Since an R4 for DM3 application must use GlobalCall for call control, i.e., for call setup and tear-down, all DM3 network interface devices must be opened with the **gc_Open( )** or **gc_OpenEx( )** function. However, if you need to open a network device and you aren't using call control (e.g., E1 clear channel configuration) or if you want to open a network board device for alarm handling, you can use **dt_open( )**.

Once the call has been established, voice and data streaming should be done using the Dialogic voice API. Functions such as **dx_playiottdata( )**, **dx_reciottdata( )**, and **dx_dial( )** can be used. Of course, in order to do so, the voice device handle must be obtained. By the same token, in order to access the DTI routing functionality with functions such as **dt_listen( )** and **dt_unlisten( )**, the corresponding DTI device handle must be obtained. The same thing applies to fax operations. A fax device must be explicitly opened using **fx_open( )** so as to retrieve the fax device handle.

When opening a DM3 network interface device using **gc_Open( )** or **gc_OpenEx( )**, the user has the option of also specifying a voice device in the **devicename** parameter. If a voice device is specified, then the network interface device is automatically associated with the voice device (it is attached and routed on the CT Bus). If a voice device is not specified, the voice device can be opened separately and then manually attached and routed to the network interface device.

NOTE: DM3 voice devices can only be attached or specified in **gc_OpenEx( )** or **gc_Open( )** with DM3 network devices. You cannot attach or specify a DM3-SpringWare combination.

The following information applies to the DM3 flexible routing configuration and presents some additional guidelines for DM3 device-handling using the GlobalCall API.

- Use **gc_Open( )** to open a network interface device. When opening a network interface device, a voice device may also be specified in the **devicename** parameter.

  - If a voice device is specified, it is automatically associated with the network interface device (it is attached and routed on the CT Bus). The **gc_GetVoiceH( )** function can be used to retrieve the voice device handle and the **gc_GetNetworkH( )** function can be used to retrieve the network interface device handle. In this case, the **devicename** string for the **gc_Open( )** function is similar to the following:

    ```
    ":N_dtiB1T1:V_dxxxB1C1:P_ISDN"
    ```

  - If a voice device is not specified, the voice device must be opened separately using **dx_open( )**, and the device must be manually associated with the network interface device using the **gc_Attach( )** function. In this case, the **devicename** string for the **gc_Open( )** function is similar to the following:

    ```
    ":N_dtiB1T1:P_ISDN"
    ```

  See *Section 5.3. gc_Open( ) and gc_OpenEx( ) Restrictions* for more information.

- In general, when using DM3 **and** earlier-generation boards, the R4 for DM3 application must use a **device-discovery** procedure to become "hardware-aware." To perform device discovery and identify whether a logical device belongs to a DM3 board, use the SCbus routing device information functions, such as **dx_getctinfo( )** and **dt_getctinfo( )**, and check the CT_DEVINFO **ct_devfamily** field for the CT_DFDM3 value. (For details see *Section 3.3. CT_DEVINFO: Channel/Timeslot Device Information Data Structure* and *Section 11.3.2. Device Discovery for DM3 Boards and Earlier-Generation* Boards.)

- For more detailed information on how to initialize GlobalCall for use with DM3 boards only, see *Section 11.3.1. Initializing the GlobalCall API for DM3 Boards Only (Flexible Routing)*.

- To close the devices and release the device handles:

  - Use **fx_close( )** to close any fax devices opened with **fx_open( )**.

- Use **gc_Detach( )** to detach any voice and network interface devices associated with **gc_Attach( )**.
- Use **dx_close( )** to close any voice devices opened with **dx_open( )**.
- Use **dt_close( )** to close any network interface devices opened with **dt_open( )**.
- Use **gc_Close( )** to close any voice and/or network interface devices opened with **gc_Open( )**.

- Application performance may be a consideration when opening and closing devices with GlobalCall. If the application uses GlobalCall to dynamically open and close devices as needed, it can impact the application's performance. One way to avoid this is to open all DM3 devices during application initialization and keep them open for the duration of the application, closing them only at the end.
- Since the fax library does not support the use of a voice handle for fax commands, you cannot use the device handle from **dx_open( )** to call fax API functions. You must use **fx_open( )** to open a channel device for fax processing and use that fax device handle. Therefore, it is recommended that when using flexible routing on DM3 boards you do the following to retrieve and use the fax device handle.
    - Use **dx_open( )** to open the voice channel device and retrieve the voice device handle.
    - **Optional:** To determine whether the channel device supports fax before you attempt to open the fax device, use **dx_getfeaturelist( )** and specify the voice device handle. If the ft_fax field of the returned FEATURE_TABLE structure contains the bitmask "FT_SENDFAX_TXFILE_ASCII | FT_FAX | FT_VFX40 | FT_VFX40E | FT_VFX40E_PLUS", then this device supports fax.

      If this is a fax only channel, close this voice device handle, as it will no longer be used. For more information on **dx_getfeaturelist( )**, see section *2.3.5. Determining Channel Capabilities in Flexible Routing Configurations*.
    - Use **fx_open( )** on the voice channel device to open the associated fax device and retrieve the fax handle. The function will succeed if the channel device has fax capabilities; otherwise the function will fail.

Table 1. Example of Assigned Device Names and Numbers in Windows

Hardware	Resource Type	Device Type	Logical Device Names/Numbers
D/480SC-2T1 (BLT board ID 5)*	Voice	Board	dxxxB1 to dxxxB12
		Channel	dxxxB1C1 to dxxxB1C4
		- - - - - - -	to
		Channel	dxxxB12C1 to dxxxB12C4
	DNI**	Board	dtiB1 to dtiB2
		Timeslot	dtiB1T1 to dtiB1T24***
		Timeslot	dtiB2T1 to dtiB2T24***
DM/V960-4T1 (logical board ID 1)*	Voice	Board	dxxxB13 to dxxxB36
		Channel	dxxxB13C1 to dxxxB13C4
		- - - - - - -	to
		Channel	dxxxB36C1 to dxxxB36C4
	DNI**	Board	dtiB3 to dtiB6
		Timeslot	dtiB3T1 to dtiB3T24***
		Timeslot	dtiB4T1 to dtiB4T24***
		Timeslot	dtiB5T1 to dtiB5T24***
		Timeslot	dtiB6T1 to dtiB6T24***

*Note that all earlier-generation devices (e.g., DualSpan D/480SC-2T1) are assigned device numbers (e.g., dxxxB1) before all DM3 devices (e.g., QuadSpan DM/V960-4T1) in Windows.

**Digital Network Interface

***T23 if using ISDN

3. R4 SCbus Routing API for DM3

The following SCbus information applies to a DM3 flexible routing configuration. For information on fixed routing, see *Chapter 12. R4 on DM3 Resource Routing and Cluster Configuration*.

NOTE: When this document refers to the SCbus, SCbus resource or SCbus routing, the information also applies to the CT Bus on DM3 products. That is, the physical interboard connection can be either SCbus or CT Bus. The SCbus protocol is used and the SCbus routing API applies to all the boards regardless of whether they use an SCbus or CT Bus physical interboard connection.

3.1. Basic SCbus Routing Features Not Supported and Partially Supported

Table 2 shows those SCbus routing functions that are not supported on DM3 devices as well as those functions that are partially supported. For those functions partially supported, the nature of the limitation is noted.

The following is a list of the SCbus routing API features that are **not** supported:

- SCbus analog network interface routing not supported
- Echo Cancellation Resource (ECR) routing not supported

3.2. SCbus Routing API Function Restrictions

See the *SCbus Routing Software Reference* for detailed information about each function.

The SCbus routing functions exist in several different libraries. For example:

- The voice library contains SCbus routing functions that begin with "**ag_**" and "**dx_**".
- The DTI library contains SCbus routing functions that begin with "**dt_**".
- The fax library contains SCbus routing functions that begin with "**fx_**".
- The GlobalCall library contains SCbus routing functions that begin with "**gc_**".
- The MSI library contains SCbus routing functions that begin with "**ms_**".

Table 2. List of SCbus Routing API Functions Restricted and Not Supported

Function Name	Notes
ag_getctinfo( )	Not supported.
ag_getxmitslot( )	Not supported.
ag_listen( )	Not supported.
ag_unlisten( )	Not supported.
dx_getxmitslotecr( )	Not supported. (This function is in the *Voice Software Reference*.)
dx_listenecr( )	Not supported. (This function is in the *Voice Software Reference*.)
dx_listenecrex( )	Not supported. (This function is in the *Voice Software Reference*.)
dx_unlistenecr( )	Not supported. (This function is in the *Voice Software Reference*.)
nr_scroute( )	**Limitations:** Does not support analog network interface devices (LSI). Supports DTI, voice, MSI, and fax devices only.
nr_scunroute( )	**Limitations:** Does not support analog network interface devices (LSI). Supports DTI, voice, MSI, and fax devices only.

3.3. CT_DEVINFO: Channel/Timeslot Device Information Data Structure

Use the SCbus routing device information functions, such as **dx_getctinfo( )** and **dt_getctinfo( )**, to obtain information about DM3 devices, which is returned in a CT_DEVINFO data structure. This information can be used to identify whether a logical device belongs to DM3 hardware. The following selected fields in the data structure return information related to DM3 devices:

Table 3. DM3 Device Information in Selected CT_DEVINFO Fields

Field	DM3 Data Values/Description
ct_prodid	Not applicable for a DM3 device (reserved for future use).
ct_devfamily	CT_DFDM3 for a DM3 device.
ct_devmode	CT_DMNETWORK for a DM3 network device or for a DM3 voice device in fixed routing.
	CT_DMRESOURCE for a DM3 voice device in flexible routing.
ct_busencoding	CT_BEULAW or CT_BEALAW for µ-law or A-law coding, respectively.
ct_rfu	Returned by **ms_getctinfo( )** (for DM3 MSI devices).
	Returns a character string containing the board and channel of the voice channel resource associated with the station interface. This data is returned in BxxCy format, where *xx* is the voice board and *y* is the voice channel. For example, dxxxB1C1 would be returned as B1C1. To subsequently use this information in a **dx_open( )** function, you must add the *dxxx* prefix to the returned character string.

3.4. CT Bus Timeslot Application Considerations

When multiple devices are listening to the same CT Bus timeslot, take into account the application considerations discussed in this section. Specifically, these considerations should be taken into account whenever a resource device (such as

SCbus Routing API for DM3

Voice, CSP, or other) listens to the same CT Bus timeslot device as a local, on-board Network Interface device (dtiBnTm).

NOTE: There is NO restriction when listening to a CT Bus timeslot device to which the Network interface device is currently transmitting.

The following figure represents the cases discussed below.

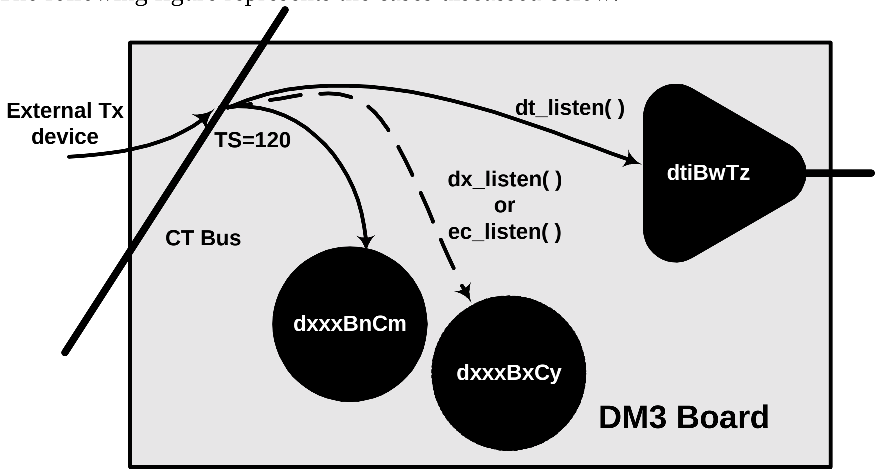


Figure 1. Multiple Devices Listening to the Same CT Bus Timeslot

In a configuration where a DM3 network interface device shares (listens) the same CT Bus timeslot device with a local, on board voice (CSP, Fax, etc.) device, these considerations apply:

1. If an application called **dt_listen( )** or **gc_Listen( )** on a DM3 network interface device (dtiBwTz) to listen to an external CT Bus timeslot device (TS=120 in the figure), followed by one or more **dx_listen( ), ec_listen( )**, **fx_listen( )**, or other related functions to a local, on board voice device (dxxxBnCm) to listen to the same external CT Bus timeslot device, then the application MUST break (unlisten) the CT Bus voice connection(s) first, using **dx_unlisten( )**, **ec_unlisten( )**, or **fx_unlisten( )**, etc.), prior to calling **dt_unlisten( )** or **gc_Unlisten( )** to the local network interface device referred above. Failure to do so causes the latter call or subsequent voice calls to fail. This scenario could arise during recording (or transaction recording) of an external source, during a two party hair-pinning connection.

2. In the same scenario, if an application called **dx_listen( )**, **ec_listen( )**, or **fx_listen( )**, etc. first, then there is no particular sequencing order that must be followed during CT Bus timeslot device tear-down. However, full density (up to 120 channels) of CSP or Transaction Record might not be possible.

3. If multiple local, on-board network interface devices are all listening to the same external CT Bus timeslot device (in this case tslot = 120), the network interface devices must undo the CT Bus connections—that is, call **dt_unlisten( )** or **gc_Unlisten( )—**so that the first network interface to listen to the CT Bus timeslot device is the last one to unlisten. This scenario could arise during broadcasting of an external source to several local network interface channels.

4. R4 Voice API for DM3

4.1. Voice API Features Not Supported (By Category)

The following is a summary of the voice features that are currently **not supported**. The categories are presented for convenience and are not a definitive way of organizing the information. Some categories appear on their own although they may strictly be defined as a subset of another category. Make sure to check all the categories.

For details on the functions and parameters affected, and for a few unsupported functions that do not fall into any of these categories, see *Section 4.3. Voice API Function and Parameter Restrictions*.

For information on all SCbus restrictions (including Echo Cancellation Resource), see *Chapter 3. R4 SCbus Routing API for DM3*.

For a list of supported system and voice features, see *Sections 2.2.1. Basic System Features Supported* and *2.2.3. Basic Voice Features Supported*.

4.1.1. Tone Detection/Generation Features Not Supported

- Voice library **r2_creatfsig( )** function for detecting R2MF signals
- Voice library **r2_playbsig( )** function for detecting R2MF signals
- Multi-frequency (MF) tone detection
- Voice library **dx_setdigtyp( )** and **dx_getdig( )** functions for detecting MF digits

4.1.2. Pulse Detection/Generation Features Not Supported

- Global Dial Pulse Detection (GDPD)
- Pulse dialing

4.1.3. Analog Line Handling Features Not Supported

- Ring detection
- Detection of loop current on/off events
- Detection of rings received
- Hookswitch control
- Detection and generation of a wink

4.1.4. Call Management Features Not Supported

- Analog Caller ID

4.1.5. Play/Record Limitations and Features Not Supported

- **Unsupported** formats include the following (see *Section 2.2.3. Basic Voice Features Supported* for a list of supported formats):
  - VOX and WAVE Linear PCM 6 kHz with 8-bit samples (48 Kb/s)
  - VOX and WAVE Linear PCM 8 kHz with 8-bit samples (64 Kb/s)
  - VOX and WAVE either μ-law or A-law coding PCM at 11 kHz with 8-bit samples (88 Kb/s)
- **Play Does Not Support:**
  - Voice Barge-In (Note: CSP Barge-in is supported.)
- **Play/Record Routing Limitation:**
  - On DM3 boards, voice channels must be listening to a CT Bus timeslot in order for any voice streaming functions, such as **dx_rec( )**, to work. In other words, you must issue a **dx_listen( )** function call on the device handle before calling any voice streaming function for that device handle. Furthermore, the **dx_listen( )** function must be called within the same process as the voice streaming functions. The actual recording operation will start only after the voice channel is listening to the proper external timeslot.

4.1.6. Running Transaction Record

The Transaction Record feature allows you to record two CT Bus time slots from a single channel. Voice activity on two channels can be summed and stored in a single file, device, and/or memory.

Transaction Record uses the external reference as its secondary input to the summing algorithm in the digital signal processors (DSP). Usually, transaction record is used to record both sides of a two-sided conversation.

The following figure shows a typical CT Bus connection where an external reference signal is used.

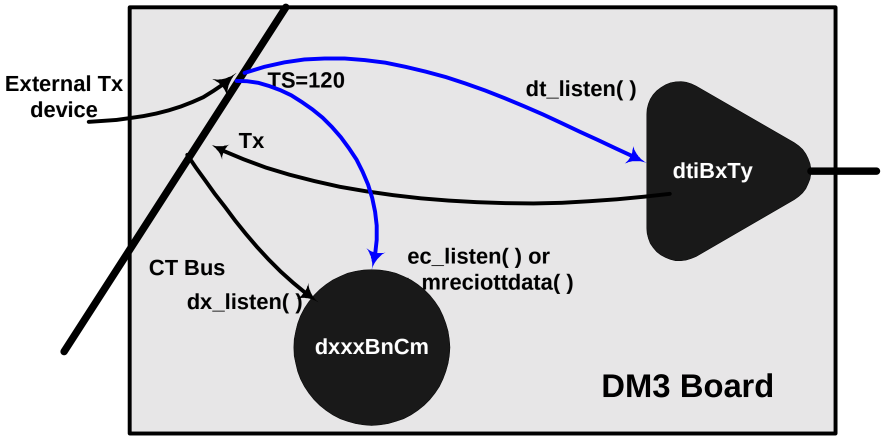


Figure 2. CT Bus Connection with External Reference

Use the guidelines in this section to achieve full density of transaction record, which is up to 120 channels of **dx_mreciottdata( )** on a single DM3 board.

NOTES:
1. These guidelines assume that this feature is used on its own without conferencing, Continuous Speech Processing (CSP), or another feature in use at the same time.
2. These are general guidelines. Not every task is described here; for example, setting up recording parameters is not included.

To run 120 channels of transaction record, follow these guidelines:

1. The voice channel used for transaction record must be on the same physical board as the network interface device that is listening to the far-end input, normally the secondary input on the transaction record (see below). In these guidelines, the voice channel used for transaction record is called the local voice channel, and the network interface device, the local network interface device.

2. On the local network device, issue **dt_listen( )** or **gc_Listen( )** to listen to the CT Bus time slot that is providing one of the inputs (normally the secondary, or external reference) on the transaction record call.

   The device that is providing the secondary input can be derived from another CT Bus time slot. This device does not have to be on the same physical board as the local network interface and local voice channel.

   NOTE: You must call **dt_listen( )** or **gc_Listen( )** to listen to the external reference before **dx_mreciottdata( )** is initiated to achieve full density (up to 120 channels) of transaction record.

3. Issue **dx_mreciottdata( )** on the local voice device where one of the inputs (normally the secondary) is the time slot previously used in the local network interface **dt_listen( )**. This is the same signal from the same time slot discussed in step 2.

4. Before tearing down the connection, wait until **dx_mreciottdata( )** on the local voice channel has completed.

5. To tear down the connection, issue **dt_unlisten( )** or **gc_unlisten( )** on the local network interface device.

   NOTE: Make sure **dx_mreciottdata( )** has completed before attempting to call **dt_unlisten( )** or **gc_Unlisten( )** to break the connection to the external reference, regardless of the number of channels engaged in transaction record or use of the external reference.

6. If receiving DTMFs is one of the termination conditions for recording, make sure that terminating DTMFs are sent over the first of the two timeslots with which the **dx_mreciottdata( )** function is dealing. If the termination DTMFs are sent over the second timeslot, the record will continue.

4.1.7. Miscellaneous Features Not Supported

- Sharing DSP resource features (including DSP Fax)
- Voice library **dx_GetRscStatus( )** resource sharing function
- Syntellect License Automated Attendant functions
- Echo Cancellation Resource

4.2. Voice API Features Partially Supported

The following is a summary of the voice features that are partially supported. See also *Section 4.1. Voice API Features Not Supported (By Category)* for a list of high-level categories containing features that are not supported.

- PerfectLevel Playback Volume Control
- PerfectPitch Playback Speed Control
- I/O terminating conditions in the DV_TPT Termination Parameter Table data structure
- Voice device (board and channel) parameters
- GTG Cadenced Tone Generation (for example: busy & ringback tones)
- Call Progress Analysis (through the **dx_dial( )** function). For more information, see section *4.6. Voice API Call Progress Analysis Support*.

For details on the functions and parameters affected, see *Section 4.3. Voice API Function and Parameter Restrictions*.

For information on all SCbus restrictions, see *Chapter 3. R4 SCbus Routing API for DM3*.

4.3. Voice API Function and Parameter Restrictions

R4 for DM3 supports all voice API functions except those noted in *Table 4. List of Voice API Functions Restricted and Not Supported*. See the *Voice Software Reference* for detailed information about each function.

Restrictions are described in the following areas:

- **Restricted Voice Functions:** Functions not supported and only partially supported (includes parameters not supported).
- **Speed and Volume Control:** Functions and parameters not supported and only partially supported.
- **Termination Conditions:** Parameters not supported and parameters fully supported.
- **Board and Channel Device Parameters:** Parameters not supported and parameters fully supported.

Errors: A voice function that is not supported or is only partially supported (that is, does not support all its parameters) by DM3 boards will return the following error codes:

- If you execute a standard voice function that is not supported by DM3 boards, it produces an EDX_NOTIMP ("not implemented") error.
- If you execute a supported voice function with a parameter that is not supported on DM3 boards, it produces an EDX_NOSUPPORT error.

NOTE: Some voice API functions may produce different voice API errors in R4 for DM3 than in R4. Use the standard error functions **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** to handle the errors.

4.3.1. Voice API Function Restrictions

The following list shows functions that are **not supported** on DM3 devices as well as those functions that are partially supported. For those functions partially supported, the nature of the limitation is noted, or else a reference is given to the location of additional details.

Table 4. List of Voice API Functions Restricted and Not Supported

Function Name	Notes
ag_getctinfo( )	Not supported. (This function is in the *SCbus Routing Software Reference*.)

Function Name	Notes
ag_getxmitslot( )	Not supported. (This function is in the *SCbus Routing Software Reference*.)
ag_listen( )	Not supported. (This function is in the *SCbus Routing Software Reference*.)
ag_unlisten( )	Not supported. (This function is in the *SCbus Routing Software Reference*.)
ATDX_ANSRSIZ( )	Not supported.
ATDX_BUFDIGS( )	Not supported
ATDX_CRTNID( )	Not supported.
ATDX_DTNFAIL( )	Not supported.
ATDX_EVTCNT( )	Not supported.
ATDX_FRQDUR( )	Not supported.
ATDX_FRQDUR2( )	Not supported.
ATDX_FRQDUR3( )	Not supported.
ATDX_FRQHZ( )	Not supported.
ATDX_FRQHZ2( )	Not supported.
ATDX_FRQHZ3( )	Not supported.
ATDX_FRQOUT( )	Not supported.
ATDX_FWVER( )	Not supported.
ATDX_HOOKST( )	Not supported.
ATDX_LINEST( )	Not supported.
ATDX_LONGLOW( )	Not supported.
ATDX_PHYADDR( )	Not supported.
ATDX_RINGCNT( )	Not supported.
ATDX_SHORTLOW( )	Not supported.
ATDX_SIZEHI( )	Not supported.
dx_addspddig( )	**Limitations:** See *Section 4.3.2. Speed and Volume Control Restrictions*.

Function Name	Notes
dx_addtone( )	**Limitations:** The arguments **digit** and **digtype** are not supported for R4 on DM3. Therefore, mapping custom-defined GTD tones to digits (DG_USER1 to DG_USERn) is not supported.
dx_addvoldig( )	**Limitations:** See *Section 4.3.2. Speed and Volume Control Restrictions*.
dx_chgdur( )	Not supported.
dx_chgfreq( )	Not supported.
dx_chgrepcnt( )	Not supported.
dx_dial( )	**Limitations:** See *Section 4.6. Voice API Call Progress Analysis Support*.
dx_dialtpt( )	Not supported.
dx_getdig( )	**Limitations:** Global DPD is not supported (DG_DPD_ASCII is not available).
dx_getdigEx( )	Not supported.
dx_getfeaturelist( )	See *Section 2.3.5. Determining Channel Capabilities in Flexible Routing Configurations* for information on using this function to identify device capabilities.
dx_getparm( )	**Limitations:** See *Section 4.3.4. Device Parameter Restrictions*.
dx_GetRscStatus( )	Not supported.
dx_getxmitslotecr( )	Not supported.
dx_gtcallid( )	Not supported.
dx_gtextcallid( )	Not supported.
dx_initcallp( )	Not supported.
dx_InitializeBargeIn( )	Not supported.
dx_listenecr( )	Not supported.
dx_listenecrex( )	Not supported.

Function Name	Notes
dx_mreciottdata( )	**Limitations:** See *Sections 4.1.5. Play/Record Limitations and Features Not Supported* and *4.1.6. Running Transaction Record*. This function is not supported on DI nor HDSI products, because DI and HDSI products do not support routable voice resources.
dx_play( )	**Limitations:** See *Section 4.1.5. Play/Record Limitations and Features Not Supported* and *Section 4.3.3. DV_TPT Termination Parameter Table Restrictions*. The following modes are not supported: • PM_ADSIALERT • PM_ADSI
dx_playf( )	**Limitations:** See *Section 4.1.5. Play/Record Limitations and Features Not Supported*.
dx_playiottdata( )	**Limitations:** See *Section 4.1.5. Play/Record Limitations and Features Not Supported*.
dx_playtoneEx( )	**Limitations:** The Cadenced Tone Generation TN_GENCAD **cycles** parm is limited to 40 cycles instead of the documented 255 maximum; any number of cycles greater than 40 is limited to 40.
dx_playvox( )	**Limitations:** See *Section 4.1.5. Play/Record Limitations and Features Not Supported*.
dx_playwav( )	**Limitations:** See *Section 4.1.5. Play/Record Limitations and Features Not Supported*.
dx_rec( )	**Limitations:** See *Section 4.1.5. Play/Record Limitations and Features Not Supported*.
dx_recf( )	**Limitations:** See *Section 4.1.5. Play/Record Limitations and Features Not Supported*.
dx_recm( )	Not supported. Use **dx_mreciottdata( )**.
dx_recmf( )	Not supported. Use **dx_mreciottdata( )**.

Function Name	Notes
dx_reciottdata( )	**Limitations:** See *Section 4.1.5. Play/Record Limitations and Features Not Supported*. This function is supported on HDSI products, provided that the correct play/record PCD file is downloaded.
dx_recvox( )	**Limitations:** See *Section 4.1.5. Play/Record Limitations and Features Not Supported*.
dx_recwav( )	**Limitations:** See *Section4.1.5. Play/Record Limitations and Features Not Supported*.
dx_sendevt( )	Not supported.
dx_setdigbuf( )	Not supported.
dx_setdigtyp( )	Not supported.
dx_setevtmsk( )	**Limitations:** Ring detection and Loop Current on/off detection are not supported (DM_RINGS, DM_RNGOFF, DM_LCOFF, DM_LCON, DM_LCREV produces an EDX_NOSUPPORT error).
dx_sethook( )	Not supported.
dx_setparm( )	**Limitations:** See *Section 4.3.4. Device Parameter Restrictions*.
dx_setsvcond( )	**Limitations:** See *Section 4.3.2. Speed and Volume Control Restrictions*.
dx_settone( )	Not supported.
dx_settonelen( )	Not supported.
dx_TSFStatus( )	Not supported.
dx_unlistenecr( )	Not supported.
dx_wink( )	Not supported.
dx_wtcallid( )	Not supported.
dx_wtring( )	Not supported.
li_attendant( )	Not supported.
li_islicensed_syntellect( )	Not supported.
r2_creatfsig( )	Not supported.

Function Name	Notes
r2_playbsig( )	Not supported.

NOTE: **dx_** functions are **not** supported on the IP media devices of IPLink boards. See the *IP Media Users Guide* for information on media control of the IP component.

4.3.2. Speed and Volume Control Restrictions

The following Speed and Volume Control functions are fully supported:

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

The following Speed and Volume Control functions are partially supported as described in the restrictions that follow:

- **dx_addspddig( )**
- **dx_addvoldig( )**
- **dx_setsvcond( )**

The following restrictions apply to Speed and Volume Control:

- Speed Control is partially supported and has the following limitation: Speed Control is supported only at the 8 kHz sampling rate using the PCM voice coder with A-law or µ-law coding, or the OKI ADPCM voice coder.
- Speed Control and Volume Control adjustments are supported by the **dx_setsvcond( )** function with the following limitation:
  - For **action** SV_TOGGLE, an **adjsize** of SV_CURLASTMOD or SV_RESETORIG is **not** supported. The **adjsize** values of SV_CURORIGIN and SV_TOGORIGIN are supported. SV_CURORIGIN sets the current speed or volume level to the origin, and SV_TOGORIGIN toggles between the origin and the last modified speed or volume level.

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

- Speed Control and Volume Control adjustments are supported by the **dx_addspddig( )** and **dx_addvoldig( )** functions with the following limitations:
  - For **dx_addspddig( ),** the **adjval** values of SV_ADD10PCT, SV_SUB10PCT, and SV_NORMAL are supported. These values increase play speed by 10%, decrease play speed by 10%, and set play speed to normal.
  - For **dx_addvoldig( ),** the **adjval** values of SV_ADD2DB, SV_SUB2DB, and SV_NORMAL are supported. These values increase play volume by 2 dB, decrease play volume by 2 dB, and set play volume to normal.

4.3.3. DV_TPT Termination Parameter Table Restrictions

Those voice termination conditions that are supported by the DV_TPT Termination Parameter Table are described below, along with those conditions that are not supported.

Table 5. Termination Conditions Not Supported shows the voice termination conditions that are **not** supported by the DV_TPT Termination Parameter Table (**tp_termno** field).

Table 5. Termination Conditions Not Supported

Termination Condition	Description
DX_LCOFF	• Loop current drop
DX_PMOFF	• Pattern of non-silence
DX_PMON	• Pattern of silence

Table 6. Termination Conditions Supported shows a summary of the voice termination conditions that are supported by the DV_TPT Termination Parameter Table (**tp_termno** field).

Table 6. Termination Conditions Supported

Termination Condition	Default Sensitivity	Description
DX_DIGTYPE	Level	• Digit for standard DTMF
DX_MAXNOSIL	Edge	• Maximum length of non-silence (see restrictions below)
DX_MAXSIL	Edge	• Maximum length of silence (see restrictions below)
DX_MAXDTMF	Level	• Maximum number of digits received
DX_IDDTIME	Edge	• Maximum delay between digits
DX_MAXTIME	Edge	• Maximum function time
DX_DIGMASK	Level	• Specific digit received
DX_TONE	Level	• Tone On/Off for user-defined GTD tone

The supported termination conditions have the following qualifications:

- DX_DIGTYPE is supported with the limitation that the ASCII value set in the **tp_value** field must match a real DTMF tone (0-9, a-d, *, #).
- For DX_MAXNOSIL and DX_MAXSIL termination conditions, the maximum length of time specified is restricted as follows:
  - The range of time that may be used is limited to a minimum of 1 second and a maximum of 30 seconds.
  - The settings within this time range are limited to the following step values: 1, 2, 3, 5, 8, 10, 15, and 30 seconds.
  - If you specify a time less than 1 second, the function will fail with a EDX_BADTPT error.
  - If you specify a time greater than 30 seconds, it will function as if you specified 30 seconds.
  - If you specify any time between 1 and 30 seconds, it will function as if you specified the next lowest step value in the range (it is rounded down). For example, if you set DX_MAXSIL to terminate on 4.9

seconds of silence, it will terminate on three seconds of silence rather than five seconds, because three seconds is the next lowest step value.

- The default sensitivity shown in *Table 6* is given for DM3 boards.
- Parameters for **tp_flags**:
  - For DX_IDDTIME, **tp_flags** TF_FIRST can be used for initial interdigit delay termination.
  - TF_10MS can be used for 10 millisecond timing.
  - The TF_LEVEL, TF_EDGE, and TF_CLREND **tp_flags** have no effect and will be ignored.
  - The TF_SETINIT flag is not supported.

4.3.4. Device Parameter Restrictions

Those board and channel device parameters that are supported by the **dx_getparm( )** and **dx_setparm( )** functions are described here, along with those parameters that are not supported.

Table 7. Voice Board and Channel Parameters Supported shows the board and channel device parameters that are supported.

Table 7. Voice Board and Channel Parameters Supported

Define	Read/ Write	Description
Board Parameters:		
DXBD_CHNUM	R	Channel Number - Returns the number of voice channels on the board.
DXBD_HWTYPE	R	Hardware Type - For DM3 boards, always returns TYP_D41.
DXBD_SYSCFG	R	System Configuration - For DM3 boards, always returns 1 to indicate a digital network interface.

Define	Read/ Write	Description
Channel Parameters:		
DXCH_PLAYDRATE	R/W	Play Digitization Rate. Valid parameter values are: 6000 - 6 kHz sampling rate 8000 - 8 kHz sampling rate
DXCH_RECRDRATE	R/W	Record Digitization Rate. Valid values are: 6000 - 6 kHz sampling rate 8000 - 8 kHz sampling rate
DXCH_SCRFEATURE	R/W	Silence Compression Record (SCR) DXCH_SCRDISABLED DXCH_SCRENABLED
DXCH_XFERBUFSIZE	R	Transfer Buffer Size - Returns the bulk queue buffer size as set by the **dx_setchxfercnt( )** function.

Table 8. Voice Board and Channel Parameters Not Supported shows the board and channel device parameters that are **not** supported.

Table 8. Voice Board and Channel Parameters Not Supported

Define	Read/ Write	Description
Board Parameters:		
DXBD_FLASHCHR	R/W	Flash character
DXBD_FLASHTM	R/W	Flash Time
DXBD_MAXPDOFF	R/W	Maximum Pulse Digit Off
DXBD_MAXSLOFF	R/W	Maximum Silence Off
DXBD_MFDELAY	R/W	MF Interdigit Delay
DXBD_MFLKPTONE	R/W	MF Length of LKP Tone
DXBD_MFMINON	R/W	Minimum MF On
DXBD_MFTONE	R/W	MF Minimum Tone Duration

Define	Read/ Write	Description
DXBD_MINIPD	R/W	Minimum Loop Interpulse Detection
DXBD_MINISL	R/W	Minimum Interdigit Silence
DXBD_MINLCOFF	R/W	Minimum Loop Current Off
DXBD_MINOFFHKTM	R/W	Minimum offhook time
DXBD_MINPDOFF	R/W	Minimum Pulse Detection Off
DXBD_MINPDON	R/W	Minimum Pulse Detection On
DXBD_MINSLOFF	R/W	Minimum Silence Off
DXBD_MINSLON	R/W	Minimum Silence On
DXBD_MINTIOFF	R/W	Minimum DTI Off
DXBD_MINTION	R/W	Minimum DTI On
DXBD_OFFHDLY	R/W	Offhook Delay
DXBD_P_BK	R/W	Pulse Dial Break
DXBD_P_IDD	R/W	Pulse Interdigit Delay
DXBD_P_MK	R/W	Pulse Dial Make
DXBD_PAUSETM	R/W	Pause Time
DXBD_R_EDGE	R/W	Ring Edge
DXBD_R_IRD	R/W	Inter-ring Delay
DXBD_R_OFF	R/W	Ring-off Interval
DXBD_R_ON	R/W	Ring-on Interval
DXBD_S_BNC	R/W	Silence and Non-silence Debounce
DXBD_T_IDD	R/W	DTMF Interdigit delay
DXBD_TTDATA	R/W	DTMF length (duration) for dialing.

Define	Read/ Write	Description
Channel Parameters:		
DXCH_AUDIOLINEIN	R/W	Audio Line In
DXCH_CALLID	R/W	Caller ID

Define	Read/ Write	Description
DXCH_DFLAGS	R/W	DTMF detection edge select
DXCH_DTINITSET	R/W	DTMF Initiation
DXCH_DTMFDEB	R/W	DTMF debounce time
DXCH_DTMFTLK	R/W	Minimum DTMF Time
DXCH_MAXRWINK	R/W	Maximum Loop Current for Wink
DXCH_MFMODE	R/W	Minimum MF KP Time
DXCH_MINRWINK	R/W	Minimum Loop Current for Wink
DXCH_RINGCNT	R/W	Ring Number
DXCH_RXDATABUFSIZE	R/W	Library Buffer Size for Receiving Data
DXCH_T_IDD	R/W	DTMF Interdigit delay
DXCH_TTDATA	R/W	DTMF length (duration) for dialing.
DXCH_WINKDLY	R/W	Wink Delay
DXCH_WINKLEN	R/W	Wink Length

4.4. Coder Enhancements

As discussed in *Section 2.2.3. Basic Voice Features Supported*, Play and Record functionality for additional coders is now included with Dialogic Voice Software. These coders are included with the player (**dx_playiottdata( )** function) and recorder (**dx_reciottdata( )** function) in the voice library.

4.4.1. DX_XPB Parameters

The DX_XPB structure specifies the file format, data format (coder), sampling rate, and resolution for all audio files associated with a DX_IOTT table. It is used by the following play and record functions: **dx_playiottdata( )** and **dx_reciottdata( )**.

Parameter	Description	
wFileFormat	Specifies one of the following audio file formats. Note that this field is ignored by the convenience functions **dx_recwav( )**, **dx_recvox( )**, and **dx_playvox( )**.	
	FILE_FORMAT_VOX	Dialogic VOX file format
	FILE_FORMAT_WAVE	Microsoft WAVE file format
wDataFormat	Specifies one of the following data formats:	
	DATA_FORMAT_DIALOGIC_ADPCM	4-bit OKI ADPCM (Dialogic registered)
	DATA_FORMAT_MULAW or DATA_FORMAT_G711_MULAW	8-bit mu-law G.711 PCM
	DATA_FORMAT_ALAW or DATA_FORMAT_G711_ALAW	8-bit a-law G.711 PCM
	DATA_FORMAT_PCM	8-bit or 16-bit Linear PCM
	DATA_FORMAT_G721	G.721

Parameter	Description	
	DATA_FORMAT_G726	G.726 bit-exact coder
	DATA_FORMAT_GSM610_ MICROSOFT	GSM 6.10 full-rate coder (Microsoft Windows compatible format)*
	DATA_FORMAT_GSM610_ TIPHON	GSM 6.10 VOX full-rate coder (TIPHON format)
	DATA_FORMAT_ TRUESPEECH	TrueSpeech data format
nSamplesPerSec	Specifies one of the following sampling rates:	
	DRT_6KHZ	6 kHz sampling rate.
	DRT_8KHZ	8 kHz sampling rate.
	DRT_11KHZ	11 kHz sampling rate. Note: 11kHz OKI ADPCM is not supported.
wBitsPerSample	Specifies the number of bits per sample. This number varies with the data format. For more information, refer to the following sections on coder support.	

*Microsoft Windows Media Recorder Audio Compression Codec: GSM 6.10 Audio CODEC

G.711 Voice Coder Support

The G.711 voice coder is supported through the following parameters in the DX_XPB structure:

Parameter	Possible Values
wFileFormat	FILE_FORMAT_WAV
	FILE_FORMAT_VOX
wDataFormat	DATA_FORMAT_G711_ALAW or DATA_FORMAT_ALAW
	DATA_FORMAT_G711_MULAW or

Parameter	Possible Values
	DATA_FORMAT_MULAW
nSamplesPerSec	DRT_6KHZ
	DRT_8KHZ
wBitsPerSample	8 (48 Kb/s or 64 Kb/s)

G.721 Voice Coder Support

The G.721 voice coder is supported through the following parameters in the DX_XPB structure:

Parameter	Possible Values
wFileFormat	FILE_FORMAT_WAV
	FILE_FORMAT_VOX
wDataFormat	DATA_FORMAT_G721
nSamplesPerSec	DRT_8KHZ
wBitsPerSample	4 (32 Kb/s)

Linear PCM Voice Coder Support

The linear PCM voice coder is supported through the following parameters in the DX_XPB structure:

Parameter	Possible Values
wFileFormat	FILE_FORMAT_WAV
	FILE_FORMAT_VOX
wDataFormat	DATA_FORMAT_PCM
nSamplesPerSec	DRT_11KHZ
wBitsPerSample	8 (88 Kb/s)
	16 (176 Kb/s)

OKI ADPCM Voice Coder Support

The OKI ADPCM voice coder is supported through the following parameters in the DX_XPB structure:

Parameter	Possible Values
wFileFormat	FILE_FORMAT_WAV FILE_FORMAT_VOX
wDataFormat	DATA_FORMAT_DIALOGIC_ADPCM
nSamplesPerSec	DRT_6KHZ DRT_8KHZ
wBitsPerSample	4 (24 Kb/s or 32 Kb/s)

G.726 Voice Coder Support

The G.726 voice coder is supported through the following parameters in the DX_XPB structure:

Parameter	Possible Values
wFileFormat	FILE_FORMAT_WAV FILE_FORMAT_VOX
wDataFormat	DATA_FORMAT_G726
nSamplesPerSec	DRT_8KHZ
wBitsPerSample	2, 3, 4, or 5 (16, 24, 32, or 40 Kb/s, respectively.)

GSM Voice Coder Support

The GSM voice coder is supported through the following parameters in the DX_XPB structure:

Parameter	Possible Values
wFileFormat	FILE_FORMAT_WAV (supported only with DATA_FORMAT_GSM610_MICROSOFT) FILE_FORMAT_VOX

Parameter	Possible Values
wDataFormat	DATA_FORMAT_GSM610_MICROSOFT
	DATA_FORMAT_GSM610_TIPHON
nSamplesPerSec	DRT_8KHZ
wBitsPerSample	0 (13 Kb/s.)

TrueSpeech Voice Coder Support

The TrueSpeech voice coder is supported through the following parameters in the DX_XPB structure:

Parameter	Possible Values
wFileFormat	FILE_FORMAT_WAV
	FILE_FORMAT_VOX
wDataFormat	DATA_FORMAT_TRUESPEECH
nSamplesPerSec	DRT_8KHZ
wBitsPerSample	0 (8.5 Kb/s)

Example

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

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

4.5. New Silence Compression Record Functionality

The new Silence Compression Record (SCR) algorithm is based on energy detection and zero crossing. This SCR uses different parameters than the standard R4 SCR. Specifically, the “Pre-Compensation and “De-Glitch” parameters are no longer needed, and there are additional new parameters.

The SCR algorithm operates on one msec blocks of speech and uses a two-fold approach to determine whether a sample is speech or silence. Two Probability of

Speech values are calculated using a zero crossing algorithm and an energy detection algorithm. These values are put together to calculate a Combined Probability of Speech.

The Energy Detection algorithm allows you to modify the background noise threshold range. Signals above the high threshold are declared Speech, and signals below the low threshold are declared silence.

Speech or silence is declared based on the previous sample, the current Combined Probability of Speech in relation to the Speech Probability Threshold and Silence Probability Threshold parameters and the Trailing Silence parameter.

See *Table 7. Voice Board and Channel Parameters Supported* for the parameters used with **dx_setparm( )** to turn SCR ON/OFF. When enabled, voice record functions automatically record with SCR. For information on modifying SCR parameters, see the *DM3 Configuration File Reference*.

4.6. Voice API Call Progress Analysis Support

The voice API supports Call Progress Analysis as follows:

- The R4 voice API on DM3 supports Call Progress Analysis through the **dx_dial( )** function.
- The **dx_dial( )** function does not support dial tone detection. Therefore, the PerfectCall dial tone detection tone definitions are not used: TID_DIAL_LCL, TID_DIAL_INTL, and TID_DIAL_XTRA.
- The supported special control characters in the dial string are T (DTMF tone dial mode, default), M (MF tone dial mode), and comma (, pause for 2 seconds). The other special control characters--&, P, L, I, and X--are either ignored or return an error. For example, P (Pulse dial mode) returns an error; the others (L, I, X) are ignored. All valid DTMF (0-9, a-d, *, #) and MF (0-9, a-c, *, #) are supported. **dx_dial( )** fails if 'd' is present in MF dial string (as d is not a valid MF character).

  NOTE: MF is supported for dialing, but not for detection.
- Only the following fields of the DX_CAP structure are supported:

Field Name	Default	
ca_intflg	DX_OPTEN	
ca_noanswer	3000	Units: 10 ms
ca_cnosig	4000	Units: 10 ms
ca_pamd_failtime	400	Units: 10 ms

- The default PerfectCall Call Progress Analysis tones that are used by **dx_dial( )** cannot be modified (for example, BUSY, BUSY_FAST, and so on). That is, **dx_chgdur( )**, **dx_chgfreq( )**, **dx_chgrepcnt( )**, and **dx_initcallp( )** are not supported. Note that these default tones differ from those used in R4 and previous releases of R4 on DM3. Complete data on the supported tone templates can be found in *Appendix A*.

The extended attribute functions that provide call analysis information and are supported on R4 on DM3 do not return information related to functionality that is not supported by R4 on DM3; for example:

- **ATDX_CONNTYPE( )** connection type CON_LPC
- **ATDX_CPTERM( )** termination reason CR_NODIALTONE

4.6.1. Support and Scenarios for Call Analysis with dx_dial( )

Table 9 provides information on what specific Call Analysis scenarios are supported with the voice function **dx_dial( )**. To invoke **dx_dial( )** under CAS, your application must wait for the connected event. This method is available regardless of the protocol being used; however, some restrictions apply when using DM3 CAS protocols. The restrictions are due to the fact that the voice capability is shared between the network device and the voice channel during the call setup time.

NOTE: The information in this section also applies to DI products, which are considered to use CAS protocols.

Table 9. dx_dial( ) Call Analysis Support

CA Feature	dx_dial( ) Support in R4 on DM3	Comments
Busy	√	CAS protocols: not supported
No Ringback	√	CAS protocols: not supported
SIT	√	CAS protocols: not supported
No answer	√	CAS protocols: not supported
Cadence break	√	CAS protocols: not supported
Discarded	None	
NA	None	
Unknown	None	
PVD	√	CAS protocols: Wait for the GCEV_CONNECTED event. Only the following fields of the DX_CAP structure are supported: ca_intflg, ca_cnosig, ca_noanswer, and ca_pamd_failtime.

CA Feature	dx_dial( ) Support in R4 on DM3	Comments
PAMD	√	CAS protocols: Wait for the GCEV_CONNECTED event. Only the following fields of the DX_CAP structure are supported: ca_intflg, ca_cnosig, ca_noanswer, and ca_pamd_failtime.
FAX	√	CAS protocols: Wait for the GCEV_CONNECTED event. Only the following fields of the DX_CAP structure are supported: ca_intflg, ca_cnosig, ca_noanswer, and ca_pamd_failtime.
In progress	None	

For information on using GlobalCall for Call Analysis, see *Section 5.6. Call Progress and Call Analysis*.

4.7. Support for ADSI 2-Way FSK

ADSI (Analog Display Services Interface) 2-way FSK (Frequency Shift Keying) is supported. Because this functionality was previously not supported in Linux, the three pertinent functions—that is, **dx_RxIottData( )**, **dx_TxIottData( )**, **dx_TxRxIottData( )**—are described here in their entirety. The ADSI_XFERSTRUC data structure which is used by these functions is also described here.

In support of ADSI 2-way FSK, the following has been added to the voice API in addition to the new functions:

- DX_MAXDATA termination condition, stored in the **tp_termno** field of the DV_TPT data structure. Specify a valid value in **tp_length** field. Valid values are 1 through 65535. A Transmit/Receive FSK session is terminated when the specified value of FSK DX_MAXDATA (in bytes) is transmitted/received.
- TM_MAXDATA return value, returned by **ATDX_TERMMSK( )** when the last I/O function terminates on DX_MAXDATA.
- DXCH_FSKINTERBLKTIMEOUT channel parameter, set and obtained by the **dx_setparm( )** and **dx_getparm( )** functions, respectively. Measured in milliseconds. The firmware gets FSK data in bursts. This parameter specifies how long the firmware should wait for the next burst of FSK data before it can conclude that no more data will be coming and can terminate the receive session. In short, this parameter denotes the maximum time between any two FSK data bursts in one receive session. This property can only be supplied for reception of FSK data with **dx_RxIottdata( )**.

NOTE: To maintain compatibility with the way other termination conditions are specified, you can also specify TF_MAXDATA in the **tp_flags** field of the DV_TPT, but the library does not take note of that flag.

4.7.1. ADSI_XFERSTRUC: ADSI Data Buffer

The **ADSI_XFERSTRUC** data structure contains parameters for the reception and transmission of ADSI data.

The ADSI_XFERSTRUC is used to specify parameters for ADSI data transmission and reception using the **dx_RxIottData( )**, **dx_TxIottData( )**, and **dx_TxRxIottData( )** functions.

This structure is declared as follows:

```
typedef struct_ADSI_XFERSTRUC{

        UINT          cbSize;
        DWORD         dwTxDataMode;
        DWORD         dwRxDataMode;

}ADSI_XFERSTRUC;
```

Voice API for DM3

The parameters used by this structure are:

Parameter	Description
cbSize	Specifies the size of the structure, in bytes
dwTxDataMode	Specifies one of the following data transmission modes: • ADSI_ALERT for FSK with Alert (CAS) • ADSI_NOALERT for FSK without Alert (CAS) • ADSI_ONHOOK_SEIZURE for On-Hook with Seizure • ADSI_ONHOOK_NOSEIZURE for On-Hook without Seizure
dwRxDataMode	Specifies one of the following data reception modes: • ADSI_ALERT for FSK with Alert (CAS) • ADSI_NOALERT for FSK without Alert (CAS) • ADSI_ONHOOK_SEIZURE for On-Hook with Seizure • ADSI_ONHOOK_NOSEIZURE for On-Hook without Seizure

4.7.2. ADSI API Functions

The **dx_RxIottData( )**, **dx_TxIottData( )**, **dx_TxRxIottData( )** functions are described next.

Name:	int dx_RxIottData(chdev, iottp, lpTerminations, wType, lpParams, mode)	
Inputs:	int chdev	• valid channel device handle
	DX_IOTT *iottp	• pointer to I/O transfer table
	DV_TPT *lpTerminations	• pointer to termination parameter table
	int wType	• data type
	LPVOID lpParams	• pointer to data type-specific information
	USHORT mode	• function mode
Returns:	0 if successful -1 if error	
Includes:	dxxxlib.h srllib.h	
Mode:	Synchronous/asynchronous	

■ Description

The **dx_RxIottData( )** function is used to receive data on a specified channel. The data may be directed to any combination of data files, memory, or custom devices. The **wType** parameter specifies the type of data to be received, for example ADSI data.

After **dx_RxIottData( )** is called, data reception continues until one of the following occurs:

- **dx_stopch( )** is called
- the data requirements specified in the DX_IOTT are fulfilled
- the channel detects end of FSK data
- one of the conditions in the DV_TPT is satisfied

If the channel detects end of FSK data, the function is terminated. Use **ATDX_TERMMSK( )** to return the cause of termination. Possible return values are:

TM_MAXTIME	Maximum function time exceeded
TM_USRSTOP	Function stopped by user
TM_EOD	End of FSK data detected on receive
TM_MAXDATA	Maximum data reached
TM_ERROR	I/O device error

To obtain the total length of FSK data received (in bytes), call **ATDX_TRCOUNT( )** after **dx_RxIottData( )** has completed successfully.

Upon asynchronous completion of **dx_RxIottData( )**, the TDX_RXDATA event is posted.

Parameter	Description
chdev	The valid Dialogic channel device handle.
iottp	The pointer to the I/O Transfer Table. The **iottp** parameter specifies the destination for the received data. This is the same DX_IOTT structure used in **dx_playiottdata( )** and **dx_reciottdata( )**.
lpTerminations	The pointer to the DV_TPT Termination Parameter Table that sets the termination conditions for the device handle. Supported values are: • DX_MAXTIME. For more information, see the DV_TPT structure in the *Voice Software Reference*. • DX_MAXDATA. Specify a valid value in the **tp_length** field of the DV_TPT structure. Valid values are 1 through 65535.
wType	Specifies the type of data to be received. To receive ADSI data, set **wType** to DT_ADSI.
lpParams	The pointer to information specific to the data type specified in **wType**. The format of the parameter block depends on **wType**. For ADSI data, set **lpParams** to point to an ADSI_XFERSTRUC structure.

Parameter	Description
mode	Specifies how the function should execute, either EV_ASYNC (asynchronous) or EV_SYNC (synchronous).

■ Cautions

Library level data is buffered when it is received. The buffer size is 255, which is the default buffer size used by the library.

■ Example

```
// Synchronous receive ADSI data

DX_IOTT iott = {0};
char *devnamep = "dxxxB1C1";
char buffer[16];
ADSI_XFERSTRUC adsimode;
DV_TPT tpt;
int chdev;

         .
         .
         .

    sprintf(buffer, "RECEIVE.ADSI");
    if ((iott.io_fhandle = dx_fileopen(buffer, O_BINARY)) == -1) {
         /* Perform system error processing */
         exit(2);
    }

    if ((chdev = dx_open(devnamep, 0)) == -1) {
         fprintf(stderr, "Error opening channel %s\n",devnamep);
         dx_fileclose(iott.io_fhandle);
         exit(1);
    }

         .
         .
         .

    // destination is a file
    iott.io_type = IO_DEV|IO_EOT;
    iott.io_bufp = 0;
    iott.io_offset = 0;
    iott.io_length = -1;

    adsimode.cbSize = sizeof(adsimode);
    adsimode.dwRxDataMode = ADSI_NOALERT;

    // Wait for inbound call.

    // Specify maximum time termination condition in the TPT.
    // Application specific value is used to terminate dx_RxIottData( )
```

```
// if end of data is not detected over a specified duration.
tpt.tp_type = IO_EOT;
if (dx_clrtpt(&tpt, 1) == -1) {
   // Process error
}
tpt.tp_termno = DX_MAXTIME;
tpt.tp_length = 1000;
tpt.tp_flags = TF_MAXTIME;

if (dx_RxIottData(chdev, &iott, &tpt, DT_ADSI, &adsimode, EV_SYNC) < 0) {
     fprintf(stderr, "ERROR: dx_RxIottData failed on Channel %s; "
              "error: %s\n", ATDV_NAMEP(chdev), ATDV_ERRMSGP(chdev));
}
```

■ Errors

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

Possible error codes from the **dx_RxIottData( )** function include the following:

Error Code	Description
EDX_BADPARM	Invalid data mode
EDX_BADIOTT	Invalid DX_IOTT (pointer to I/O transfer table)
EDX_BUSY	Channel already executing I/O function
EDX_SYSTEM	Error from operating system; use **dx_fileerrno( )** to obtain error value

■ See Also

- **dx_TxIottData( )**
- **dx_TxRxIottData( )**

Name:	int dx_TxIottData(chdev, iottp, lpTerminations, wType, lpParams, mode)	
Inputs:	int chdev	• valid channel device handle
	DX_IOTT *iottp	• pointer to I/O transfer table
	DV_TPT *lpTerminations	• pointer to termination parameter table
	int wType	• data type
	LPVOID lpParams	• pointer to data type-specific information
	USHORT mode	• function mode
Returns:	0 if successful -1 if error	
Includes:	dxxxlib.h srllib.h	
Mode:	Synchronous/asynchronous	

■ Description

The **dx_TxIottData( )** function is used to transmit data on a specified channel. The data may come from any combination of data files, memory, or custom devices. The **wType** parameter specifies the type of data to be transmitted, for example ADSI data. The **iottp** parameter specifies the messages to be transmitted.

Upon asynchronous completion of **dx_TxIottData( )**, the TDX_TXDATA event is posted. Use **ATDX_TERMMSK( )** to return the reason for the last I/O function termination on the channel **chdev**. Possible return values are:

TM_MAXTIME	Maximum function time exceeded
TM_USRSTOP	Function stopped by user
TM_EOD	End of FSK data reached on transmit
TM_MAXDATA	Maximum data reached
TM_ERROR	I/O device error

Parameter	Description
chdev	The valid Dialogic channel device handle returned from **dx_open( )**.
iottp	The pointer to the I/O Transfer Table structure. The source of message(s) to be transmitted is specified by this transfer table. This is the same DX_IOTT structure used in **dx_playiottdata( )** and **dx_reciottdata( )**.
lpTerminations	The pointer to the DV_TPT Termination Parameter Table that sets the termination conditions for the device handle. Supported values are: • DX_MAXTIME. For more information, see the DV_TPT structure in the *Voice Software Reference*. • DX_MAXDATA. Specify a valid value in the **tp_length** field of the DV_TPT structure. Valid values are 1 through 65535.
wType	Specifies the type of data to be transmitted. To transmit ADSI data, set **wType** to DT_ADSI.
lpParams	The pointer to information specific to the data type specified in **wType**. The format of the parameter block depends on **wType**. For ADSI data, set **lpParams** to point to an ADSI_XFERSTRUC structure.
mode	Specifies how the function should execute, either EV_ASYNC (asynchronous) or EV_SYNC (synchronous).

■ Example

```
// Synchronous transmit ADSI data

DX_IOTT iott = {0};
char *devnamep = "dxxxB1C1";
char buffer[16];
ADSI_XFERSTRUC adsimode;
int chdev;

          .
          .
          .

    sprintf(buffer, "MENU.ADSI");
    if ((iott.io_fhandle = dx_fileopen(buffer, _O_RDONLY|O_BINARY)) == -1) {
```

```
        /* Perform system error processing */
        exit(1);
    }

    if ((chdev = dx_open(devnamep, 0)) == -1) {
        fprintf(stderr, "Error opening channel %s\n",devnamep);
        dx_fileclose(iott.io_fhandle);
        exit(2);
    }

    // source is a file
    iott.io_type = IO_DEV|IO_EOT;
    iott.io_bufp = 0;
    iott.io_offset = 0;
    iott.io_length = -1;

    adsimode.cbSize = sizeof(adsimode);
    adsimode.dwTxDataMode = ADSI_ALERT;  // send out ADSI data with CAS


    // Wait for inbound call

    if (dx_TxIottData(chdev, &iott, NULL, DT_ADSI, &adsimode, EV_SYNC) < 0) {
        fprintf(stderr, "ERROR: dx_TxIottData failed on Channel %s; "
            "error: %s\n",  ATDV_NAMEP(chdev), ATDV_ERRMSGP(chdev));
    }
```

■ Errors

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

Possible error codes from the **dx_TxIottData( )** function include the following:

Error Code	Description
EDX_BADPARM	Invalid data mode
EDX_BADIOTT	Invalid DX_IOTT (pointer to I/O transfer table)
EDX_BUSY	Channel already executing I/O function
EDX_SYSTEM	Error from operating system; use **dx_fileerrno( )** to obtain error value

■ See Also

- **dx_RxIottData( )**
- **dx_TxRxIottData( )**

Name:	int dx_TxRxIottData(chdev, lpTxIott, lpTxTerminations, lpRxIott, lpRxTerminations. wType, lpParams, mode)	
Inputs:	int chdev	• valid channel device handle
	DX_IOTT *lpTxIott	• pointer to I/O transfer table
	DV_TPT *lpTxTerminations	• pointer to termination parameter table
	DX_IOTT *lpRxIott	• pointer to I/O transfer table
	DV_TPT *lpRxTerminations	• pointer to termination parameter table
	int wType	• data type
	LPVOID lpParams	• pointer to data type-specific information
	USHORT mode	• function mode
Returns:	0 if successful -1 if error	
Includes:	dxxxlib.h srllib.h	
Mode:	Synchronous/asynchronous	

Description

The **dx_TxRxIottData( )** function is used to start a transmit-initiated reception of data (two-way ADSI), where faster remote terminal device (CPE) turnaround occurs, typically within 100 msec. Faster turnaround is required for two-way FSK so that the receive data is not missed while the application turns the channel around after the last sample of FSK transmission is sent.

The **wType** parameter specifies the type of data that is to be transmitted and received, i.e., two-way ADSI. The transmitted data may come from, and the received data may be directed to, any combination of data files, memory, or custom devices. The data is transmitted and received on a specified channel.

The **lpTxIott** parameter specifies the location of the messages to be transmitted. The destination for the retrieved messages is specified by **lpRxIott**.

The transmit portion of the **dx_TxRxIottData( )** function continues until one of the following occurs:

- all data specified in DX_IOTT has been transmitted
- **dx_stopch( )** is issued on the channel
- one of the conditions specified in DV_TPT is satisfied

The receive portion of the **dx_TxRxIottData( )** function continues until one of the following occurs:

- **dx_stopch( )** is called
- the data requirements specified in the DX_IOTT are fulfilled
- the channel detects end of FSK data
- one of the conditions in the DV_TPT is satisfied

If the channel detects end of FSK data during the receive portion, the function is terminated. Use **ATDX_TERMMSK( )** to return the cause of termination. Possible return values are:

TM_MAXTIME	Maximum function time exceeded
TM_USRSTOP	Function stopped by user
TM_EOD	End of FSK data detected on receive
TM_MAXDATA	Maximum data reached
TM_ERROR	I/O device error

Upon asynchronous completion of the transmit portion of the function, a TDX_TXDATA event is generated. Upon asynchronous completion of the receive portion of the function, a TDX_RXDATA event is generated.

Parameter	Description
chdev	The valid Dialogic channel device handle.
lpTxIott	The pointer to the I/O Transfer Table. **lpTxIott** specifies the source of the messages to be transmitted. This is the

Parameter	Description
	same DX_IOTT structure used in **dx_playiottdata( )** and **dx_reciottdata( )**.
lpTxTerminations	The pointer to the DV_TPT Termination Parameter Table that sets the termination conditions for the device handle. Supported values are: • DX_MAXTIME. For more information, see the DV_TPT structure in the *Voice Software Reference*. • DX_MAXDATA. Specify a valid value in the **tp_length** field of the DV_TPT structure. Valid values are 1 through 65535.
lpRxIott	The pointer to the I/O Transfer Table structure. **lpRxIott** specifies the destination of the messages to be received. This is the same DX_IOTT structure used in **dx_playiottdata( )** and **dx_reciottdata( )**.
lpRxTerminations	The pointer to the DV_TPT Termination Parameter Table that sets the termination conditions for the device handle. Supported values are: • DX_MAXTIME. For more information, see the DV_TPT structure in the *Voice Software Reference*. • DX_MAXDATA. Specify a valid value in the **tp_length** field of the DV_TPT structure. Valid values are 1 through 65535.
wType	Specifies the type of data to be transmitted and received. To transmit and receive ADSI data, set **wType** to DT_ADSI.
lpParams	The pointer to a structure that specifies additional information about the data that is to be sent and received. The structure type is determined by the data type (ADSI) specified by **wType**. For ADSI data, set **lpParams** to point to an ADSI_XFERSTRUC parameter block structure.
mode	Specifies how the function should execute, either EV_ASYNC (asynchronous) or EV_SYNC (synchronous).

Cautions

Library level data is buffered when it is received. The buffer size is 255, which is the default buffer size used by the library.

Example

```
// Synchronous transmit initiated receive ADSI data

DX_IOTT TxIott = {0};
DX_IOTT RxIott = {0};
DV_TPT tpt;
char *devnamep = "dxxxB1C1";
char buffer[16];
ADSI_XFERSTRUC adsimode;
int chdev;

         .
         .
         .

    sprintf(buffer, "MENU.ADSI");
    if ((TxIott.io_fhandle = dx_fileopen(buffer, _O_RDONLY|O_BINARY)) == -1) {
         /* Perform system error processing */
         exit(1);
    }

    sprintf(buffer, "RECEIVE.ADSI");
    if ((RxIott.io_fhandle = dx_fileopen(buffer, O_BINARY)) == -1) {
         /* Perform system error processing */
         dx_fileclose(TxIott.io_fhandle);
         exit(2);
    }

    if ((chdev = dx_open(devnamep, 0)) == -1) {
         fprintf(stderr, "Error opening channel %s\n",devnamep);
         dx_fileclose(TxIott.io_fhandle);
         dx_fileclose(RxIott.io_fhandle);
         exit(1);
    }

         .
         .
         .

    // source is a file
    TxIott.io_type = IO_DEV|IO_EOT;
    TxIott.io_bufp = 0;
    TxIott.io_offset = 0;
    TxIott.io_length = -1;

    // destination is a file
    RxIott.io_type = IO_DEV|IO_EOT;
    RxIott.io_bufp = 0;
    RxIott.io_offset = 0;
    RxIott.io_length = -1;

    adsimode.cbSize = sizeof(adsimode);
```

Voice API for DM3

```
adsimode.dwTxDataMode = ADSI_ALERT;
adsimode.dwRxDataMode = ADSI_NOALERT;

// Specify maximum time termination condition in the TPT for the
// receive portion of the function.  Application specific value is
// used to terminate dx_TxRxIottData( ) if end of data is not
// detected over a specified duration.
tpt.tp_type = IO_EOT;
if (dx_clrtpt(&tpt, 1) == -1) {
   // Process error
}
tpt.tp_termno = DX_MAXTIME;
tpt.tp_length = 1000;
tpt.tp_flags = TF_MAXTIME;

// Wait for inbound call
if (dx_TxRxIottData(chdev, &TxIott, NULL, &RxIott, &tpt, DT_ADSI,
     &adsimode, EV_SYNC) < 0) {
     fprintf(stderr, "ERROR: dx_TxRxIottData failed on Channel %s; "
            "error: %s\n", ATDV_NAMEP(chdev), ATDV_ERRMSGP(chdev));
}
```

■ Errors

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

Possible error codes from the **dx_TxRxIottData( )** function include the following:

Error Code	Description
EDX_BADPARM	Invalid data mode
EDX_BADIOTT	Invalid DX_IOTT (pointer to I/O transfer table)
EDX_BUSY	Channel already executing I/O function
EDX_SYSTEM	Error from operating system; use **dx_fileerrno( )** to obtain error value

■ See Also

- **dx_TxIottData( )**
- **dx_RxIottData( )**

5. R4 GlobalCall API for DM3

The following is a summary of the restrictions and limitations when using the R4 GlobalCall API for DM3.

R4 for DM3 supports all GlobalCall API functions except those noted in *Table 12. List of GlobalCall API Functions Restricted and Not Supported* .

5.1. GlobalCall Features Not Supported by R4 for DM3

The following GlobalCall features, which are documented in the *GlobalCall Application Developer's Guide*, are **not** supported by R4 on DM3:

- GlobalCall Alarm Management System (GCAMS)
- Feature Transparency and Extension (FTE)
- Real Time Configuration Management (RTCM)
- GlobalCall Service Request (GCSR)

5.2. GlobalCall Call State Model Restrictions

R4 on DM3 does not support the following GlobalCall call states:

- GCST_GETMOREINFO
- GCST_SENDMOREINFO
- GCST_CALLROUTING
- GCST_PROCEEDING

5.3. gc_Open( ) and gc_OpenEx( ) Restrictions

In R4 applications that use DM3 flexible routing configurations, it is possible to specify a network device, protocol, and voice device in an ASCII string pointed to by the **devicename** parameter in **gc_OpenEx( )** and **gc_Open( )** commands.

An example of the **devicename** parameter is as follows:

```
:N_dtiB1T1:P_ISDN:V_dxxxB1C1
```

where:

- N_ denotes a network timeslot device, in this example, dtiB1T1
- P_ denotes a protocol, in this example, an ISDN protocol
- V_ denotes a voice channel device, in this example, dxxxB1C1

The following restrictions apply:

- The network timeslot device field (N_) is required.
- The protocol identifier field (P_) is not used since, for DM3 boards, the protocol is determined at board initialization time and not when a GlobalCall device is opened. However, to provide compatibility with R4 on earlier-generation boards, the protocol identifier can be specified.
- The voice channel device field (V_) is optional with the following consequences:
  - Attachment to different types of DM3 voice devices is dependent on the protocol downloaded. For example, if one board has ISDN for protocols and another has T1 CAS, the T1 CAS network devices cannot be attached to the voice devices on the ISDN board. For more information on limitations, see section *2.3.5. Determining Channel Capabilities in Flexible Routing Configurations*.
  - If the voice channel device is specified, it is automatically associated with the specified network device (it is attached and routed).
  - If the voice channel device is not specified, the application may include function calls, typically, **dx_open( )**, **gc_Attach( )**, **gc_GetNetworkH( )**, and **nr_scroute( )*** to manually associate the voice channel device with the network device. This option may be desirable when porting an existing ISDN application that uses earlier-generation boards to an application that uses DM3 boards.

    ***Note:** Routing can also be accomplished using **gc_Listen( )** and **gc_GetXmitSlot( )**.

See *Section 11.3.1. Initializing the GlobalCall API for DM3 Boards Only (Flexible Routing)* for examples.

5.4. Associating Network and Voice Devices

For R4 applications that use DM3 flexible routing configurations, it is possible to open a line device using the **gc_Open( )** and **gc_OpenEx( )** functions, open a voice device using the **dx_open( )** function, then use the **gc_Attach( )** function to associate the voice device with the line device, using the line device ID and the voice device handle. See *Section 11.3.1. Initializing the GlobalCall API for DM3 Boards Only (Flexible Routing)* for more information.

5.5. Analog Call Analysis

The **gc_LoadDxParm( )** is not supported under R4 for DM3.

NOTE: The **gc_LoadDxParm( )** function is used for analog call analysis only.

5.6. Call Progress and Call Analysis

Call analysis consists of both pre-connect and post-connect information about the progress of the call. Pre-connect call progress determines the status of the call connection, that is, busy, no dial tone, no ringback, etc. Post-connect call analysis, which is also known as "media type detection," determines the destination party's media type, that is, answering machine, fax, modem, voice, etc.

NOTES:
1. In GlobalCall terminology, the term call analysis is used interchangeably with the term call progress.
2. The information in this section also applies to DI products, which are considered to use CAS protocols.

There are two methods available for Call Analysis in R4 on DM3: the GlobalCall method and the **dx_dial( )** method.

The GlobalCall media detection method is especially useful for performing **post-connect call analysis**. When activated by setting the **GCPR_MEDIADETECT** parameter to GCPV_ENABLE for a particular channel, post-connect call analysis is performed as part of the **gc_MakeCall( )** function's operation. The **gc_MakeCall( )** function is used to place a call, the signal detector analyzes the incoming signals to perform call progress analysis.

After the normal **gc_MakeCall( )** processing finishes and **GCEV_CONNECTED** event is sent, call analysis runs and generates a **GCEV_MEDIADETECTED** event that tells the application the result of the analysis (for example, FAX, PVD, or PAMD is detected).

The outcome of the analysis determines the events generated and the action that can be taken as follows:

- If the call is successful, **gc_MakeCall( )** finishes and a **GCEV_CONNECTED** event is sent, call analysis runs, and generates a **GCEV_MEDIADETECTED** event. The **gc_ResultValue( )** and **gc_GetCallInfo( )** functions can then be used to get more information about the type of media detected, such as voice, answering machine, and fax. See *Table 11* for details on the information being detected.
- If the call is not successful--for example, there is no ringback--a GCEV_DISCONNECTED event is generated and the **gc_ResultValue( )** function can be used to retrieve the reason for the failure. See *Appendix B GlobalCall Error Code and Result Value Summary* in the *GlobalCall API Software Reference* manual and the *gcerr.h* file for more information.

NOTE: The information above applies when using **gc_MakeCall( )** in asynchronous or synchronous mode. However, in synchronous mode, since the **gc_MakeCall( )** function must complete, the GCEV_MEDIADETECTED event is generated after the call is connected.

GCPR_MEDIADETECT parameter settings actually allow the application to specify whether pre- or post-connect call analysis or both should be activated. This method for achieving this is shown in the following table:

Table 10. GCPR_MEDIADETECT Settings

GCPR_MEDIADETECT	ISDN Result (Fixed Routing Only)	CAS Result
GCPV_DISABLE (Default)	No call progress	Pre-connect call progress **only**
GCPV_ENABLE	Full call progress	Full call progress

As can be seen in this table, the default behavior (GCPR_MEDIADETECT=GCPV_DISABLE) disables media detection for ISDN, but it actually activates pre-connect call progress for CAS protocols. To enable full call progress analysis for either ISDN or CAS, set the **GCPR_MEDIADETECT** parameter to GCPV_ENABLE for the respective channel. **ISDN support is provided** in **fixed routing configurations only.**

NOTE: For this GlobalCall media detection to work, a voice device must be attached to the line device and properly routed. Failure to do so will cause subsequent outgoing call attempts to fail.

5.6.1. Support and Scenarios for Call Analysis with GlobalCall

Table 11 explains call analysis support via the GlobalCall interface. The table applies to DM3 CAS protocols with flexible routing clusters, providing a voice device is attached to the DTI device. The table also applies to any fixed routing configuration regardless of the protocol. DM3 CAS protocols include the PDK protocols; however, check on a protocol-by-protocol basis, as some might not support CA at all.

Table 11. CA Support in Fixed Routing and Flexible Routing with CAS

CA Feature	GC Support in R4 on DM3	How Obtained/Notes
Busy	√	Upon DISCONNECT event, call **gc_ResultValue( )**.
No Ringback	None	
SIT	√	Upon DISCONNECT event, call **gc_ResultValue( )**.
No answer	√	Upon DISCONNECT event, call **gc_ResultValue( )**.
Cadence break	None	GCCT_CAD is not a supported connect type returned by **gc_GetCallInfo( )**.
Discarded	None	GCCT_DISCARDED is not a supported connect type returned by **gc_GetCallInfo( )**.
NA	√	Use GCPR_MEDIADETECT parameter. Upon MEDIADETECTED event, call **gc_GetCallInfo( )**.
Unknown	√	Use GCPR_MEDIADETECT parameter. Upon MEDIADETECTED event, call **gc_GetCallInfo( )**.
PVD	√	Use GCPR_MEDIADETECT parameter. Upon MEDIADETECTED event, call **gc_GetCallInfo( )**.
PAMD	√	Use GCPR_MEDIADETECT parameter. Upon MEDIADETECTED event, call **gc_GetCallInfo( )**.

CA Feature	GC Support in R4 on DM3	How Obtained/Notes
FAX	√	Use GCPR_MEDIADETECT parameter. Upon MEDIADETECTED event, call **gc_GetCallInfo( )**.
In progress	√	Use GCPR_MEDIADETECT parameter. Upon MEDIADETECTED event, call **gc_GetCallInfo( )**.

Note that the Call Analysis timeout parameters values apply, and they are configurable by the host. However, they apply only to post-connect Call Analysis and are not used until the call moves from an initiated to proceeding, alerting, or connected state. The parameters are CaSignalTimeout, CaAnswerTimeout and CaPvdTimeout; their values are found in the CHP section of the configuration file.

The following configurable parameters are used in pre-connect Call Progress:

- **StartTimeout**: Set at the Initiation of a new call (outbound). Reset (disabled) after detection of a wink. This timer is set again for each wink expected (that is, if multiple winks are used, such as after dialing). If this timer expires, the call fails.
- **AnswerTimeout**: Set after detection of the last wink, with time to wait for the call to be answered. If timer expires, call fails. Timer is reset once the call is answered.

This functionality is determined by the protocol, and there are several configurable protocol timers that determine these values. See the *DM3 Configuration File Reference* for details on setting any of the above parameters.

Another option for call analysis is provided by the voice API, which provides post-connect call analysis on DM3 boards through the **dx_dial( )** function. Because there currently is no alerting event to invoke **dx_dial( )** under CAS, your application must wait for the connected event. Note that the GlobalCall and the **dx_dial( )** methods are mutually exclusive, so you must choose one or the other.

With ISDN in flexible routing configurations, the **dx_dial( )** method for call analysis must be used, and both pre-connect call progress and post-connect call analysis are available. See *Section 4.6. Voice API Call Progress Analysis Support* for details on **dx_dial( )** support.

5.7. gc_ResetLineDev( ) Operation

When a timeslot (for example, dtiB1T1) is opened, it is put in the Out-of-service state, blocking all incoming calls on that timeslot. The application must issue a **gc_WaitCall( )** to accept incoming calls or a **gc_MakeCall( )** to make outgoing calls. When **gc_ResetLineDev( )** is issued, it puts the timeslot in the Out-of-service state, disconnecting any existing calls and blocking any further incoming calls. Also, any synchronous call issued on this timeslot--for example, **WaitCall( )--**in the same process is aborted. The **gc_ResetLineDev( )** function does not terminate a synchronous call issued in a different process.

When using ISDN protocols, DM3 products will transmit a SERVICE message to the network when **gc_ResetLineDev( )** is issued.

When a B channel is placed into service, a SERVICE message may be transmitted, depending upon the value of CHP SetParm 0x1312 in the CONFIG file, CHP SetParm 0x1312 controls the sending of SERVICE messages when a B channel is placed into service. This parameter does not affect SERVICE messages sent upon **gc_ResetLineDev( )**. For more information on the CONFIG file settings, see the *DM3 Configuration File Reference*.

5.8. Layer 1 Alarms

For R4 on DM3, GlobalCall does not allow applications to retrieve alarm information that can be used to troubleshoot problems on line devices. See Section *6.2. Detecting Layer 1* Alarms in the DTI Network Interface API chapter for more information on using Layer 1 alarms with R4 on DM3.

5.9. ISDN Feature Implementation and Support

The following information describes the implementation of ISDN-specific features using R4 for DM3 and any feature limitations.

5.9.1. gc_MakeCall( ) Restrictions

For applications that use ISDN protocols, a pointer to a MAKECALL_BLK data structure, defined in the *cclib.h* header file, can be provided in the cclib field of the GC_MAKECALL_BLK data structure. The following parameters in the MAKECALL_BLK data structure are **not** supported :

- **BC_xfer_mode**
- **usr_rate**
- **facility_feature_service**
- **facility_coding_value**
- **USRINFO_ELEM**
- **NFACILITY_ELEM**
- **destination_sub_number_plan**
- **destination_sub_phone_number**
- **origination_sub_number_plan**
- **origination_sub_phone_number**

The **destination_number_type** and **origination_number_type** parameters support the following values only:

- INTL_NUMBER
- NAT_NUMBER
- EN_BLOC_NUMBER (supported by **origination_number_type** parameter only)

NOTE: If a GC_MAKECALL_BLK structure is not specified, the default values are taken from the FCD (Feature Configuration Description) file; see the *DM3 Configuration File Reference*.

See the *GlobalCall ISDN Technology User's Guide* for more information.

NOTE: Neither the **gc_MakeCall( )** function nor the **gc_SetParm( )** function can be used to modify these parameters.

5.9.2. gc_GetNetCRV( ) Restrictions

The **gc_GetNetCRV( )** function is supported, but must be manually enabled for use.

In Linux, to enable this function, add CC.NetCRV Support = 1 in */usr/dialogic/cfg/cheetah.cfg*. To subsequently disable this function, remove this line from the .cfg file.

In Windows, to enable this function, set parameter NetCRV Support to 1 in Key HKEY_LOCAL_MACHINE\SOFTWARE\Dialogic\Cheetah\CC. To subsequently disable this function, set this parameter to 0.

5.9.3. gc_SndMsg( ) Restrictions

For applications that use ISDN protocols, the **gc_SndMsg( )** function supports only the following values for the **msg_type** parameter

- SndMsg_Information
- SndMsg_Congestion
- SndMsg_UsrInformation
- SndMsg_Facility
- SndMsg_Notify
- SndMsg_Status
- SndMsg_StatusEnquiry

See the *GlobalCall ISDN Technology User's Guide* for more information.

5.9.4. Send and Receive Any IE and Any Message

The Send Any IE (Info Element) and Send Any Message features, provided by the **gc_SetInfoElement( )** and **gc_SndMsg( )** functions, are supported by all call control functions, except **gc_ReleaseCall( )**. The Receive Any IE and Receive Any Message features are supported in R4 on DM3 in the same manner as in R4 on earlier-generation products.

5.9.5. Overlap Send

To activate overlap send and prevent the automatic sending of a Sending Complete IE within the SETUP message, the following modifications should be made to the config file for the desired outbound protocol variant requiring overlap send support. The **CalledNumberCount** parameter, which has a default value of zero, should be set to a large positive value. For example, in the ISDN Protocol Variant Definitions section of the configuration file being used, change:

```
Variant CalledNumberCount 99
```

NOTE: You can have more than one CalledNumberCount setting per board, in order to do so, please create a new Variant Define and apply that define using the defineBSet command in the respective TSC section.

See the *DM3 Configuration File Reference* for more information on how to perform the changes outlined above.

A **gc_MakeCall( )** function call that specifies fewer digits than the **CalledNumberCount** results in the sending of a SETUP message that does **not** contain a Sending Complete IE. If more digits are specified, the Sending Complete IE is included in the SETUP message.

The function **gc_SendMoreInfo( )** is not supported, so to send extra digits, the application should wait for the GCEV_SETUP_ACK event, which indicates the inbound side acknowledges the SETUP message, construct an IE block containing the digits to be sent, and then call **gc_SndMsg(GlobalCallDeviceHandle, CRN, SndMsg_Information, &IEBlock)** to send the digits. Even after sending a number of digits greater than **CalledNumberCount**, the Sending Complete IE is not sent automatically.

The following is an example of how to send extra digits using overlap send:

```
void mdfSendOverlap(CH_INFO_PTR chanInfop, char* digits)
{
    IE_BLK info;
    GC_IE_BLK gcInfo;
    char length;
    unsigned char type = 0x00;
    unsigned char plan = 0x00;
    for(length = 0; ; length++, digits++)
    {
        if(*digits)
        {
            info.data[3 + length]= *digits & 0x7F; // Bit 8 set to 0
        }
        else
        {
            break;
        }
    }
    info.data[2] = 0x80 | plan |(type<<4); // Octet 3: Number Type + Numbering Plan
    info.data[1] = length + 1; // Octet 2: Element Length
    info.data[0] = 0x70; // Octet 1: Called Number ID
```

```
    info.length = length + 3; // Information block
    gcInfo.gclib = NULL;
    gcInfo.cclib = &info;
    if(gc_SndMsg(chanInfop->hGC, chanInfop->crn, SndMsg_Information, &gcInfo) < 0)
    {
         mdfError(EGCALL, chanInfop, "gc_SndMsg (SndMsg_Information) failed "
                 "CRN: 0x%X", chanInfop->crn);
    }
}
```

NOTE: Any changes to the CONFIG file for a particular protocol requires the regeneration of the FCD file and the subsequent downloading of the firmware to the boards. The FCDGEN tool, available in the *dialogic\bin* directory, is used to convert a CONFIG file to an FCD file.

5.9.6. Direct Layer 2 Access

R4 on DM3 supports direct layer 2 access on a per trunk basis. Direct layer 2 access is enabled by including the following command in the appropriate [CSS.x] section of the config file, where x identifies a specific trunk (span):

```
Setparm=0x9,1
```

If this command is not included, direct layer 2 access is disabled. Also, using a 0 instead of a 1 in the command above disables direct layer 2 access.

NOTE: Any changes to the CONFIG file for a particular protocol requires the regeneration of the FCD file and the subsequent downloading of the firmware to the boards. The FCDGEN tool, available in the *dialogic\bin* directory, is used to convert a CONFIG file to an FCD file. For more information, see the *DM3 Configuration File Reference.*

GlobalCall supports direct layer 2 access using the **gc_GetFrame( )** and **gc_SndFrame( )** functions.

5.9.7. D Channel Status

In R4 on DM3, a GCEV_D_CHAN_STATUS event is always generated once the board device is initialized and the initial D channel status is known. The resulting value associated with the event indicates this initial D channel status. Any subsequent change in the D channel status is also notified by means of GCEV_D_CHAN_STATUS event. In R4 on earlier-generation boards, when the initial D channel status was UP, no initial event was generated. In R4 on DM3, an initial event is always generated, regardless of the initial status of the D channel.

On download, by default both the trunk and channels are out of service. When the first **gc_Open( )** or **gc_OpenEx( )** is executed on a device, the trunk (D channel) and the channel (B channel) associated with the device are placed into service (trunk in service, channel idle). Although the channel is IDLE, calls cannot be received/processed until **gc_WaitCall( )** is issued. When the application uses **gc_Close( )** to close the channel, the channel returns to out of service, but the trunk remains in service.

The application should use the **gc_ResultValue( )** function to find the reason (UP or DOWN) associated with the GCEV_D_CHAN_STATUS event. A reason of UP indicates that the D channel is active and the **gc_GetFrame( )** and **gc_SndFrame( )** functions can be used to get or send frames respectively. The **gc_GetLinedevState( )** function can be used to retrieve the status of the line device.

See the *GlobalCall API Software Reference* for more information.

5.9.8. B Channel Status

Regarding B channel state, the initial channel state (in service or out of service) is controlled by a CHP parameter (parameter 0x1311) in the CONFIG file. By default, all channels are out of service when a system is initialized. Thereafter, when the application issues **gc_WaitCall( )** the channel (line device) is placed into service. If **gc_ResetLineDev( )** is subsequently issued, the channel is placed out of service until the application issues **gc_WaitCall( )** again.

Also on the channel devices, if **gc_WaitCall( )** is not issued but **gc_MakeCall( )** is issued, the channel is placed into service for the duration of the call. Once the call is released, the channel is once again out of service.

5.10. Handling Multiple Call Objects Per Channel

R4 on DM3 supports the handling of multiple call objects per channel in a glare condition. An application running on bi-directional circuits is capable of handling two CRNs on a single line device, where one call can be in an Idle state, while the other call is in Active state. For example, a glare condition occurs when a call has been dropped but not released and an inbound call is detected. In order to avoid a long delay in processing the inbound call, the GlobalCall library does not wait for the outbound call to be released before notifying the application of the inbound call.

Application	**GlobalCall Library**
gc_MakeCall(**crn1**) -->	
	GCEV_DISCONNECTED(**crn1**) <--
gc_DropCall(**crn1**) -->	
	GCEV_OFFERED(**crn2**) <--
gc_AcceptCall(**crn2**) -->	
	GCEV_DROPCALL(**crn1**) <--
gc_ReleaseCall(**crn1**) -->	

Alternatively, the application can just respond to events using their associated CRN, simply performing a **gc_ReleaseCall( )** upon reception of any GCEV_DROPCALL event whether the CRN is the **active** one or not. Using this procedure, the application only needs to store one CRN per line device.

5.11. Using E-1 CAS R2MF Protocols with R4 on DM3

For R4 on earlier-generation boards, E-1 CAS R2MF protocols (ICAPI and PDK protocols) are provided on the *GlobalCall Protocols* CD that is separately orderable.

For R4 on DM3, you can select E-1 CAS R2MF protocols with the **DM3 PDK Manager**. When configuring DM3 QuadSpan or DualSpan boards to use one or more of the E-1 CAS R2MF protocols, use the PDK Manager utility. The PDK Manager does not integrate the CDP files into the FCD file. Instead, it "hotloads" protocol information separately to the board after the FCD file is downloaded and the board is initialized. See the *DM3 Configuration File Reference* for information on the DM3 PDK Manager.

The E-1 CAS R2MF protocols suitable for use with DM3 products are available as a point release that can be downloaded from the Technical Support website, **http://support.dialogic.com**. The download comprises a zip file that includes the protocol files. The zip file must be unpacked and the resulting files copied to the *\data* directory under the Dialogic home directory, which is *C:\Program Files\Dialogic* by default.

5.12. Country Dependent Parameter (CDP) Files

For T-1 and ISDN protocols, Country Dependent Parameter (CDP) files are not used. Country dependent parameters are specified in the [CHP] section of the CONFIG file. The purpose of the CONFIG file is to define the parameter settings necessary to configure a DM3 hardware/firmware product for a particular feature set. See the *DM3 Configuration File Reference* for more information about the FCDGEN utility. The FCDGEN utility is then used to convert the CONFIG file to a Feature Configuration Description (FCD) file. The FCD file configures the DM3 GlobalCall Resource (including protocol country dependent parameters) as part of the firmware download process.

For the E-1 CAS R2MF protocols that can be downloaded with a point release (see *Section 5.11. Using E-1 CAS R2MF Protocols with R4 on DM3*), *.cdp* files are installed in the *\data* directory under the Dialogic home directory, which is *C:\Program Files\Dialogic* by default.

Caution

For configurations with both DM3 and earlier-generation boards that use the GlobalCall API, country dependent parameter (*.cdp*) files for protocols used with DM3 boards have the same names as the *.cdp* files for protocols used with earlier-generation boards. The DM3 *.cdp* files are located in the *\data* directory (as opposed to the *\cfg* directory for protocols used with earlier-generation boards). When editing a *.cdp* file, be careful you are editing the correct *.cdp* file.

5.13. Using gc_MakeCall( ) with a DI/0408-LS-A

The **gc_MakeCall( )** serves a useful purpose when used with the DI/0408-LS-A product by allowing an application to take an analog loop start interface offhook. To do this, issue a **gc_MakeCall( )** with only a comma character as the dial string. This takes the loop start interface offhook and generates a GCEV_CONNECTED event.

5.14. GlobalCall API Function Restrictions

R4 for DM3 supports all GlobalCall functions except those noted in *Table 12. List of GlobalCall API Functions Restricted and Not Supported* . See the *GlobalCall API Software Reference* for detailed information about each function.

Table 12 lists the GlobalCall functions that are either **not supported** or are supported with restrictions, indicates if a function is line-related or call-related, and references more information if appropriate.

NOTE: If you execute a GlobalCall function that is not supported by DM3 boards, an EGC_UNSUPPORTED error is produced, which corresponds to the error message "Function is not Supported".

Table 12. List of GlobalCall API Functions Restricted and Not Supported

GlobalCall Function	Type		Notes
	Call	Line	
gc_AcceptCall( )	*		**Limitations**: The **rings** parameter is ignored.
gc_AlarmName( )		*	Not supported.
gc_AlarmNumber( )		*	Not supported.
gc_AlarmNumberToName( )		*	Not supported.
gc_AlarmSourceObjectID( )		*	Not supported.

GlobalCall Function	Type		Notes
	Call	Line	
gc_AlarmSourceObjectIDToName( )		*	Not supported.
gc_AlarmSourceObjectName( )		*	Not supported.
gc_AlarmSourceObjectNameToID( )		*	Not supported.
gc_AnswerCall( )	*		**Limitations**: The **rings** parameter is ignored.
gc_Attach( )		*	**Limitations**: DM3 voice resources may only be attached to DM3 network resources; that is, they cannot be attached to earlier-generation devices. Similarly, earlier-generation resources cannot be attached to DM3 devices.
gc_AttachResource( )		*	**Limitations**: See **gc_Attach( )** above.
gc_BlindTransfer( )	*		Not supported.
gc_CallProgress( )	*		Not supported. See section *5.6. Call Progress and Call Analysis* for more information.
gc_CompleteTransfer( )	*		Not supported.
gc_Detach( )		*	**Limitations**: See **gc_Attach( )** above.
gc_Extension( )	*	*	Not supported except for IP. See *GlobalCall IP Technology User's Guide* for more information.

GlobalCall Function	Type		Notes
	Call	Line	
gc_GetAlarmConfiguration( )		*	Not supported.
gc_GetAlarmFlow( )		*	Not supported.
gc_GetAlarmParm( )		*	Not supported.
gc_GetAlarmSourceObject List( )		*	Not supported.
gc_GetAlarmSourceObject NetworkID( )		*	Not supported.
gc_GetBilling( )	*		Not supported.
gc_GetCallInfo( )	*		**Limitation**: ISDN billing information (AOC) is not supported.
gc_GetCallProgressParm( )		*	Not supported.
gc_GetConfigData( )	*	*	Not supported.
gc_GetInfoElem( )	*		Not supported. **Note**: **gc_GetSigInfo( )** must be used as an alternative.
gc_GetNetCRV( )	*		Supported, but must be manually enabled for use. For more information, see *5.9.2. gc_GetNetCRV( ) Restrictions*.
gc_GetParm( )		*	**Limitations:** See **gc_SetParm( )** below.
gc_GetUserInfo( )	*	*	Not supported.
gc_HoldACK( )	*		Not supported.
gc_HoldCall( )	*		Not supported.

GlobalCall Function	Type: Call	Type: Line	Notes
gc_HoldRej( )	*		Not supported.
gc_LoadDxParm( )		*	Not supported. See *Section 5.5. Analog Call Analysis*.
gc_MakeCall( )	*		**Limitations:** See *Section 5.9.1. gc_MakeCall( ) Restrictions* for more information. **Note**: Timeout is supported in both the synchronous and asynchronous modes.
gc_Open( )		*	**Limitations:** See *Sections 5.3. gc_Open( ) and gc_OpenEx( ) Restrictions*, *5.4. Associating Network and Voice Devices*, and *2.3.6. Getting, Using, and Closing Device Handles in a DM3 Flexible Routing Configuration* for more information.
gc_OpenEx( )		*	**Limitations:** See *Sections 5.3. gc_Open( ) and gc_OpenEx( ) Restrictions*, *5.4. Associating Network and Voice Devices*, and *2.3.6. Getting, Using, and Closing Device Handles in a DM3 Flexible Routing Configuration* for more information.
gc_QueryConfigData( )	*	*	Not supported.

GlobalCall Function	Type		Notes
	Call	Line	
gc_ReqANI( )	*		Not supported.
gc_ReqMoreInfo( )	*		Not supported.
gc_ReqService( )	*	*	Not supported.
gc_ResetLineDev( )		*	**Limitations:** See *Section 5.7. gc_ResetLineDev( ) Operation* for more information.
gc_RespService( )	*	*	Not supported.
gc_RetrieveAck( )	*		Not supported.
gc_RetrieveCall( )	*		Not supported.
gc_RetrieveRej( )	*		Not supported.
gc_SendMoreInfo( )	*		Not supported.
gc_SetAlarmConfiguration( )		*	Not supported.
gc_SetAlarmFlow( )		*	Not supported.
gc_SetAlarmNotifyAll( )		*	Not supported.
gc_SetAlarmParm( )		*	Not supported.
gc_SetBilling( )	*		Not supported.
gc_SetCallProgressParm( )		*	Not supported.
gc_SetConfigData( )	*	*	Not supported except for IP. See *GlobalCall IP Technology User's Guide* for more information.

GlobalCall Function	Type: Call	Type: Line	Notes
gc_SetInfoElem( )	*		**Limitations:** Can be used with any call control function except **gc_ReleaseCall( )**. See the *GlobalCall ISDN Technology User's Guide* for more information.
gc_SetParm( )		*	**Limitations:** The supported parameters are: • GCPR_MINDIGITS • GCPR_CALLINGPARTY • GCPR_RECEIVE_INFO_BUF • RECEIVE_INFO_BUF • GCPR_MEDIADETECT (see section *5.6. Call Progress and Call Analysis* for more information)
gc_SetUpTransfer( )	*		Not supported.
gc_SetUserInfo( )	*	*	Not supported except for IP. See *GlobalCall IP Technology User's Guide* for more information.
gc_SndMsg( )	*		**Limitations:** See *Section 5.9.3. gc_SndMsg( ) Restrictions* for more information.
gc_StartTrace( )		*	Not supported.
gc_StopTrace( )		*	Not supported.
gc_StopTransmitAlarms( )		*	Not supported.

GlobalCall Function	Type		Notes
	Call	Line	
gc_SwapHold( )	*		Not supported.
gc_TransmitAlarms( )		*	Not supported.

6. R4 DTI API for DM3

6.1. DTI Network Interface API Function Restrictions

The Network Interface library (DTI library) contains functions for controlling the digital network interface (E-1 or T-1) to the PSTN (Public Switched Telephone Network). The DTI library includes functions for alarm handling, diagnostics, resource management, SCbus routing and others. See the *Digital Network Interface Software Reference* for detailed information about DTI functions.

R4 for DM3 does not support most DTI functions since the same functionality can be achieved using GlobalCall. However, R4 for DM3 does support the resource management and SCbus routing functions as shown in *Table 13. List of DTI API Functions Fully Supported*. R4 on DM3 now uses the DTI API for Layer 1 alarms. In previous versions of R4 on DM3, the GlobalCall API handled those alarms (see *6.2. Detecting Layer 1* Alarms). All other DTI functions are not supported.

NOTE: If you execute a network interface library function that is not supported on DM3 boards, it produces an EDT_NOTIMP ("not implemented") error.

Table 13. List of DTI API Functions Fully Supported

Function Name	Notes
dt_close( )	Supported. Although **dt_open( )** and **dt_close( )** are supported on DM3 network interface devices, use of the network interface device handle is extremely limited because GlobalCall provides R4 on DM3 application call control rather than the DTI API. In general, use the **gc_OpenEx( )**, **gc_GetNetworkH( )**, and **gc_Close( )** functions instead.
dt_getctinfo( )	Supported.
dt_getxmitslot( )	Supported.

Function Name	Notes
dt_listen( )	Supported.
dt_open( )	Supported. See note for **dt_close( )**.
dt_setevtmsk( )	**Limitations:** Supports enabling layer 1 alarms.
dt_unlisten( )	Supported.
dt_xmitalrm( )	Supported.

6.2. Detecting Layer 1 Alarms

For R4 on DM3, the DTI Network Interface API allows applications to retrieve alarm information that can be used to troubleshoot problems on line devices. The API provides the **dt_setevtmsk( )** function to enable an application to configure which alarms are to be received. The detail on this function is as follows:

- **dt_setevtmsk( )** – enables and disables notification for events that occur on a Digital Network Interface logical board or time slot device. This function allows the application to set a bitmask of transition events. The bitmask determines which transitions cause an event to be generated.

When using this function, check for **DTEV_E1ERRC** or **DTEV_T1ERRC** for E1 or T1 events, respectively.

NOTE: Since **dt_getevtmsk( )** is not supported, the application must keep track of the bitmask of events previously set with the **dt_setevtmsk( )** function.

See the *Digital Network Interface Software Reference* for more information.

7. R4 Fax API for DM3

7.1. Fax Device Information

See *Section 3.3. CT_DEVINFO: Channel/Timeslot Device Information Data Structure* for information on how to identify whether a logical device belongs to DM3 hardware.

In Fax Resource Only Cluster (FROC) configurations (flexible routing), you must issue **fx_listen( )** prior to calling **fx_sendfax( )**, **fx_rcvfax( )** or **fx_originate( )**. Otherwise, these functions will return an error.

7.1.1. Use Fax Handles Only for Fax Commands

R4 on DM3 does not support the use of a voice handle for fax commands; that is, you cannot use the device handle from **dx_open( )** to call fax API functions. You must use **fx_open( )** to open a channel device for fax processing and use that fax device handle.

7.1.2. Opening Board Devices Not Supported

R4 on DM3 does not support the use of **fx_open( )** on a board device; it can only be used on a channel device. As a workaround, use **dx_open( )** on the board device followed by **ATDV_SUBDEVS( )** to determine the number of subdevices (or channels) available. Then perform **fx_open( )** on each subdevice. If **fx_open( )** returns -1, then the subdevice does not support fax.

For more information on **ATDV_SUBDEVS( )**, see the *Standard Runtime Library Programmer's Guide*.

7.1.3. Using Multiple Handles

Applications that create multiple handles for a single fax device should set the parameters consistently on all the handles that perform fax operations, for example, on **fx_sendfax( )** and **fx_rcvfax( )**.

7.2. Fax API Features Not Supported

The following features are not supported:

- DSP-based fax is not supported.
- The voice library **dx_GetRscStatus( )** resource sharing function, which is used for DSP Fax resource sharing, is not supported.
- The *loadfont* ASCII Font Loader Utility is not supported.
- The only font that is supported on DM3 fax boards is the normal font. The compressed font is not supported on DM3 fax boards. Because of this, on DM3 boards the default font for the fax header is different from VFX boards. On DM3 boards, the normal font (ID 0) is used in the fax header, while on earlier-generation boards, the compressed font (ID 3) is the default. Because of these font restrictions on DM3 fax boards (and because of the fixed left and right margins on DM3 fax boards as noted in *Table 15. Fax Data Structure Restrictions*), DM3 fax boards provide fewer characters per line in the fax header. This means that the header may wrap to a second line.
- T.30 subaddress messaging is not supported.

7.3. Fax API Function Restrictions

R4 for DM3 supports all Fax API functions except those noted in *Table 14. List of Fax API Functions Restricted and Not Supported*. This table lists functions that are either **not supported** or are supported with certain restrictions on DM3 devices. The nature of the limitation is noted, and if applicable a reference is given to the location of additional details. See the *Fax Software Reference* for details on each function.

NOTE: If you execute a standard fax function that is not supported by DM3 boards, it produces an EFX_NOTIMP ("not implemented") error. If you execute a supported fax function with a parameter that is not supported on DM3 boards, it produces an EFX_UNSUPPORTED error.

Table 14. List of Fax API Functions Restricted and Not Supported

Function Name	Notes
ATFX_PHDCMD( )	**Limitations:** Operator intervention request is not supported. Therefore, DFS_PRI_EOP, DFS_PRI_MPS, DFS_PRI_EOM are not returned by this function.
ATFX_RTNPAGES( )	Not supported.
ATFX_TERMMSK( )	**Limitations:** Operator intervention request is not supported. Therefore, TM_VOICEREQ is not returned by this function.
ATFX_TFBADTAG( )	Not supported.
ATFX_TFNOTAG( )	Not supported.
ATFX_TFPGBASE( )	Not supported.
ATFX_TRCOUNT( )	**Limitations:** When an error occurs during a send, this value is reset to zero (0).
fx_close( )	**Limitations:** If an **fx_close( )** is sent in the middle of a fax transmission, the fax is aborted (which is equivalent to issuing **fx_stopch( )** and **fx_close( )**). Do not issue an **fx_close( )** during a fax send.

Function Name	Notes
fx_getparm( )	**Limitations:** The following parameters are not supported and return EFX_INVALARG: • FC_FONT0 • FC_FONT3 • FC_RTN • FC_RTP • FC_TXSUBADDR • FC_REMOTESUBADDR • FC_ENDDOC (specifically DFS_REMOTESUBADDR) • FC_HDRATTRIB (specifically DF_HDRUNDERLINE and HDR_BOLD) The following parameters are ignored: • FC_RETRYCNT • FC_TFPGBASE • FC_TFTAGCHECK The following parameter is supported, but the user information is not displayed because of a character number limitation: • FC_HDRUSER
fx_loadfont( )	**Limitations:** Return not implemented.
fx_loadfontfile( )	**Limitations:** Return not implemented.
fx_open( )	**Limitations:** See *Section 7.1.1. Use Fax Handles Only for Fax Commands*.
fx_rcvfax( )	**Limitations:** Operator intervention request is not supported. Therefore, DF_ACCEPT_VRQ and DF_ISSUE_VRQ bit flags are unsupported.
fx_rcvfax2( )	**Limitations:** Operator intervention request is not supported. Therefore, DF_ACCEPT_VRQ and DF_ISSUE_VRQ bit flags are unsupported.
fx_rtvContinue( )	Not supported.

Function Name	Notes
fx_sendfax( )	**Limitations:** 1) Operator intervention request (voice request) is not supported. Therefore, DF_ENABLE_RTV, DF_ACCEPT_VRQ, DF_ISSUE_VRQ, and DF_TXSUBADDR bit flags are unsupported. 2) All DF_IOTT structures are checked before any fax is sent, and a fax is not sent if a bad DF_IOTT structure is included anywhere in the **fx_sendfax( )** function. In such a case, a phase B event is not generated even for any initial good DF_IOTT structures.
fx_setparm( )	**Limitations:** See limitations for **fx_getparm( )**. For information on enhancements, see section *7.5.4. fx_setparm( ).*

7.4. Fax Data Structure Restrictions

The following table lists fax data structure fields that are unsupported or restricted on DM3 devices. The nature of the limitation is noted, and if applicable a reference is given to the location of additional details.

Table 15. Fax Data Structure Restrictions

Data Structure or Field Name	Notes
DF_ASCIIDATA	The DF_ASCIIDATA structure is not used. When an ASCII file is converted to a fax image, the following rules apply. Regardless of the page length you specify, the converted fax image has no maximum size (unlimited length). No pagination is performed by the firmware. Font is fixed at 10 lines per inch (each line is approximately 1/10 inch in height); prints approximately 12 characters per inch; 16 scan lines of MH data; 16 (horizontal) by 16 (vertical) pixels or 80 characters maximum per line. Top Margin is set to 3, Left Margin to 14 and Right Margin to 94.
DF_IOTT	Restrictions apply to the ASCII data type.
io_datatype	If ASCII data type, see notes listed under DF_ASCIIDATA.

7.5. Fax API Enhancements

Enhancements to the R4 Fax API for DM3 boards are described in the following topics.

7.5.1. ATFX_CHTYPE( )

The **ATFX_CHTYPE( )** function returns the following new value for a fax channel device on a DM3 board: DFS_FAXDM3.

You cannot use **ATFX_CHTYPE( )** to determine if a channel device has fax capabilities, because its **dev** parameter requires a fax device handle previously obtained from **fx_open( )**. If **fx_open( )** succeeds, this means that the channel device is already fax-capable. The **ATFX_CHTYPE( )** function only identifies the hardware on which the fax-capable channel device sits. If the hardware is DM3, then this function returns DFS_FAXDM3.

For more information on fax device handling, see sections *2.3.6. Getting, Using, and Closing Device Handles in a DM3 Flexible Routing Configuration* and *7.1. Fax Device Information*.

7.5.2. ATFX_RESLN( )

On DM3 boards, the data provided by the **ATFX_RESLN( )** function is updated each time the fax transfer completes Phase B (rather than Phase D) of the T.30 protocol. By enabling the Phase B event, you can issue **ATFX_RESLN( )** when the TFX_PHASEB event occurs.

7.5.3. ATFX_WIDTH( )

On DM3 boards, the data provided by the **ATFX_WIDTH( )** function is updated each time the fax transfer completes Phase B (rather than Phase D) of the T.30 protocol. By enabling the Phase B event, you can issue **ATFX_WIDTH( )** when the TFX_PHASEB event occurs.

7.5.4. fx_setparm( )

On DM3 boards, the default for the FC_SENDCONT parameter is DFC_AUTO, rather than DFC_EOM on SpringWare boards.

On DM3 boards, the FC_RXCODING parameter supports the following new value, DF_MR, in addition to DF_MH and DF_MMR. Note that DF_MR is not supported on SpringWare boards.

On DM3 boards, the FC_TXCODING parameter supports the following new values, DF_JPEG_GREY and DF_JPEG_COLOR, in addition to DF_MH, DF_MR, DF_MMR, and DF_ECM. DF_JPEG_GREY and DF_JPEG_COLOR are not supported on SpringWare boards. DF_ECM is automatically implied with DF_MMR, DF_JPEG_GREY and DF_JPEG_COLOR settings.

7.6. Color Fax

Color Fax functionality includes enhancements to the R4 Fax API Library to support sending and receiving of JPEG files to and from Group 3 color fax devices.

7.6.1. Color Fax Features

Features of the Basic Color Fax include the following:

- R4 on DM3 Fax API Library
- Support for up to two DM/F240 PCI boards
- Transmission and reception of JPEG encoded color facsimile images to and from color fax devices
- Encoding of color fax images using the JPEG format as specified in ITU Rec. T.85 standard and the ITU Rec. T.4 Annex E standard (ITU Rec. T.85 defines a specific color profile for color fax images. ITU Rec. T.4 Annex E defines the specific JPEG profile for color fax.)

The following baseline JPEG options are supported (as defined in ITU Rec. T.4 Annex E):

- Baseline DCT with Huffman entropy coding
- CIElab color space (L=Luminance [also used for grayscale], A=green/red hue, and B=blue/yellow hue)
- 8 bits/pel/component
- 4:1:1 sub-sampling
- One scan per image file
- Default CIE illuminant D.50
- Default gamut for LAB
- G3FAX APP1 marker: Version=1994 and resolution=200 dpi

7.6.2. Using the Fax API Library for Color Fax

This section includes information about the Fax API library used for color fax.

NOTE: A sample JPEG file is provided with the software. The file name is **colour01.jpg**. This file has the correct JPEG characteristics, so it can be used as a reference image when testing color fax transmissions.

- Two new keywords have been added: DF_JPEG_GREY and DF_JPEG_COLOR. They are intended to be used with a IOTT (iott.io_coding field) that has io_datatype = DF_RAW.
- To enable the JPEG mode for sending or receiving, FC_TXCODING must be set to DF_JPEG_GREY or DF_JPEG_COLOR (this implies automatic DF_MMR and DF_ECM).
- In order to receive in JPEG, the application must receive fax in raw format. Also, ATFX_CODING must be set to DF_JPEG_GREY and DF_JPEG_COLOR, if JPEG files are transmitted.

The R4 FAX API allows you to control many aspects of the T.30 protocol. The only commands you have to configure are the line settings:

- FC_TXCODING
- FC_TXBAUDRATE
- FC_RXBAUDRATE

The basic approach is to extend FC_TXCODING:

```
#define DF_MH           0     // 1-D Group 3 Modified Huffmann encoding
#define DF_MR           1     // 2-D Group 3, T.4 Modified Read encoding
#define DF_MMR          2     // T.6 Modified Modified Read encoding
#define DF_JPEG_GREY    3     // set ECM and T6 + JPEG    (<--- ignore DF_ECM value)
#define DF_JPEG_COLOR   4     // set ECM and T6 + JPEG Full Color
#define DF_ECM          0x8000     // OR with FC_TXCODING value to use ECM
```

This is valid for sending and receiving.

Sending a JPEG Fax

1. Set **fx_setparm( )** FC_TXCODING = DF_JPEG_GREY or DF_JPEG_COLOR

2. Fill the IOTT:

```
Iott->io_type      = IO_DEV | IO_EOT;
Iott->io_fhandle   = dx_fileopen("d:\\F21_200.jpg", _O_RDONLY | _O_BINARY , 0);
Iott->io_bufferp   = NULL;
Iott->io_offset    = 0;
Iott->io_length    = -1;
Iott->io_nextp     = (DF_IOTT *) NULL;
Iott->io_prevp     = (DF_IOTT *) NULL;
Iott->io_width     = DF_WID1728;
Iott->io_resln     = DF_RESHI;
Iott->io_coding    = DF_JPEG_COLOR  (for JPEG: DF_JPEG_GREY or DF_JPEG_COLOR)
Iott->io_phdcont   = DFC_AUTO;
Iott->io_datatype  = DF_RAW #(mandatory for a JPEG)
```

NOTE: A file can also be sent from memory (IO_MEM instead of IO_DEV)

Sending a JPEG-only File

```
          FC_TXCODING set to DF_JPEG_COLOR
JPEG Color
         (MPS)
JPEG Color
         (MPS)
JPEG Color
         (EOP)
```

Additional information:

- If the IOTT contains a JPEG file, and the TX_CODING is not correct (for example, JPEG Color file and TX_CODING is JPEG_GREY), an error is reported.
- If IOTT doesn't contain a JPEG entry, turn off JPEG, even if TX_CODING expects JPEG.
- When there is a JPEG file to send, headers are turned off for ALL the pages (also MH/MR/MMR/ASCII file).
- If you are sending a JPEG file, check that the DIS of the receiving side supports JPEG
- If the user forces a MPS as PhaseD command, or if there is a different JPEG value (Disable/GREY/Color), then an EOM is forced.

Receiving a JPEG File

NOTE: The only mode supported is RAW.

- FC_TXCODING = DF_JPEG_GREY or DF_JPEG_COLOR
- If the application sets FC_TXCODING to DF_JPEG_GREY or DF_JPEG_COLOR, and if the **fx_rcvfax( )** is not issued with the DF_RAW, the function will return an error.
- **ATFX_CODING( )** reports DF_JPEG_GREY or DF_JPEG_COLOR.
- As was done previously, FC_RXCODING specifies the receive file, except if **ATFX_CODING( )** reports DF_JPEG_GREY or DF_JPEG_COLOR.
- Backward compatibility is preserved with MH/MR/MMR reception.

Receiving a JPEG Fax

```
     FIS_PrmJPEG_JP_JPEG | FIS_PrmJPEG_JP_FULL_COLOR
DCS = QFC3_MsgReportCapsEvt_JPEG_JP_DISABLED
     ATFX_CODING = DF_MH, DF_MR or DF_MMR
     -> receive raw file
EOM
DCS = QFC3_MsgReportCapsEvt_JPEG_JP_JPEG
     ATFX_CODING = DF_JPEG_GREY
     -> receive raw file
EOM
DCS = QFC3_MsgReportCapsEvt_JPEG_JP_FULL_COLOR
     ATFX_CODING = DF_JPEG_GREY
     -> receive raw file
EOP
```

7.7. Support for fx_originate( )

The **fx_originate( )** function is supported on DM3 boards only. Detailed information on this function follows.

fx_originate( ) allows the DCS on hold feature

Name:	int fx_originate(int dev, int mode)	
Inputs:	int dev	• fax channel device handle
	int mode	• asynchronous/synchronous
Returns:	0 on success	
	-1 on failure	
Includes:	srllib.h	
	dxxxlib.h	
	faxlib.h	
Category:	originate fax	
Mode:	synchronous/asynchronous	

■ Description

The **fx_originate( )** function allows the DCS on hold feature to be used. The function results in a TFX_ORIGINATE or TFX_FAXERROR event, and the command can be stopped by **fx_stopch( )**.

Upon receipt of a TFX_ORIGINATE event, an **fx_sendfax( )** should follow, and the application can call **ATFX_BSTAT( )** and access FC_REMOTEID, **fx_getDIS( )**, **fx_getNSF( )**, **ATFX_SPEED( )**, **ATFX_CODING( )**, **ATFX_ECM( )**. All functions are filled with DF_DIS (Digital Identification Signal) values to avoid having to decode the DIS frame. Values are updated later, during the **fx_sendfax( )**, with DF_DCS (Digital Command Signal) values. When the **fx_sendfax( )** is issued, the file/image format can be specified, including JPEG. See *7.6.2. Using the Fax API Library for Color Fax* for additional information on sending JPEGs.

Parameter	**Description**
dev:	Specifies the device handle returned for the fax channel when the channel was opened.
mode:	The operation mode specifies whether the function should run asynchronously or synchronously.

Parameter	Description	
	EV_ASYNC	• Run asynchronously. Returns -1 to indicate failure to initiate. Returns 0 to indicate successful initiation and generates a TFX_ORIGINATE message once the DIS is detected, or TFX_FAXERROR if it is not detected.
	EV_SYNC	• Run synchronously. Returns 0 on success and -1 on failure.

■ Cautions

- This function is supported on DM3 boards only.
- You must call the **fx_initstat(DF_TX)** function at least once prior to calling **fx_originate( )**.
- In Fax Resource Only Cluster (FROC) configurations (Fax24/Fax30 = Fax Cluster that has a timeslot), you must issue **fx_listen( )** prior to calling **fx_sendfax( )**, **fx_rcvfax( )** or **fx_originate( )**. Otherwise, these functions will return an error.
- If the **fx_sendfax( )** is not received within the 20s (limited by the T1 timer + security margin), the fax session will be aborted and result in TFX_FAXERROR.

■ Example

```
#include <stdio.h>
#include <errno.h>
#include <windows.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <faxlib.h>
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#include <conio.h>

int        dev; /* Fax channel device handle. */
DF_IOTT   iott;

int catchall( );
```

```
int main(int argc, char* argv[])
{
    int CanRun = 1;
    int mode = SR_STASYNC;

    /* Set SRL to turn off creation of internal thread */
    if( sr_setparm( SRL_DEVICE, SR_MODELTYPE, &mode ) == -1 ){
        printf( "Error: cannot set srl mode\n" );
        exit( 1 );
    }
    /*
    * Open the channel using fx_open( ) to obtain the FAX
    * channel device handle in dev.
    */
    if ((dev = fx_open("dxxxB1C1", NULL)) == -1) {
        /* Error opening device. */
        printf("Error opening channel, errno = %d\n", errno);
        exit(1);
    }

    /*
    * If this is not a Dm3 Fax channel (return type DFS_FAXDM3)
    * warn the user.
    */
    if (ATFX_CHTYPE(dev) != DFS_FAXDM3) {
        printf("Function fx_originate is not supported\n");
    } else {

        /* Open the file as read-only. */
        iott.io_fhandle = dx_fileopen("sample.tif" ,O_RDONLY|O_BINARY,0);
        /*
        * Set up the DF_IOTT structure as the default and then
        * change the necessary fields.
        */
        fx_setiott(&iott, iott.io_fhandle, DF_TIFF, DFC_AUTO);
        iott.io_type |= IO_EOT;
        printf("Press SPACE to show fx_originate usage, or ESC to exit\n");
        while (CanRun) {
            if (sr_waitevt(100) != -1) {
                catchall();
            }
            if (kbhit()) {
                switch(getch()) {
                case 27: /* Esc */
                    CanRun = 0;
                    break;
                case ' ': /* Space */
                    /* Set initial state of FAX channel to TRANSMITTER. */
                    if (fx_initstat(dev, DF_TX) == -1) {
                        printf("Error - %s (error code %d)\n", ATDV_ERRMSGP(dev),
                            ATDV_LASTERR(dev));
                        if (ATDV_LASTERR(dev) == EDX_SYSTEM) {
                            printf("errno = %d\n", errno);
                        }
                    } else {
                        if (iott.io_fhandle != -1) {
                            printf("Issue our fx_originate\n");
                            fx_originate(dev, EV_ASYNC);
                        }
                    }
                    break;
```

```
                }
            }
        }
        dx_fileclose(iott.io_fhandle);
    }
    /*
    * close the channel using fx_close( )
    */
    fx_close(dev);
    return 0;
}


/* Event handler. */
/*
* This routine gets called when sr_waitevt( ) receives any event.
* Maintain a state machine for every channel and issue the
* appropriate function depending on the next action to be
* performed on the channel.
*/
int catchall( )
{
    int dev;
    dev = sr_getevtdev( );

    /* Determine the event. */
    switch(sr_getevttype( )) {
    case TFX_ORIGINATE:
        {
            long BStat = ATFX_BSTAT(dev);
            char DisBuf[100]={"N/A"};
            char RemoteId[100]={"N/A"};
            if (BStat & DFS_REMOTEID) {
                if (fx_getparm(dev, FC_REMOTEID, RemoteId)!=0) {
                }
            }
            if (BStat & DFS_DIS) {
                DF_DIS DfDis;
                if (fx_getDIS(dev, &DfDis)==-1) {
                }
                else {
                    /* should translate DIS to a string */
                    strcpy(DisBuf, "Present");
                }
            }
            if (BStat & DFS_NSF) {
            /*...    */
            }
            printf("Receiver is capable of: speed %ld, resln %ld,"
                "width %ld, Ecm %ld\n", ATFX_SPEED(dev),
                ATFX_RESLN(dev), ATFX_WIDTH(dev),
                ATFX_ECM(dev));
            printf("Information: Csid='%s' and Dis is %s\n",
                RemoteId, DisBuf);
            printf("Issue our fx_sendfax\n");
            /*
            * Depending the capabilities, Parameter can be adjusted
            * and the user is capable of pointing to the appropriate
            * iott structure (e.g., Raw Color Fax or simple tiff image)
            */
            fx_sendfax(dev, &iott,EV_ASYNC|DF_PHASEB|DF_PHASED);
        }
```

```
        break;
    case TFX_FAXSEND:
        /* The document has been successfully sent. */
        printf("Sent %ld pages at speed %ld, resln %ld,"
            "width %ld\n", ATFX_PGXFER(dev), ATFX_SPEED(dev),
            ATFX_RESLN(dev), ATFX_WIDTH(dev));
        /* Fax session completed. */
        printf("Press ESC to exit\n");
        break;
    case TFX_PHASEB:
        printf("Phase B event\n");
        /* extract usual information from here */
        break;
    case TFX_PHASED:
        printf("Phase D event\n");
        /* extract usual information from here */
        break;
    case TFX_FAXERROR:
        /* Error during the fax session. */
/*        print_err(dev); */
        printf("Phase E status %ld\n", ATFX_ESTAT(dev));
        /* Application specific error handling. */
            break;
    default:
            break;
    } /* End of switch. */
    return(0);
}
```

■ Errors

TFX_ORIGINATE	Successful fax origination
TFX_FAXERROR	Error in processing
EFX_UNSUPPORTED	Unsupported function. This error returned if this function is attempted on a non-DM3 board

Refer to the error type tables found in the *Fax Software Reference*. Error defines can be found in *faxlib.h*.

■ See Also

- **fx_getDIS( )**
- **fx_getNSF( )**
- **fx_stopch( )**
- **ATFX_SPEED( )**
- **ATFX_CODING( )**

- **ATFX_ECM( )**

7.8. Support for fx_getctinfo( )

The **fx_getctinfo( )** function, previously undocumented, is supported on DM3 boards and SpringWare boards. Information on this function follows.

fx_getctinfo( ) returns information about a fax channel device handle

Name:	int fx_getctinfo(chdev, ct_devinfop)	
Inputs:	int chdev	• fax channel device handle
	CT_DEVINFO *ct_devinfop	• pointer to device information structure
Returns:	0 on success -1 on failure	
Includes:	srllib.h dxxxlib.h faxlib.h	
Category:	SCbus routing	
Mode:	synchronous	

■ Description

The **fx_getctinfo( )** function returns information about a fax channel device handle.

Parameter	Description
chdev:	Specifies the valid fax channel device handle obtained when the channel was opened using **fx_open( )**.
ct_devinfop:	Specifies a pointer to the data structure CT_DEVINFO.

On return from the function, the CT_DEVINFO structure contains the relevant information and is declared as follows:

```
typedef struct {
     unsigned long ct_prodid;
     unsigned char ct_devfamily;
     unsigned char ct_devmode;
     unsigned char ct_nettype;
     unsigned char ct_busmode;
     unsigned char ct_busencoding;
     unsigned char ct_rfu[7];
} CT_DEVINFO;
```

Valid values for each member of the CT_DEVINFO structure are defined in *dxxxlib.h.* For more information on CT_DEVINFO specific to DM3 boards, see section *3.3. CT_DEVINFO: Channel/Timeslot Device Information Data Structure*.

■ Cautions

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

■ Example

```
#include <windows.h>     /* include in Windows applications only; exclude in Linux */
#include <srllib.h>
#include <dxxxlib.h>
#include <faxlib.h>
#include <errno.h>      /* include in Linux applications only; exclude in Windows */

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

   /* Open board 1 channel 1 devices */
   if ((chdev = fx_open("dxxxB1C1", 0)) == -1) {
     printf("Cannot open channel\n");
     /* Perform system error processing */
     exit(1);
   }

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

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

```
        ct_devinfo.ct_busmode, ct_devinfo.ct_busencoding);
}
```

■ Errors

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

Equate	Returned When
EFX_BADPARM	Invalid value for fax parameter (on DM3 boards only)
EDX_BADPARM	Invalid value for parameter (on SpringWare boards)
EDX_SH_BADEXTTS	SCbus time slot is not supported at current clock rate
EDX_SH_BADINDX	Invalid Switch Handler library index number
EDX_SH_BADTYPE	Invalid channel type (voice, analog, etc.)
EDX_SH_CMDBLOCK	Blocking command is in progress
EDX_SH_LIBBSY	Switch Handler library is busy
EDX_SH_LIBNOTINIT	Switch Handler library is uninitialized
EDX_SH_MISSING	Switch Handler is not present
EDX_SH_NOCLK	Switch Handler clock fallback failed
EDX_SYSTEM	Operating system error

■ See Also

- **dt_getctinfo( )**
- **dx_getctinfo( )**

8. R4 DCB API for DM3

8.1. DCB Audio Conferencing API Features Not Supported

The following DCB features are not supported:

- Monitoring of a single conference by multiple participants through the **dcb_monconf( )** and **dcb_unmonconf( )** functions. However, an alternative is to add a receive-only conferee to the conference.
- Volume control for any conferee.

8.2. DCB Audio Conferencing API Function Restrictions

R4 for DM3 supports all DCB API functions except those noted in *Table 16. List of DCB Audio Conferencing API Functions Restricted and Not Supported* . This table lists functions that are either **not supported** or are supported with certain restrictions on DM3 devices. The nature of the limitation is noted, and if applicable a reference is given to the location of additional details. See the *Dialogic Audio Conferencing Software Reference* for details on each function.

NOTE: If you execute a standard DCB function that is not supported by DM3 boards, it produces an EDT_INVMSG (“invalid message”) error. If you execute a supported DCB function with a parameter that is not supported on DM3 boards, it produces an EDT_PARAMERR (“invalid parameter”) error.

Table 16. List of DCB Audio Conferencing API Functions Restricted and Not Supported

Function Name	Notes
dcb_addtoconf( )	**Limitations:** See *Section 8.2.1. DCB Parameters Restricted and Not Supported*. **Enhancements for DM3 Boards:** See *Section 8.5.1. New DCB Conferencing Party Attribute Parameters*.
dcb_estconf( )	**Limitations:** See *Section 8.2.1. DCB Parameters Restricted and Not Supported*. **Enhancements for DM3 Boards:** See *Section 8.5.1. New DCB Conferencing Party Attribute Parameters*.
dcb_GetAtiBits( )	Not supported on DM3 boards. However, this functionality is provided by the **dcb_GetAtiBitsEx( )** function, which is supported on DM3 boards only. See *Section 8.5.2. New DCB Functions*.
dcb_getbrdparm( )	**Limitations:** Volume control (MSG_VOLDIG) is not supported. Tone Clamping (MSG_TONECLAMP) is not supported on DI products. **Enhancements for DM3 Boards:** See *Section 8.3. Enhancements for dcb_getbrdparm( )*.
dcb_getcde( )	**Limitations:** See *Section 8.2.1. DCB Parameters Restricted and Not Supported*. **Enhancements for DM3 Boards:** See *Section 8.5.1. New DCB Conferencing Party Attribute Parameters*.
dcb_getdigitmsk( )	Not supported on DI products.
dcb_monconf( )	Not supported.

Function Name	Notes
dcb_setbrdparm( )	**Limitations:** Volume control (MSG_VOLDIG) is not supported. Tone clamping (MSG_TONECLAMP) is not supported on DI products. **Enhancements for DM3 Boards:** See *Section 8.4. Enhancements for dcb_setbrdparm( )*.
dcb_setcde( )	**Limitations:** See *Section 8.2.1. DCB Parameters Restricted and Not Supported*. **Enhancements for DM3 Boards:** See *Section 8.5.1. New DCB Conferencing Party Attribute Parameters*.
dcb_setdigitmsk( )	Not supported on DI products.
dcb_unmonconf( )	Not supported.

8.2.1. DCB Parameters Restricted and Not Supported

The following DCB parameters are either not supported or not applicable to the **dcb_addtoconf( )**, **dcb_estconf( )**, **dcb_getcde( )**, and **dcb_setcde( )** functions.

Not Applicable to DM3 Boards:

- MSPA_DIG: Digital front end (not applicable to, and is ignored for, DM3 boards).

Not Supported:

- MSPA_NOAGC: Disable AGC.
- For the **dcb_estconf( )** function, conference attribute (**conf_attr**) parameter MSCA_NN is not supported.

8.3. Enhancements for dcb_getbrdparm( )

8.3.1. Tone Clamping

The **MSG_TONECLAMP** parameter is supported and can be used to retrieve the value set by **dcb_setbrdparm( )** for tone clamping (TONECLAMP_ON or TONECLAMP_OFF).

NOTE: The DI products do not support tone clamping.

8.3.2. Retrieving the Interval for Active Talker Notification Events

Active Talker indications determine which conferees in any given conference are currently talking. Active Talkers are those conferees providing “non-silence” energy.

The MSG_ACTTALKERNOTIFYINTERVAL parameter is supported and can be used to retrieve the interval value (in 100 ms units) for Active Talker notification events set by **dcb_setbrdparm( )**.

8.4. Enhancements for dcb_setbrdparm( )

8.4.1. Tone Clamping

The **MSG_TONECLAMP** parameter is supported and can be set to TONECLAMP_ON or TONECLAMP_OFF (default) by the **dcb_setbrdparm( )** function. This parameter enables tone clamping to reduce the amount of DTMF tones heard in a conference.

NOTE: The DI products do not support tone clamping.

8.4.2. Changing the Interval for Active Talker Notification Events

Active Talker indications determine which conferees in any given conference are currently talking. Active Talkers are those conferees providing “non-silence” energy.

The **MSG_ACTTALKERNOTIFYINTERVAL** parameter can now be used to change the default firmware interval for Active Talker notification events. If necessary, an application can change the interval using this run-time board level parameter. The value must be passed in 100 ms units.

8.5. DCB Audio Conferencing API Enhancements for DM3 Boards

The following DCB API enhancements are supported on DM3 boards only:

- New parameters for the **dcb_addtoconf( )**, **dcb_estconf( )**, **dcb_getcde( )**, and **dcb_setcde( )** functions
- Parameters for the **dcb_addtoconf( )**, **dcb_estconf( )**, **dcb_getcde( )**, and **dcb_setcde( )** functions
- New functions:
    - **dcb_DeleteAllConferences( )**
    - **dcb_GetAtiBitsEx( )**

8.5.1. New DCB Conferencing Party Attribute Parameters

The following party attribute parameters are **supported** only on DM3 boards for the **dcb_addtoconf( )**, **dcb_estconf( )**, **dcb_getcde( )**, and **dcb_setcde( )** functions:

- MSPA_ECHOXCLEN: Enable echo cancellation (default: disabled). The echo cancellation feature supplies 128 tap (16 msec) echo cancellation with the DCB interface. This feature is disabled by default for compatibility with earlier-generation operation. Note that this is a DM3-only feature.
- MSPA_BROADCASTEN: Enable broadcast to all conferees. This parameter sets one party to talk and all others are muted.
- MSPA_MODENULL: Null party. No transmit or receive timeslots (a null party is often used as a placeholder for establishing a conference for a fixed number of parties).

- MSPA_MODERECVONLY: Receive-only party. Conferee participates in conference in receive-only mode (equivalent to MSPA_RO on earlier-generation boards).
- MSPA_MODEXMITONLY: Transmit-only party. Conferee participates in conference in transmit-only mode.
- MSPA_MODEFULLDUPLX: Full Duplex party (conferee may transmit and receive). No special attributes for conferee. Conferee hears everyone except the coach (MSPA_NULL is used for this purpose on earlier-generation boards and may also be used for DM3 boards).

8.5.2. New DCB Functions

The **dcb_DeleteAllConferences( )** and **dcb_GetAtiBitsEx( )** functions are supported on DM3 boards only. Detailed information on these functions follows.

dcb_DeleteAllConferences( ) *deletes all established conferences*

Name:	int dcb_DeleteAllConferences(devh, mode, rfu)	
Inputs:	int devh	• valid Dialogic DCB DSP device handle
	unsigned short mode	• asynchronous/synchronous
	void* rfu	• reserved for future use
Returns:	0 on success -1 on failure	
Includes:	srllib.h dtilib.h msilib.h dcblib.h	
Category:	Conference Management	
Mode:	synchronous/asynchronous	

■ Description

The **dcb_DeleteAllConferences( )** function deletes all established conferences. This function is provided for recovery purposes. A typical use would be if an application had to be restarted due to an abnormal termination: rather than downloading firmware to re-initialize the boards, this function could be called to delete all conferences that might be left active in firmware.

Parameter	Description
devh:	The valid DCB DSP device handle returned by a call to **dcb_open( )**.
mode:	The operation mode specifies whether the function should run asynchronously or synchronously. EV_ASYNC • Run asynchronously. Returns -1 to indicate failure to initiate. Returns 0 to indicate successful initiation and then generates either the DCBEV_DELALLCONF termination event to indicate successful

Parameter	Description	
		completion (all conferences deleted), or the DCBEV_ERREVT in case of error. (Use the SRL Event Management functions to handle the termination events.)
	EV_SYNC	• Run synchronously. Returns 0 on success and -1 on failure.

NOTE: Calling this function frees all resources in use by all the conferences on the specified DCB DSP device.

Cautions

- This function fails when the specified device handle is invalid.
- This function is supported only on DM3 boards.
- This function is intended for application program recovery purposes. (The **dcb_delconf( )** function is intended for standard conference management purposes and should be used with the appropriate **xx_unlisten( )** routing functions.)
- In a multi-threaded or multi-process environment, DCB functions should not be invoked at the same time **dcb_DeleteAllConferences( )** is called.
- This function will not perform **xx_unlisten( )** operations upon devices listening to conference time slots when this function is executed. It is the application's responsibility to perform unrouting using **xx_unlisten( )** function(s).

Example 1: Asynchronous

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

#include "srllib.h"
#include "dtilib.h"
#include "msilib.h"
#include "dcblib.h"

int dspdevh;

long EventHandler (unsigned long temp)
{
```

```
  int dev=sr_getevtdev();
  long event=sr_getevttype();
  int nSize = sr_getevtlen();
  int *pData = (int*) sr_getevtdatap();

  switch(event)  {
    case DCBEV_DELALLCONF :
      printf("EventHandler->DCBEV_DELALLCONF DeviceHandle = %d Device = %s Size = %d
Data = 0x%x\n",
          dev,ATDV_NAMEP(dev),nSize,*pData);
      break;
    case DCBEV_ERREVT :
      printf("EventHandler->DCBEV_ERREVT DeviceHandle = %d Device = %s Size = %d Data =
0x%x\n",
          dev,ATDV_NAMEP(dev),nSize,*pData);
      break;
    default :
      printf("EventHandler->Unknown event received...Event = 0x%x DeviceHandle = %d
Device = %s Size = %d\n",
            event,dev,ATDV_NAMEP(dev),nSize);
      break;
  }  //switch event ends
  return 0;
}  // EventHandler ends

void main(void)
{
  char DeviceName[16];
  void *RFU=0;
  int srlmodel = SR_STASYNC;

  // Set SR_MODELTYPE to SR_STASYNC to disable creation of internal thread to
  // monitor events for callback handler.  Instead, the application will
  // use sr_waitevt() to poll for events.
  if (sr_setparm(SRL_DEVICE, SR_MODELTYPE, (void *)&srlmodel) == -1) {
    /* process error  */
  }

  sprintf(DeviceName,"dcbB1D1");
  printf("Trying to open device = %s\n",DeviceName);
  if ( (dspdevh = dcb_open(DeviceName, 0)) == -1) {
    printf("Cannot open device %s. errno = %d\n", DeviceName,errno);
    exit(1);
  }
  else {
    printf("Open successfull for device %s... dspdevh=0x%x\n",DeviceName,dspdevh);
  }

  /* Set up handler function to handle DeleteAllConferences completion  */
  if (sr_enbhdlr(EV_ANYDEV, EV_ANYEVT, &EventHandler) < 0)  {
    /* process error  */
  }

  /* Establish Conferences and Continue Processing  */

  /* Delete all Active Conferences  */
  if (dcb_DeleteAllConferences(dspdevh, EV_ASYNC, 0) == -1)
  {
    printf("dcb_DeleteAllConferences failed for %s.  Error message = %s",
            ATDV_NAMEP(dspdevh), ATDV_ERRMSGP(dspdevh));
    /* process error */
  }
```

```
  else
  {
    printf("dcb_DeleteAllConferences issued for %s... \n",
            ATDV_NAMEP(dspdevh));
  }

  /* Use sr_waitevt to wait for the completion of dcb_DeleteAllConferences.
    On receiving the completion event, control is transferred to the handler
    function previously established using sr_enbhdlr
  */

  /* Done processing - Close device */
  if ( dcb_close(dspdevh) == -1) {
    printf("Cannot close device %s. errno = %d\n", DeviceName,errno);
    exit(1);
  }
  else
    printf("dcb_close successfull for device %s... dspdevh=0x%x\n",DeviceName,dspdevh);
}  // main() ends
```


Example 2: Synchronous

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

#include "srllib.h"
#include "dtilib.h"
#include "msilib.h"
#include "dcblib.h"
#include "errno.h"

int dspdevh=0;

int  DeleteAllConferences(void);

void main(void)
{
  char DeviceName[16];

  sprintf(DeviceName,"dcbB1D1");
  printf("Trying to open device = %s\n",DeviceName);
  if ( (dspdevh = dcb_open(DeviceName, 0)) == -1) {
    printf("Cannot open device %s. errno = %d\n", DeviceName,errno);
    exit(1);
  }
  else  {
    printf("Open successfull for %s... dspdevh=0x%x\n",DeviceName,dspdevh);
  }

  /* Establish Conferences and Continue Processing  */

  /* Delete all Active Conferences  */
  DeleteAllConferences();

  /* Done processing - Close device */
  if ( dcb_close(dspdevh) == -1) {
    printf("Cannot close device %s. errno = %d\n", DeviceName,errno);
    exit(1);
  }
  else  {
```

```
    printf("dcb_close successfull ... dspdevh=0x%x\n",dspdevh);
  }
}  // main() ends


int DeleteAllConferences()
{
  void *RFU=0;
  if (dcb_DeleteAllConferences(dspdevh, EV_SYNC, RFU) == -1)
  {
    printf("dcb_DeleteAllConferences failed for %s.  Error message = %s",
            ATDV_NAMEP(dspdevh), ATDV_ERRMSGP(dspdevh));
    return(-1);
  }
  else
  {
    printf("dcb_DeleteAllConferences successfull for %s... \n",
            ATDV_NAMEP(dspdevh));
  }
  return(0);
}  //DeleteAllConferences ends
```


■ Errors

If the function does not complete successfully, it will return -1 to indicate an error. Use the Standard Attribute functions **ATDV_LASTERR( )** to obtain the applicable error value, or **ATDV_ERRMSGP( )** to obtain a more descriptive error message.

Refer to the error type tables found in *Chapter 2* of the *Dialogic Audio Conferencing Software Reference*. Error defines can be found in *dtilib.h*, *msilib.h* or *dcblib.h*.

■ See Also

- **dcb_addtoconf( )**
- **dcb_delconf( )**
- **dcb_estconf( )**
- **dcb_remfromconf( )**

dcb_GetAtiBitsEx( ) *returns the active talker indicators*

Name:	int dcb_GetAtiBitsEx(devh, numpty, ActiveTalkerInd, rfu)	
Inputs:	int devh	• valid Dialogic DCB DSP device handle
	int *numpty	• pointer to number of active talkers
	DCB_CT *ActiveTalkerInd	• pointer to array of active talker indicators
	void* rfu	• reserved for future use
Returns:	0 if success -1 if failure	
Includes:	srllib.h dtilib.h msilib.h dcblib.h	
Category:	Auxiliary	
Mode:	synchronous	

■ Description

The **dcb_GetAtiBitsEx( )** function returns the active talker indicators at the time the function is called. The current number of active talkers is returned in **numpty**, with specific information on each active talker returned in **ActiveTalkerInd**.

Parameter	Description
devh:	The valid DCB DSP device handle returned by a call to **dcb_open( )**.
numpty:	Pointer to number of active talkers.
ActiveTalkerInd:	Pointer to array of DCB_CT structure where active talker indicators are returned.

The format of the DCB_CT structure is as follows:

```
typedef struct {
        int confid;        /* conference ID number */
        int chan_num;      /* time slot number */
```

Parameter	Description

	`int chan_sel; /* time slot selector */`
	`} DCB_CT;`
	
	The **confid** specifies the conference ID number to which the active talker belongs. The **chan_num** specifies the CT bus transmit timeslot number of the active talker. The **chan_sel** provides additional information on the channel; at present, for DM3 boards, **chan_sel** always returns MSPN_TS (CT Bus timeslot).

NOTES: 1. The developer must allocate and deallocate an array of DCB_CT structures large enough to store information for the maximum possible number of active talkers.

2. The snapshot of information provided by **dcb_GetAtiBitsEx( )** is accurate for a split second. This information may not be accurate by the time the application processes it.

■ Cautions

- This function fails when the device handle is invalid.
- This function is supported only on DM3 boards.

■ Example

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

#include "srllib.h"
#include "dtilib.h"
#include "msilib.h"
#include "dcblib.h"
#include "errno.h"

const int MAX_ACTIVETALKERBITS = 120;

void main(void)
{
  char DeviceName[16];
  char BoardName[16];
  int dspdevh=-1,BoardDevHandle=-1;

  sprintf(BoardName,"dcbB1");
  printf("Trying to open device = %s\n",BoardName);
  if ( (BoardDevHandle = dcb_open(BoardName, 0)) == -1) {
    printf("Cannot open device %s. errno = %d\n", BoardName,errno);
```

```
    getchar();
    exit(1);
  }
  else {
    printf("Open successfull for %s ...
BoardDevHandle=0x%x\n",BoardName,BoardDevHandle);
  }

  /* Set Active Talker ON  */
  int nStatus=ACTID_ON;
  if(dcb_setbrdparm(BoardDevHandle,MSG_ACTID,(void*)&nStatus)==-1)  {
    printf("dcb_setbrdparm->MSG_ACTID->Error Setting Board Parm for %s : ERROR = %s\n",
          ATDV_NAMEP(BoardDevHandle),ATDV_ERRMSGP(BoardDevHandle));
    /* process error */
  }
  else  {
    printf("SetBoardParm->MSG_ACTID->Success Setting Board Parm for %s\n",
          ATDV_NAMEP(BoardDevHandle));
  }

  if ( dcb_close(BoardDevHandle) == -1) {
    printf("Cannot close device %s. errno = %d\n", ATDV_NAMEP(BoardDevHandle),errno);
    /* process error */
    exit(1);
  }
  else {
    printf("dcb_close successfull for %s... BoardDevHandle=0x%x\n",
          ATDV_NAMEP(BoardDevHandle),BoardDevHandle);
  }

  sprintf(DeviceName,"dcbB1D1");
  printf("Trying to open device = %s\n",DeviceName);
  if ( (dspdevh = dcb_open(DeviceName, 0)) == -1) {
    printf("Cannot open device %s. errno = %d\n", DeviceName,errno);
    /* process error  */
    exit(1);
  }
  else  {
    printf("Open successfull for %s... dspdevh=0x%x\n",DeviceName,dspdevh);
  }

  /* Establish Conferences and Continue Processing  */

  /*  GetAtiBitsEx  */
  int nCount,i=0;
  DCB_CT ActiveTalkerIndicators[MAX_ACTIVETALKERBITS];
  void * RFU=0;
  if(dcb_getatibitsEx(dspdevh, &nCount, ActiveTalkerIndicators, RFU)==-1)
  {
    printf("GetAtiBits->dcb_getatibitsEx failed on %s Error = %s\n",
          ATDV_NAMEP(dspdevh),ATDV_ERRMSGP(dspdevh));
    /* process error  */
  }
  else
  {
    printf("GetAtiBits->dcb_getatibitsEx Successful on %s Count = %d\n",
          ATDV_NAMEP(dspdevh),nCount);
    for(i=0;i<nCount;i++)
    {
      printf("i = %d ConferenceID = 0x%x ChanNum = %d ChanSel = 0x%x\n",
            i,ActiveTalkerIndicators[i].confid,
            ActiveTalkerIndicators[i].chan_num,
```

```
            ActiveTalkerIndicators[i].chan_sel);
    }
  }

  /* Done processing - Close device */
  if ( dcb_close(dspdevh) == -1) {
    printf("Cannot close device %s. errno = %d\n", ATDV_NAMEP(dspdevh),errno);
    /* process error  */
    exit(1);
  }
  else
    printf("dcb_close successfull for %s... dspdevh=0x%x\n",
          ATDV_NAMEP(dspdevh),dspdevh);
}  // main() ends
```


■ Errors

If the function does not complete successfully, it will return -1 to indicate an error. Use the Standard Attribute functions **ATDV_LASTERR( )** to obtain the applicable error value, or **ATDV_ERRMSGP( )** to obtain a more descriptive error message.

Refer to the error type tables found in *Chapter 2* of this guide. Error defines can be found in *dtilib.h, msilib.h* or *dcblib.h*.

■ See Also

- **dcb_getatibits( )**
- **dcb_evtstatus( )**
- **dcb_gettalkers( )**

9. R4 MSI API for DM3

9.1. MSI API Function Restrictions

The following table lists R4 MSI API functions that are not supported or have limitations that restrict their use. For information about station interfaces and using the R4 MSI API, refer to the *MSI/SC Software Reference*. For information on the specific boards that support these functions, see the Release Guide that came with your System Release.

Table 17. R4 MSI API Functions Restricted and Not Supported

Function Name	Description
ATMS_DNLDVER( )	Not supported.
ATMS_STATINFO( )	**Limitation**: This function is only supported on a Station Interface Box (SIB) configured for a density of 120 channels in an HDSI system.
ms_chgxtder( )	Not supported.
ms_delxtdcon( )	Not supported.
ms_dsprescount( )	Not supported.
ms_estxtdcon( )	Not supported.
ms_genringex( )	**Limitation**: This release supports six of the eight MSI predefined ring cadences using this function. The following two distinctive rings are **not** supported**:** • MS_RNGA_SPLASH3-0.5On-0.25Off-0.5On-0.25Off-0.5On-4Off • MS_RNGA_SPLASH4-0.25On-0.25Off-0.25On-0.25Off-0.25On-0.25Off-0.25On-4.25Off
ms_genziptone( )	Not supported. On DM3 boards, this function fails with error: "Can't map or allocate memory in driver."
ms_getbrdparm( )	**Limitation**: Only MSG_RING is supported.
ms_monconf( )	Not supported.

Function Name	Description
ms_setbrdparm( )	**Limitation**: Only MSG_DISTINCTRNG is supported.
ms_unmonconf( )	Not supported.

9.1.1. MSI API Parameters Not Supported

The following R4 MSI API parameters are not supported.

Table 18. R4 MSI API Parameters Not Supported

Parameter ID	Description
MSCB_ND	Defines the notify-on-add tone generated to notify conference parties that a party has joined or left the conference.
MSG_DBOFFTM	Defines the minimum length of time before an off-hook transition is detected.
MSG_DBONTM	Defines the minimum length of time before an on-hook transition is detected.
MSG_MAXFLASH	Defines a maximum length of time for a station to be in an on-hook state before a hook flash signal is detected.
MSG_MINFLASH	Defines a minimum length of time for a station to be in an on-hook state before a hook flash signal is detected.
MSG_PDRNGCAD	Used to select one of the predefined ring cadence patterns on the MSI board.
MSG_RNGCAD	Used to get the ring cadence pattern.
MSG_UDRNGCAD	Used to set User Defined Ring Cadence.

9.2. New Functions for CallerID

Two new station functions were added to the existing MSI R4 API to support the CallerID feature on DM3 boards.

- **ms_genringCallerID( )**

- **ms_SetMsgWaitInd( )**

NOTE: These functions are only supported on DM3 boards and do not apply to earlier-generation products.

ms_genringCallerID( ) *transmits Caller ID information*

Name:	int ms_genringCallerID(devh, len, mode, cadid, OrigAddr, rfu)	
Inputs:	int devh	• station device handle
	unsigned short len	• length in cycles for ring
	unsigned short mode	• asynchronous/synchronous
	char* OrigAddr	• call origination information
	void* rfu	• reserved for future use
Returns:	0 if success for asynchronous	
	>0 if success for synchronous	
	-1 if failure	
Includes:	dtilib.h	
	srllib.h	
	msilib.h	
Category:	Station	
Mode:	synchronous/asynchronous	

■ Description

The **ms_genringCallerID( )** function allows transmission of analog Caller ID data (Call Originator information) to telephones equipped with FSK Caller ID detectors.

Parameter	Description
Cadid	Either a CadenceID for Distinctive rings or the current default ring (MS_RNG_DEFAULT) may be specified.
OrigAddr	An ASCII character string that holds the information about the origination party. The maximum length is 127 characters. With this feature, OrigAddr can be divided into multiple sub-fields identified by field identifiers to hold analog caller identification (FSK) transmission data.

The sub fields used by the **OrigAddr** parameter include:

- **Caller Name:** identifies the name of the call originator if available.

- **Caller Number:** identifies the number of the call originator if available.
- **Date Time:** identifies the date and time at which the call is sent.
- **Caller Number Absence Reason:** identifies why call originator's number is not available. Possible reasons are Private (P) or Out of Area (O).
- **Caller Name Absence Reason:** identifies why call originator's name is not available. Possible reasons are Private (P) or Out of Area (O).

Sub-group identifiers with format 'X:' are used to specify sub-fields for caller ID transmission. Note that the user strings embed this character as part of sub-field identifiers. Thus this sub-group identifier is implicit by nature.

The following sub-group identifiers are supported:

- **T:** identifies beginning of Time and Date sub-field.
- **N:** identifies beginning of Caller Name sub-field.
- **I:** identifies beginning of Caller Number sub-field.
- **A:** identifies beginning of Caller Number Absence Reason sub-field.
- **B:** identifies beginning of Caller Name absence Reason sub-field.

NOTES:

1. Use the character '/' as an escape character to indicate that ':' is part of the string. For example, Next string "N:J/:NamathI:993-3000" uses escape character / to embed the name J:Namath.
2. The end of a sub-field is recognized by the character ':' or the end of string when a sub-field is located at the end of the string.

■ Cautions

- This function is supported on DM3 boards only.

■ Examples

Example 1--Application sends Caller Name and Date to a Caller ID phone connected to a station.

```
OriginationAddress[128] = “T:01310930N:John Doe” which is:
Caller Name = John Doe
Date Time = Jan 31, 9 30 am
```

Example 2--Application sends Caller Number absence reason (O: Out Of Area) and Date and Time to a Caller ID phone connected to a station.

```
OriginationAddress[128] = “T:01310930A:O” which is:
Caller Number Absence Reason : Out Of Area
Date Time = Jan 31, 9 30 am
```

Example 3--Application sends proprietary data to a Caller ID phone connected to a station.

```
OriginationAddress[128] = “R:xxxxxxxxxxx” which is:
xxxx  represents the proprietary data to be sent
```

■ See Also

- **ms_SetMsgWaitInd( )**

ms_SetMsgWaitInd( ) *illuminates MWI*

Name:	ms_SetMsgWaitInd (devh, IndicatorState, rfu1, rfu2)	
Inputs:	int devh	• station device handle
	unsigned short IndicatorState	• **MS_MSGINDON** or **MS_MSGINDOFF**
	void* rfu1	• reserved for future use #1
	void* rfu2	• reserved for future use #2
Returns:	0 if success -1 if failure	
Includes:	dtilib.h srllib.h msilib.h	
Category:	Station	
Mode:	synchronous	

■ Description

The **ms_SetMsgWaitInd( )** function generates an FSK signal to illuminate the message waiting LED.

Parameter	Description
IndicatorState	This parameter allows manipulation of an FSK message waiting indicator (MWI) on a phone set. The **IndicatorState** values are: • **MS_MSGINDON:** turns the MWI on. • **MS_MSGINDOFF:** turns the MWI off.

■ Cautions

- This function is supported on DM3 boards only.
- This function can only be issued when the station is on-hook.

■ See Also

- **ms_genringCallerID( )**

10. R4 Audio Input API for DM3

10.1. AI API Functions

The three functions discussed in this chapter belong to a new Audio Input (AI) API, which is an additional feature supported on DM3 boards equipped with an audio input jack, such as the DI Series boards. The AI API is used to provide music or other information on-hold. For example, your application could arrange for music to be played to any phone connected to a supported board by listening to the timeslot number provided by the **ai_getxmitslot( )** function. See the API function descriptions and associated sample code for additional information on this process. For information on finding out how many AI devices are available, see *10.2. Determining the Number of AI Devices in a System*.

These functions are not currently documented elsewhere, but the information is detailed here for your convenience:

- **ai_open( )** - Opens an audio input device
- **ai_close( )** - Closes an audio input device
- **ai_getxmitslot( )** - Provides the SCbus time slot number of the audio input transmit channel

ai_open( ) — opens an audio input device

Name:	int ai_open(namep)	
Inputs:	const char* namep	• points to an ASCIIZ string that contains the name of a valid audio input device
Returns:	audio input device handle on success -1 on failure	
Includes:	dxxxlib.h	
Category:	Audio Input	
Mode:	synchronous	

Description

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

Parameter	Description
namep	Points to an ASCIIZ string that contains the name of a valid audio input device, in the form *aiBn*, where *n* is the audio input device number.

Example

```
#include <windows.h>        /* include in Windows applications only */
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>

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

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

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

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

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

    return 0;
}
```

See Also

- **ai_close( )**
- **ai_getxmitslot( )**
- **sr_getboardcnt( )**

ai_close( ) — closes an audio input device

Name:	int ai_close(devh)	
Inputs:	int devh	• specifies the valid device handle obtained when an audio input device is opened
Returns:	0 on success -1 on failure	
Includes:	dxxxlib.h	
Category:	Audio Input	
Mode:	synchronous	

Description

The **ai_close( )** function closes an audio input device previously opened with the **ai_open( )** function. This function releases the handle and breaks any link that the calling process has with the device through the handle.

Parameter	Description
devh	Specifies the valid device handle obtained when an audio input device is opened with the **ai_open( )** function.

Example

```
#include <windows.h>      /* Include in Windows applications only */
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>

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

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

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

```
    sc_tsinfo.sc_tsarrayp = &scts;

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

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

     return 0;
}
```

See Also

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

AI API for DM3

ai_getxmitslot( ) — provides the SCbus time slot number

Name:	int ai_getxmitslot(devh, sc_tsinfo)	
Inputs:	int devh	• specifies the valid device handle obtained when an audio input device is opened
	SC_TSINFO* sc_tsinfo	• provides a pointer to the SCbus time slot information structure
Returns:	0 on success -1 on failure	
Includes:	dxxxlib.h	
Category:	Audio Input	
Mode:	synchronous	

■ Description

The **ai_getxmitslot( )** function provides the SCbus time slot number of the audio input transmit channel. The SCbus timeslot information is contained in an SC_TSINFO structure.

Parameter	Description
devh	Specifies the valid device handle obtained when the audio input device is opened with the **ai_open( )** function.
sc_tsinfo	Specifies a pointer to the SCbus timeslot information structure. The **sc_numts** member of the SC_TSINFO structure must be initialized with the number of SCbus time slots requested (1). The **sc_tsarrayp** member of the SC_TSINFO structure must be initialized with a valid pointer to a long variable. Upon successful return from the function, the long variable will contain the number of the SCbus time slot on which the audio input device transmits.

Example

```
#include <windows.h>          /* Include in Windows applications only */
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>

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

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

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

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

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

See Also

- **ai_open( )**
- **ai_close( )**

10.2. Determining the Number of AI Devices in a System

For the Audio Input (AI) board devices, there is a new class name, DEV_CLASS_AUDIO_IN, that is used with the **sr_getboardcnt( )** function to return the number of AI devices in the system. So, if you issue **sr_getboardcnt**(DEV_CLASS_AUDIO_IN, &BrdCt), BrdCt is filled in with the number of AI board devices in the system.

For example:

```
int BrdCt = 0;
sr_getboardcnt(DEV_CLASS_AUDIO_IN, &BrdCt);
```

The device name for AI devices is *aiBn*, where n is the board number starting at 1. Note that there are no channel devices on AI devices, only board devices.

11. Implementing Applications for R4 on DM3

11.1. General Considerations for Developing or Porting R4 on DM3 Applications

The following information describes how to develop applications on R4 for DM3, with a focus on modifying existing R4 API applications so that they can accommodate DM3 boards.

With R4 for DM3, applications supporting Span Cards can be expanded to provide support for DM3 hardware as well. DM3 is a powerful DSP architecture introduced by Dialogic to provide customers with greater channel density and performance for building CTI solutions on PCI and cPCI bus architectures.

R4 for DM3 uses a subset of functions supported in R4, hence the application developer must make some trade-offs and decisions when migrating to R4 for DM3.

DM3 voice devices may only be attached to DM3 network devices.

11.1.1. Using GlobalCall for Call Control

One of the challenges of migrating a Dialogic digital interface application from R4 to R4 for DM3 lies in the lack of R4 for DM3 support for much of the DTI API functionality that many legacy applications use today. In particular, this is true for T-1 robbed-bit applications written prior to Dialogic introducing GlobalCall support for T-1 robbed-bit digital trunks.

The use of GlobalCall simplifies the life of the CTI application developer, since the GlobalCall level of abstraction allows for seamless support for any telephony interface, including T-1, E-1, ISDN or even analog. Once an application has been designed to use GlobalCall, minimum changes (if any) are required for the same application to run on various Dialogic hardware, including D/4x, Span Cards, DualSpan and DM3 boards as well.

It is a good architectural decision to use GlobalCall because of the greater flexibility and portability provided by GlobalCall. This is true, not just for R4 for DM3, but for any CTI application that uses Dialogic hardware.

11.1.2. DTI API versus GlobalCall

The standard R4 DTI API presents several functions--for example, SCbus routing, network interface alarms, and timeslot signaling control.

The standard GlobalCall API presents a higher level of call control abstraction than the DTI API.

There are numerous digital interface telephony protocols in use worldwide. To name some, there are robbed-bit T-1, CAS E-1, ISDN, SS7, IP, and ATM. They all have one thing in common: they all try to solve the problem of connecting two or more people/machines anywhere in the world in direct conversation. Once the connection has been established, various data and voice streaming mechanisms can be used, such as digitized human voice, IP packets, or any other digital data.

Those protocols mentioned above all use a similar high level "layer 3" protocol. The end result is that one end can initiate a call (make call), be informed of an incoming call, or drop the call. The GlobalCall API presents the developer with a similar level of abstraction at the API level, hiding the internals of the specific protocol.

That is, in order to make a call under a T-1 robbed-bit trunk, the protocol indicates that one must flip the A & B signaling bits, while to do the same under an ISDN PRI protocol, one must send a specific HDLC packet over the ISDN data channel. All of these complicated operations are hidden from the GlobalCall developer.

It is technically possible to design a GlobalCall application in such a way that the same application can run with an E-1 CAS trunk or an E-1 ISDN trunk without requiring changes.

GlobalCall is the API of choice for R4 for DM3 for a number of reasons. These are some of the most obvious:

- GlobalCall presents the right level of abstraction for rapid digital interface telephony application deployment.
- The existing DM3 architecture with the TSC resource does not provide access to low-level channel associated signaling (CAS, robbed-bit), hence most of the DTI API cannot be provided.
- GlobalCall also enables easier digital network interface to analog network interface portability, where analog network interface is supported.

11.1.3. ISDN Primary Rate API versus GlobalCall

Many existing R4 applications today make use of the rich Dialogic ISDN Primary Rate API. This API has been evolving over time, and provides access to two levels of abstraction, known as layer 3 and layer 2.

R4 for DM3 does not provide access to the ISDN Primary Rate API; however, much of its layer 3 functionality can be accomplished today directly and at a similar level of abstraction using GlobalCall.

To port an existing R4 application to R4 for DM3, you must replace the ISDN Primary Rate functions with equivalent GlobalCall functions. Most of the ISDN Primary Rate functions have the “cc_” function name prefix and are documented in the *ISDN Software Reference*. If you compare the *GlobalCall API Software Reference* with the *ISDN Software Reference*, you will see the great similarity between the two APIs, and that it should require a minimal effort to port an application that uses the Primary Rate API to an application that uses the GlobalCall API.

11.1.4. Programming Models in Linux

For Linux, all existing SRL programming models are preserved under R4 for DM3, except for the Signal Callback Model, which is not supported.

Asynchronous event notification is done via the SRL in the same exact way it is currently done with standard R4 devices.

11.2. Multi-Threading and Multi-Processing

The APIs discussed in this Guide support multi-threading and multi-processing, with some restrictions on the latter. The restrictions on multi-processing are outlined below.

First, one particular channel can only be opened in one process at a time. There can, however, be multiple processes accessing different sets of channels. In other words, ensure that each process is provided with a unique set of devices to manipulate.

Second, if a channel was opened in process A and then closed, process B is then allowed to open the same channel. However, since closing a channel is an asynchronous operation in R4 on DM3, there is a small gap between the time when the **xx_close( )** function returns in process A and the time when process B is allowed to open the same channel. If process B opens the channel too early, things could go wrong. For this reason, this type of sequence should be avoided.

Third, there is a restriction on use of firmware tones in case of multi-processing in the Voice API: Multiple processes that define tones (GTD or GTG) do not share tone definitions in the firmware. That is, if you define tone A in process #1 for channel dxxxB1C1 on DM3 board X and the same tone A in process #2 for channel dxxxB1C1 on the same DM3 board X, two firmware tones are consumed on board X. In other words, the same tone defined from different processes is not shared in the firmware; hence this limits the number of tones that can be created overall.

Fourth, with the DCB API, multiple processes cannot access the same conference.

11.3. Initializing an R4 on DM3 Application

DM3 devices have similar characteristics to earlier-generation devices. The device must first be opened in order to obtain its handle, which can then be used to access the device functionality.

Since R4 for DM3 applications must use GlobalCall for call control, i.e., for call setup and tear-down, all Dialogic network interface devices must be opened using the **gc_Open( )** or **gc_OpenEx( )** function. Where call control is not required, such as with ISDN NFAS, **dt_open( )** may be used to open DM3 network interface devices.

Once the call has been established, voice and or data streaming should be done using the Dialogic voice API. Functions such as **dx_playiottdata( )**, **dx_reciottdata( )**, and **dx_dial( )** can be used. Of course, in order to do so, the voice device handle must be obtained. By the same token, in order to access the DTI routing functionality with functions such as **dt_listen( )** and **dt_unlisten( )**, the corresponding DTI device handle must be obtained accordingly.

Application initialization differs depending on the types of hardware and the APIs used. The simplest hardware and API scenario is that where the system contains only one type of board, so that the application uses either earlier-generation boards or DM3 boards but not both. In these cases, the initialization routine is the simplest in that it does not need to discover the board family type. See *Section 11.3.1. Initializing the GlobalCall API for DM3 Boards Only (Flexible Routing)* for more information.

Applications that want to make use of both earlier-generation and DM3 devices must have a way of differentiating what type of device is to be opened. The SCbus routing device information functions, such as **dx_getctinfo( )** and **dt_getctinfo( )**, provide a programming solution to this problem. DM3 hardware is identified by the CT_DFDM3 value in the **CT_DEVINFO.ct_devfamily** field. Only DM3 devices will have the **ct_devfamily** field set to CT_DFDM3. See *Section 11.3.2. Device Discovery for DM3 Boards and Earlier-Generation Boards (Flexible Routing)* for more information.

The following guidelines for initializing applications apply to an R4 on DM3 flexible routing configuration.

11.3.1. Initializing the GlobalCall API for DM3 Boards Only (Flexible Routing)

This scenario is one where an R4 for DM3 application uses only DM3 boards in a flexible routing configuration.

When initializing an application to use DM3 boards, you must use GlobalCall to handle the call control. Initializing GlobalCall with only DM3 boards in the system is no different than when initializing GlobalCall with only earlier-generation devices in the system. This is because R4 on DM3 is flexible enough to support the different methods of GlobalCall initialization for both ISDN and CAS protocols.

Take note of the following flexibility that exists for the **gc_Open( )** or **gc_OpenEx( )** functions when opening a GlobalCall line device on DM3 boards:

- Due to the nature of DM3, the protocol name is irrelevant at the time of opening the GlobalCall line device; that is, the protocol name is ignored. Although it is unnecessary to specify a protocol name, you can retain a protocol name in this field to support earlier-generation boards and so as to retain compatibility with code for earlier-generation boards. Also, for R4 on DM3, all protocols are bi-directional. You do not need to dynamically open and close devices to change the direction of the protocol.
- It is not necessary to specify a voice device name when opening a GlobalCall line device. If you specify the voice device name, the network interface device is automatically associated with the voice device (they are attached and routed on the CT Bus). If you do not specify the voice device name when you open the GlobalCall line device, you can separately open a voice device, and then attach and route it to the network interface device. This flexibility allows you to port a GlobalCall application from earlier-generation boards to DM3 boards with little change and regardless of whether the application uses an ISDN or CAS protocol.

  Note that when opening a GlobalCall line device for CAS protocols on earlier-generation boards, the voice device name, network interface device name, and protocol name are required; otherwise the function fails. For ISDN protocols on earlier-generation boards, it is invalid to specify a voice device name; otherwise the function fails. For DM3 boards in a flexible routing configuration, only the network device name is required.

The following procedure shows how to initialize GlobalCall when using only DM3 boards (in a flexible routing configuration). For some steps, two alternatives are described (A and B), depending upon whether you want your application to retain the greatest degree of compatibility with GlobalCall using an ISDN protocol or a CAS protocol on earlier-generation boards. Since this procedure is oriented toward retaining compatibility with two common ways to initialize GlobalCall on earlier-generation boards, it is not intended as a recommendation of a preferred way to initialize GlobalCall on DM3 boards. The R4 GlobalCall API on DM3 allows design flexibility. The procedure for GlobalCall initialization for a given application would depend upon things such as whether earlier-generation boards and DM3 boards are used in the same system, what protocol is used, the purpose of the application program, and its design.

NOTE: In Windows, use the **sr_getboardcnt( )** function with the class name set to DEV_CLASS_DTI and DEV_CLASS_VOICE to determine the number of network and voice boards in the system, respectively. In Linux, use SRL device mapper functions to return information about the structure of the system. These functions are described in the *Voice Software Reference - Standard Run-time Library*.

1. Start/Initialize GlobalCall with **gc_Start( )**.
2. Use **gc_Open( )** or **gc_OpenEx( )** to open a GlobalCall line device.
   - Specify the network interface device name and the protocol name in the **devicename** parameter, as in the following example:

     ":N_dtiB1T1:P_ISDN"
   - Alternatively, specify the network interface device name, **the voice device name**, and the protocol name in the **devicename** parameter, as in the following example:

     ":N_dtiB1T1:V_dxxxB1C1:P_ar_r2_io"
3. Obtain the voice channel device handle.
   - Open a voice channel device (e.g., dxxxB1C1) with **dx_open( )** to get its handle.
   - Alternatively, if you specified the voice device name in the **devicename** parameter in step *2*, use **gc_GetVoiceH( )** to get the handle.
4. Attach the voice and network interface devices.
   - Use **gc_Attach( )** to attach the voice resource and the network interface line device.
   - Alternatively, if you specified the voice device name in the **devicename** parameter in step *2*, the voice and network interface devices are attached by nature of the **gc_Open( )**, so no action is necessary for this step.

5. Use **gc_GetNetworkH( )** to obtain the network interface timeslot device handle that is associated with the line device.
6. Set up SCbus full duplex routing between the network interface device and voice device.
   - Use **nr_scroute(FULL DUPLEX)**.
   - Alternatively, if you specified the voice device name in the **devicename** parameter in step *2*, the network interface device and voice device are automatically routed on the SCbus by nature of the **gc_Open( )**.

Repeat steps *2* to *6* for all GlobalCall device names.

11.3.2. Device Discovery for DM3 Boards and Earlier-Generation Boards (Flexible Routing)

The following procedure shows how to initialize an R4 on DM3 application and perform **device discovery** when the application supports both DM3 and earlier-generation boards.

NOTE: In Windows, use the **sr_getboardcnt( )** function with the class name set to DEV_CLASS_DTI and DEV_CLASS_VOICE to determine the number of network and voice boards in the system, respectively. In Linux, use SRL device mapper functions to return information about the structure of the system. These functions are described in the *Voice Software Reference - Standard Run-time Library*.

1. Open the first network interface timeslot device (e.g., dtiB1T1) on the first trunk with **dt_open( )**.
2. Call **dt_getctinfo( )** and check the **CT_DEVINFO.ct_devfamily** value.
3. If **ct_devfamily** is CT_DFDM3, then flag all the network interface timeslot devices associated with the trunk as DM3 type.
4. Close the DTI device with **dt_close( )**.
5. Repeat steps *1* to *4* for each trunk.
6. Open the first voice channel device on the first voice board in the system with **dx_open( )**.
7. Call **dx_getctinfo( )** and check the **CT_DEVINFO.ct_devfamily** value.

8. If **ct_devfamily** is CT_DFDM3, then flag all the voice channel devices associated with the board as DM3 type.
9. Close the voice channel with **dx_close( )**.
10. Repeat steps *6* to *9* for each voice board.
11. For those voice and network interface devices that are **not** DM3 devices, proceed with the standard initialization process for earlier-generation boards as performed in the original application.
12. For those voice and network interface devices that are DM3 devices, proceed with the initialization as described in *Section 11.3.1. Initializing the GlobalCall API for DM3 Boards Only (Flexible Routing)*.

12. R4 on DM3 Resource Routing and Cluster Configuration

12.1. Fixed and Flexible Routing Configurations

R4 on DM3 supports two types of routing configuration (one is primarily for backward compatibility with previous releases of R4 on DM3, while the other provides routing capabilities compatible with the R4 API):

- **Fixed Routing Configuration:** This configuration is primarily for backward compatibility with R4 on DM3 in DNA 3.3 and System Release 5.0. The fixed routing configuration applies only to DM3 boards. With fixed routing, the resource devices (voice/fax) and network interface devices are permanently coupled together in a fixed configuration. Only the network interface timeslot device has access to the CT Bus. For example, on a DM/V960-4T1 board, each voice resource channel device is permanently routed to a corresponding network interface timeslot device on the same physical board. The routing of these resource and network interface devices is predefined and static. The resource device also does not have access to the CT Bus and so cannot be routed independently on the CT Bus. No off-board sharing or exporting of voice/fax resources is allowed.
- **Flexible Routing Configuration:** This configuration is compatible with R4 API routing on earlier-generation boards; that is, earlier-generation boards use flexible routing. Flexible routing is available for DM3 boards starting with R4 on DM3 in System Release 5.01. With flexible routing, the resource devices (voice/fax) and network interface devices are independent, which allows exporting and sharing of the resources. All resources have access to the CT Bus. For example, on a DM/V960-4T1 board, each voice resource channel device and each network interface timeslot device can be independently routed on the CT Bus.

These routing configurations are also referred to as cluster configurations, because the routing capability is based upon the contents of the DM3 cluster.

The fixed routing configuration is one that uses permanently **coupled resources**, while the flexible routing configuration uses **independent resources**. From a

DM3 perspective, the **fixed routing cluster** is restricted by its coupled resources and the **flexible routing cluster** allows more freedom by nature of its independent resources, as shown in *Figure 3. Cluster Configurations for Fixed and Flexible Routing*.

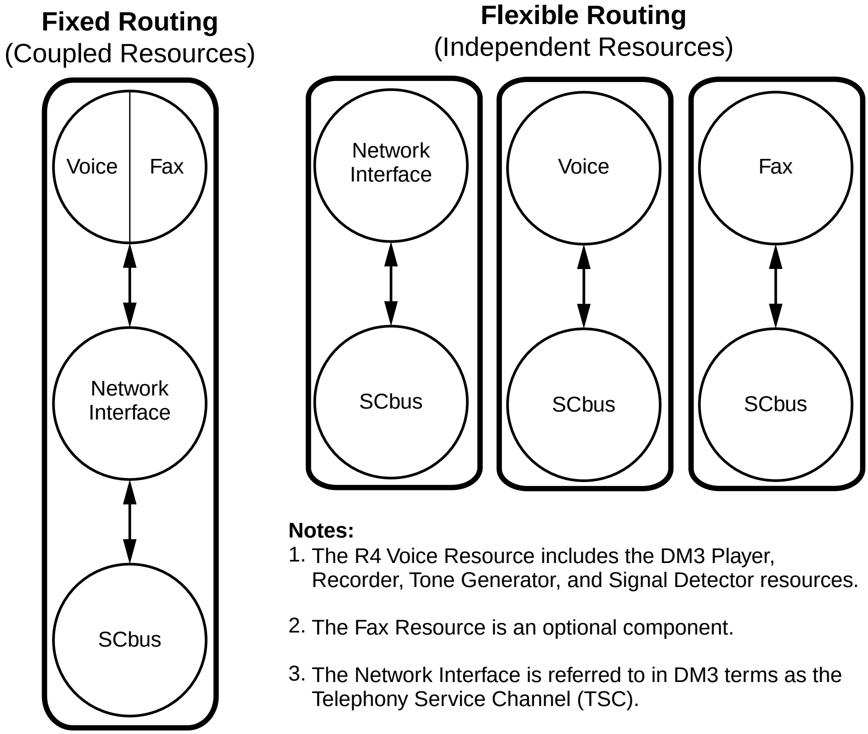


Figure 3. Cluster Configurations for Fixed and Flexible Routing

12.1.1. Selecting a Fixed or Flexible Routing Configuration

You select the routing configuration when you use the **Dialogic Configuration Manager** (DCM) to assign a **firmware file** (pcd file) to each board in your Dialogic product configuration. In other words, by choosing, in the DCM “Assign Firmware File” dialog a DM3 **product configuration description (pcd) file** to be used with a board, you select the routing configuration for that board. Only fixed routing product descriptions identify the type of routing configuration. If the DCM displays a "Firmware Description" that is not identified as supporting fixed routing, the product supports flexible routing. The routing configuration takes effect at board initialization.

The availability of flexible routing for a specific Dialogic product depends upon the Dialogic software release in which the product is supported. Some products support fixed routing only and other products support flexible routing only. In other cases, Dialogic provides a choice of fixed or flexible routing, such as products using ISDN. In this case, you should select the product that provides the desired routing configuration (either fixed or flexible)

NOTE

Fixed and flexible routing are supported on Dialogic products as indicated in the Dialogic software *Release Guide* and *Release Updates*, so refer to those documents for information on specific product support.

You can only select the routing configuration at the product level. You cannot select the routing configuration at a resource level or configure a board to use fixed routing for some of its resources and flexible routing for other resources.

12.1.2. Overview of DM3 Clusters

Because DM3 clusters impact the operation of SCbus routing with R4 for DM3 (notably the fixed or flexible routing capability), it may be helpful to understand some basic information about clusters. The following information is provided as a short summary of background material and is not essential for using DM3 boards in a fixed or flexible routing configuration. However, understanding the underlying model of DM3 clusters may be helpful to some people. For in-depth background information, see the *DM3 Mediastream Architecture Overview* document.

- The DM3 mediastream architecture uses the concept of clusters to provide a logical grouping of DM3 resources. DM3 clustering has an effect on the R4 switching and resource/device management model. The clustering method affects the routing capabilities, as well as how an application obtains, uses, and closes device handles. Each cluster provides access to an SCbus transmit timeslot and acts as the communication manager for the cluster's components. All switching is accomplished using a cluster.

- In the DM3 model, a standard R4 API voice resource consists of a combination of several different DM3 resources grouped together in a cluster: The voice resource includes a Player, Recorder, Tone Generator, and Signal Detector. The network interface timeslot is referred to as a Telephony Service Channel (TSC). The SCbus resource provides access to the CT Bus.
- In DM3 terms, fixed routing indicates that the network interface device is combined with the resource device in a single cluster. On the other hand, flexible routing indicates that the network interface device and the resource device exist in separate clusters. See *Figure 3. Cluster Configurations for Fixed and Flexible Routing*.

12.2. Additional API Restrictions in an R4 on DM3 Fixed Routing Configuration

The following restrictions apply to DM3 fixed routing configurations only.

NOTE: Except where explicitly stated, these restrictions are **in addition** to those described for the DM3 flexible routing configuration.

The following areas are restricted or not supported when using a DM3 fixed routing configuration:

- SCbus voice resource routing is not supported
- SCbus fax resource routing restricted
- Voice, Fax, and GlobalCall resource/device management restricted

Table 19. List of Additional API Functions Restricted and Not Supported in a DM3 Fixed Routing Configuration

Function Name	Notes
dx_close( )	**Limitations:** Although **dx_open( )** and **dx_close( )** are operational on DM3 voice devices, their purpose is extremely limited by nature of the voice resource membership in a DM3 cluster. (Instead, you must use the **gc_OpenEx( )**, **gc_GetVoiceH( )**, and **gc_Close( )** functions.)
dx_getxmitslot( )	Not supported. The function fails with error code EDX_SH_MISSING, indicating "Switching Handler is not present".
dx_listen( )	Not supported. The function fails with error code EDX_SH_MISSING, indicating "Switching Handler is not present".
dx_open( )	**Limitations:** Although **dx_open( )** and **dx_close( )** are operational on DM3 voice devices, their purpose is extremely limited by nature of the voice resource membership in a DM3 cluster. (Instead, you must use the **gc_OpenEx( )**, **gc_GetVoiceH( )**, and **gc_Close( )** functions.)
dx_unlisten( )	Not supported. The function fails with error code EDX_SH_MISSING, indicating "Switching Handler is not present".
fx_close( )	**Limitations:** See **fx_open( )**.
fx_getxmitslot( )	Not supported. The function fails with error code EDX_SH_MISSING, indicating "Switching Handler is not present".
fx_listen( )	Not supported. The function fails with error code EDX_SH_MISSING, indicating "Switching Handler is not present".

Function Name	Notes
fx_open( )	**Limitations:** The following additional limitations apply to fax resources in a DM3 fixed routing configuration. Because of the fax resource membership in a DM3 cluster, the voice and fax devices are automatically associated with a network interface device. Therefore, it is recommended that you do the following to retrieve and use the fax device handle: • Use **gc_Open( )** to open the network interface line device. • Use **gc_GetVoiceH( )** to retrieve the voice device handle. • Use **ATDV_NAMEP( )** to retrieve the voice device name for the voice device handle returned by the **gc_GetVoiceH( )** function. • Use **fx_open( )** on this voice device name to open the associated fax device and retrieve the fax device handle. The function will succeed if the channel device has fax capabilities; otherwise the function will fail. • Use **fx_close( )** to close the fax device.
fx_unlisten( )	Not supported. The function fails with error code EDX_SH_MISSING, indicating "Switching Handler is not present".
gc_Attach( )	Not supported. See *Section 12.3.2. Associating Network and Voice Devices (Fixed Routing)* for more information.
gc_Detach( )	Not supported. See *Section 12.3.2. Associating Network and Voice Devices (Fixed Routing)* for more information.

Function Name	Notes
gc_Open( )	**Limitations**: See *Sections 12.3.1. gc_Open( ) and gc_OpenEx( ) Restrictions (Fixed Routing)*, *12.3.2. Associating Network and Voice Devices (Fixed Routing)*, and *12.4.2. Getting, Using, and Closing Device Handles in a DM3 Fixed Routing Configuration* for more information.
gc_OpenEx( )	**Limitations**: See *Sections 12.3.1. gc_Open( ) and gc_OpenEx( ) Restrictions (Fixed Routing)*, *12.3.2. Associating Network and Voice Devices (Fixed Routing)*, and *12.4.2. Getting, Using, and Closing Device Handles in a DM3 Fixed Routing Configuration* for more information.
nr_scroute( )	**Limitations:** Does not support voice, fax, analog network interface devices (LSI), or MSI devices. Supports DTI devices only.
nr_scunroute( )	**Limitations:** Does not support voice, fax, analog network interface devices (LSI), or MSI devices. Supports DTI devices only.

12.3. Using the GlobalCall API in an R4 on DM3 Fixed Routing Configuration

12.3.1. gc_Open( ) and gc_OpenEx( ) Restrictions (Fixed Routing)

In R4 applications that use a DM3 fixed routing configuration, it is not possible to specify the protocol and voice device in the ASCII string pointed to by the **devicename** parameter in **gc_Open( )** and **gc_OpenEx( )** commands.

The **devicename** can be as simple as:

```
:N_dtiB1T1
```

where:

- N_ denotes a network timeslot device, in this example, dtiB1T1

The following restrictions apply:

- The network timeslot device field (N_) is required.
- The protocol identifier field (P_), if specified, is ignored. The protocol name is unnecessary here because it is selected when using Dialogic Configuration Manager for PCD/FCD file selection. Also, for R4 on DM3 products, all protocols are bi-directional. You do not need to dynamically open and close devices to change the direction of the protocol. For R4 on earlier-generation devices, most protocols are unidirectional.
- The voice channel device field (V_), if set to a value other than the voice device automatically assigned during download, causes the **gc_Open( )** command to fail. Voice devices are automatically associated with network devices as part of the cluster configuration during firmware download.

12.3.2. Associating Network and Voice Devices (Fixed Routing)

For R4 applications that use earlier-generation boards, it is possible to open a line device using the **gc_Open( )** function, open a voice device using the **dx_open( )** function, then use the **gc_Attach( )** function to associate the voice device with the line device, using the line device ID and the voice device handle.

For R4 applications that use DM3 fixed routing configurations, this is neither necessary nor possible. A voice device is automatically associated with a line device as part of a DM3 cluster configuration during DM3 initialization. Therefore, the **gc_Attach( )** and **gc_Detach( )** functions are not supported.

NOTE: Including a voice device name that is different than the voice device automatically associated with the line device, in the **devicename** parameter of the **gc_Open( )** function returns an error.

12.3.3. ISDN Direct Layer 2 Access (Fixed Routing)

For R4 applications that use a DM3 fixed routing configuration, the **gc_GetFrame( )** and **gc_SndFrame( )** functions are supported. The DM3 firmware supports direct layer 2 access in DM3 fixed routing configurations.

12.4. Operating in an R4 on DM3 Fixed Routing Configuration

All information specific to using a fixed routing configuration is provided here, including information on interoperability, device management, and guidelines and examples for implementing an application.

12.4.1. Interoperability with DM3 Application Foundation Code

The DM3 fixed routing configuration is consistent with the clustering operation used in the DM3 Application Foundation Code (AFC).

12.4.2. Getting, Using, and Closing Device Handles in a DM3 Fixed Routing Configuration

Since an R4 for DM3 application must use GlobalCall for call control on DM3 boards, the DM3 network interface devices must be opened with the **gc_Open( )** or **gc_OpenEx( )** function. Where call control is not required, such as with ISDN NFAS, **dt_open( )** may be used to open DM3 network interface devices.

Also, since the voice device and network interface timeslot in a DM3 fixed routing configuration are permanently routed and attached to each other, SCbus routing and attaching of the devices is unnecessary with DM3 hardware. The same is true of a fax device and its network interface timeslot in the DM3 fixed routing configuration. The resource device (voice/fax) and its associated DTI network interface timeslot on the same physical port are tied together in a DM3 cluster. The resource channel is explicitly tied to the network interface timeslot; and the voice resource cannot be shared or separately routed to another network interface device. Thus, the resource device and associated network interface device are bound together in a static link, and there is no support for routing the resource independently to the SCbus. However, the DTI network interface can be routed to the SCbus, allowing access to other SCbus resources.

With the DM3 fixed routing configuration, the voice or fax resource device has no direct access to the CT Bus timeslots and it is neither necessary nor possible to attach the resource and network interface devices together from the host. Therefore, in most cases, an R4 for DM3 application does not need to open the voice channel device and/or the network interface timeslot device directly using the **dx_open( )** and **dt_open( )** functions, respectively. However, the fax device must explicitly be opened using **fx_open( )**. Although the fax device also has no direct access to the CT Bus timeslots and therefore cannot be routed over the CT Bus, it is necessary to retrieve the fax device handle to do fax operations.

DM3 Fixed Routing Configuration and Device-Handling Guidelines for the GlobalCall API

The following summary applies only to a DM3 fixed routing configuration and presents some guidelines for DM3 device-handling using the GlobalCall API.

- Use **gc_Start( )** to initialize GlobalCall prior to using any devices.
- Use **gc_Open( )** to open the voice resource and network interface devices; then use **gc_GetVoiceH( )** to retrieve the voice device handle and **gc_GetNetworkH( )** to retrieve the network interface device handle.

  The **gc_Open( )** function internally opens the associated voice channel on DM3 boards. Do **not** specify the name of the voice device channel in the **gc_Open( )** device string. The device name string for the **gc_Open( )** function should look similar to the following (see *Section 12.3.1. gc_Open( ) and gc_OpenEx( ) Restrictions (Fixed Routing)* for more information):

  ```
  ":N_dtiB1T1:P_ISDN"
  ```

- Since the fax library does not support the use of a voice handle for fax commands, you cannot use the device handle from **dx_open( )** to call fax API functions. You must use **fx_open( )** to open a channel device for fax processing and use that fax device handle. Therefore, it is recommended that you do the following to retrieve and use the fax device handle:
  - Use **ATDV_NAMEP( )** to retrieve the voice device name for the voice device handle returned by the **gc_GetVoiceH( )** function.
  - Use **fx_open( )** on this voice device name to open the associated fax device and retrieve the fax device handle.
- The voice device is automatically and permanently associated with and connected to the network interface line device, so the **gc_Attach( )** and **gc_Detach( )** functions cannot be used and are not supported for DM3 boards.

- In general, when using DM3 **and** earlier-generation boards, the R4 for DM3 application must use a **device-discovery** procedure to become "hardware-aware." To perform device discovery and identify whether a logical device belongs to a DM3 board, use the SCbus routing device information functions, such as **dx_getctinfo( )** and **dt_getctinfo( ),** and check the CT_DEVINFO **ct_devfamily** field for the CT_DFDM3 value. For details see *Section 3.3. CT_DEVINFO: Channel/Timeslot Device Information Data Structure* and *Section 12.5.2. Initializing an Application Under Different Hardware and API Scenarios (Fixed Routing)*.
- Application performance may be a consideration when opening and closing devices with GlobalCall. If the application uses GlobalCall to dynamically open and close devices as needed, it can impact the application's performance. One way to avoid this is to open all DM3 devices during application initialization and keep them open for the duration of the application, closing them only at the end.

12.5. Implementing Applications in an R4 on DM3 Fixed Routing Configuration

12.5.1. Porting an Application to R4 on DM3 with Fixed Routing

The following information describes how to develop applications for an R4 for DM3 fixed routing configuration, with a focus on modifying existing R4 API applications so that they can accommodate DM3 boards.

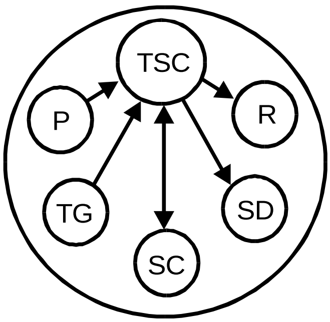
This information addresses general areas for migrating an existing R4 application to an R4 for DM3 application and provides examples from porting the existing Dialogic PANSR (polled asynchronous) Answering Machine demonstration program to R4 for DM3.

NOTE: Although it is not reiterated throughout the following information, it is assumed that the information applies specifically to a fixed routing configuration and not a flexible routing configuration.

CT Bus, SCbus and DM3 Clusters with Fixed Routing

General information on DM3 clustering is available in *Section 12.1. Fixed and Flexible Routing Configurations*. The following information applies to coupled resource clusters, which are used in the fixed routing configuration.

- The DM3 architecture uses the concept of clusters to provide a logical grouping of DM3 resources. For the QuadSpan DM3 product and the IPLink DM3 product, there are 96 clusters in T-1 boards and 120 clusters in E-1 boards, configured by default with a Telephony Service Channel (TSC) component, a Signal detector, a Tone generator, a Player, a Recorder, and an SCbus component as shown in *Figure 4*. This configuration is based on the DM3 cluster for DM/V960-4T1 and DM/V1200-4E1 boards. Note that for other DM3 boards, the cluster may not contain all of the DM3 resources.



P	Player Resource
TG	Tone Generator
SC	SCbus Resource
SD	Signal Detector
R	Recorder Resource
TSC	Telephony Service Resource

Figure 4. Default DM3 Fixed Routing Cluster for QuadSpan DM/V960-4T1 and DM/V1200-4E1 Boards

DM3 Clusters & Routing

Figure 5 shows a default DM3 cluster for DM/VF boards such as DM/VF240-T1 and DM/VF300-E1. Note the addition of a fax resource in the same cluster.

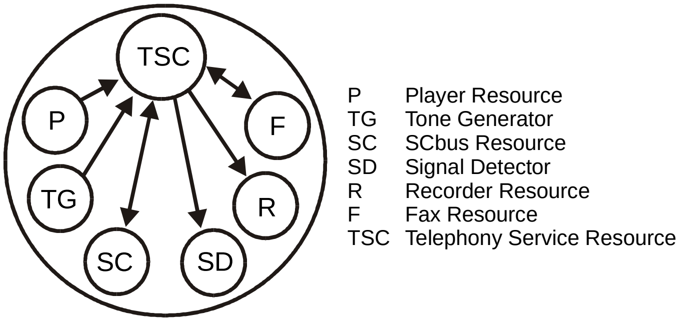


Figure 5. Default DM3 Fixed Routing Cluster with Fax Resource for DM/VF Boards

The SCbus resource is the DM3 resource that provides access to the CT Bus. As indicated in *Figure 4* and *Figure 5*, the only DM3 resource that has access to the SCbus resource is the Telephony Service Channel (TSC). None of the other resources have access to the SCbus resource and therefore cannot access the CT Bus directly.

NOTE: In the following discussion, **voice channel device** and **voice device** refer to voice-capable or voice/fax-capable devices.

In R4 for DM3, the TSC resource instance is abstracted by the **DTI timeslot device**, or dtiBnTm, while the rest of the resource instances are all grouped and abstracted by the **voice channel device**, or dxxxBnCm.

The immediate consequence of this DM3 clustering scheme is that the DTI devices, dtiBnTm, have full access and provide full support to the R4 SCbus routing API. On the other hand, the voice devices, dxxxBnCm, do not have direct access to the CT Bus, hence the R4 voice resource SCbus routing API has no effect on the DM3 voice devices.

Note that even though the DM3 hardware uses the CT Bus, R4 for DM3 abstracts its functionality using the R4 SCbus routing API.

The preceding concepts present what are perhaps the most challenging aspects of porting an existing R4 application to R4 for DM3. Here are some points to note about this configuration:

- From an R4 point of view, it appears as if the dtiBnTm network interface and the R4 dxxxBnCm voice channel device are tied together, thus a default full duplex connection exists between the network interface timeslot and voice channel devices.
- Since the player and tone generator do not get routed off board, there is no reference point for an off-board ASR resource to perform echo cancellation. The issue specifically arises in terms of the Antares card performing speech recognition. The Antares ASR resource is receiving data from the network via an SCbus connection with the DM3 TSC. However the Antares ASR resource might also need to listen to what is being transmitted out the network interface in order to perform echo cancellation. Due to the cluster configuration, the Antares ASR has no external reference point (i.e., SCbus timeslot) for echo cancellation.

R4 for DM3 applications can continue using SCbus routing functions on the DTI network interface device, dtiBnTm, because DM3 clusters do not affect their operation.

The application can therefore find the CT Bus transmit timeslot and attach (listen) or de-attach (unlisten) a DTI timeslot device as desired. This is of particular interest for those applications that wish to connect two network interface devices in conversation, so that an inbound caller can be connected in conversation with an outbound call.

R4 for DM3 applications should not attempt to perform SCbus routing on voice channel devices, since the API function call would fail with error code EDX_SH_MISSING, indicating that the operation is not possible on the device.

12.5.2. Initializing an Application Under Different Hardware and API Scenarios (Fixed Routing)

DM3 devices have similar characteristics to earlier-generation devices. The device must first be opened in order to obtain its handle, which can then be used to access the device functionality.

Since R4 for DM3 applications must use GlobalCall for call control, i.e., for call setup and tear-down, all Dialogic network interface devices must be opened using the **gc_Open( )** or **gc_OpenEx( )** function.

Once the call has been established, voice and or data streaming should be done using the Dialogic voice API. Functions such as **dx_playiottdata( )**, **dx_reciottdata( )**, and **dx_dial( )** can be used. Of course, in order to do so, the voice device handle must be obtained. By the same token, in order to access the DTI routing functionality with functions such as **dt_listen( )** and **dt_unlisten( )**, the corresponding DTI device handle must be obtained accordingly.

Application initialization differs depending on the types of hardware and the APIs used. The simplest hardware and API scenarios are those where the system contains only one type of board, so that the application uses either earlier-generation boards or DM3 boards but not both. In these cases, the initialization routine is the simplest in that it does not need to discover the board family type. See the following sections for descriptions and procedures for the stated API and hardware combinations:

- *Section GlobalCall API with Earlier-Generation Boards Only*
- *Section GlobalCall API with DM3 Boards Only (Fixed Routing)*

Applications that want to make use of both earlier-generation and DM3 devices must have a way of differentiating what type of device is to be opened.

The **dx_getctinfo( )** and **dt_getctinfo( )** functions provide a programming solution to this problem. R4 for DM3 contains a new equate for the **CT_DEVINFO.ct_devfamily**, equal to CT_DFDM3. Only DM3 devices will have the **ct_devfamily** set to CT_DFDM3.

See the following sections for descriptions and procedures for the stated API and hardware combinations:

- *Section GlobalCall API with DM3 and Earlier-Generation Boards (Fixed Routing)*
- *Section GlobalCall API with DM3 Boards and DTI API with Earlier-Generation Boards (Fixed Routing)*

GlobalCall API with Earlier-Generation Boards Only

This scenario is one where a standard R4 application uses only earlier-generation boards. Standard R4 digital interface applications typically perform the following initialization routine:

NOTE: Use the **sr_getboardcnt( )** function with the class name set to DEV_CLASS_DTI and DEV_CLASS_VOICE to determine the number of network and voice boards in the system, respectively. In Linux, use SRL device mapper functions to return information about the structure of the system. These functions are described in the *Voice Software Reference - Standard Run-time Library*.

1. Start/Initialize GlobalCall with **gc_Start( )**.
2. Open a GlobalCall device using **gc_Open( )**.
3. Open a DTI device using **dt_open( )**.

   NOTE: Step *3* might not be necessary in certain protocols where both the GlobalCall and DTI device handles are the same.
4. Open a voice device using **dx_open( )**.
5. Set up the SCbus full duplex routing between the DTI and voice devices using **nr_scroute(FULL DUPLEX)**.
6. Attach voice device and GlobalCall device using **gc_Attach( )**.
7. Repeat steps *2* to *6* for each GlobalCall device.

This is necessary since digital interface timeslot devices and their corresponding voice channel devices are not attached nor routed to each other on the SCbus by default. This is true even for the default devices such as the first trunk timeslot and the first voice channel device for the same physical Span Card hardware.

This peculiar characteristic of earlier-generation Dialogic digital hardware, such as the Span Cards, forces the application to do the proper SCbus routing and attaching of the devices (steps *5* and *6*).

GlobalCall API with DM3 Boards Only (Fixed Routing)

This scenario is one where an R4 for DM3 application uses only DM3 boards.

SCbus routing and attaching of the devices is not necessary with DM3 hardware and the DM3 clustering scheme. The voice device and network interface timeslot that share the same DM3 cluster are permanently routed and attached to each other, and the voice device has no direct access to the CT Bus timeslots.

For the same reason, it is not possible to move audio from a voice device channel to a DTI network interface timeslot device when they do not share the same DM3 cluster. In other words, a **dx_playiottdata( )** command over a specific device handle will result in audio being pumped to its corresponding DTI device that is within the same cluster.

In most cases, R4 for DM3 applications do not need to open the voice channel device and/or the network interface timeslot device directly using the voice and DTI APIs respectively, since it is neither necessary nor possible to attach both devices together from the host.

Furthermore, for the same reasons mentioned above, the **gc_Open( )** device string should not indicate the name of the voice device channel. For example, the device name string for the **gc_Open( )** function should look similar to the following:

```
":N_dtiB1T1:P_ISDN"
```

Although it is unnecessary to specify the ISDN protocol for R4 on DM3 boards, it is specified in this example to retain compatibility with R4 on earlier-generation boards. The **gc_Open( )** function will internally open the associated voice channel.

An R4 for DM3 digital interface application would typically perform the following initialization routine:

NOTE: Use the **sr_getboardcnt( )** function with the class name set to DEV_CLASS_DTI and DEV_CLASS_VOICE to determine the number of network and voice boards in the system, respectively. In Linux, use SRL device mapper functions to return information about the structure of the system. These functions are described in the *Voice Software Reference - Standard Run-time Library*.

1. Start/Initialize GlobalCall with **gc_Start( )**.
2. Open a GlobalCall timeslot device using **gc_Open( )**.

3. Obtain an associated DTI timeslot device handle using **gc_GetNetworkH( )**.
4. Obtain an associated voice channel device handle using **gc_GetVoiceH( )**.
5. Repeat steps *2* to *4* for each GlobalCall timeslot device.

GlobalCall API with DM3 and Earlier-Generation Boards (Fixed Routing)

The following procedure shows how to initialize an application and perform **device discovery** when the application supports both DM3 and earlier-generation boards through the R4 GlobalCall API:

NOTE: Use the **sr_getboardcnt( )** function with the class name set to DEV_CLASS_DTI and DEV_CLASS_VOICE to determine the number of network and voice boards in the system, respectively. In Linux, use SRL device mapper functions to return information about the structure of the system. These functions are described in the *Voice Software Reference - Standard Run-time Library*.

1. Start/Initialize GlobalCall with **gc_Start( )**.
2. Open a GlobalCall timeslot device using **gc_Open( )**.
3. Obtain an associated DTI timeslot device handle using **gc_GetNetworkH( )**.
4. Call **dt_getctinfo( )** and check **CT_DEVINFO.ct_devfamily** value:
    - If **ct_devfamily** is CT_DFDM3, then obtain associated voice channel device handle using **gc_GetVoiceH( )**, and flag all the GlobalCall timeslot devices associated with the trunk as DM3 type.
    - Otherwise, open the standard R4 voice device and obtain a device handle. Attach the standard R4 voice channel device to the GlobalCall timeslot device using **gc_Attach( )**. See *Section GlobalCall API with Earlier-Generation Boards Only* for more information.
5. Repeat steps *2* to *4* for all GlobalCall timeslot devices.

This initialization routine will **not** work well for an application that uses the R4 DTI API for the earlier-generation network interface devices in addition to the R4 for DM3 GlobalCall API for DM3 devices. This is the typical case of a T-1 robbed-bit application developed prior to the availability of the GlobalCall US T-1 protocol.

GlobalCall API with DM3 Boards and DTI API with Earlier-Generation Boards (Fixed Routing)

The following procedure shows how to initialize an application and perform **device discovery** when the application supports both DM3 and earlier-generation boards through the R4 for DM3 GlobalCall API and the R4 DTI API, respectively:

NOTE: Use the **sr_getboardcnt( )** function with the class name set to DEV_CLASS_DTI and DEV_CLASS_VOICE to determine the number of network and voice boards in the system, respectively. In Linux, use SRL device mapper functions to return information about the structure of the system. These functions are described in the *Voice Software Reference - Standard Run-time Library*.

1. Open first DTI timeslot (e.g., dtiB1T1) on the first trunk with **dt_open( )**.
2. Call **dt_getctinfo( )** and check **CT_DEVINFO.ct_devfamily** value.
3. If **ct_devfamily** is CT_DFDM3, then flag all the DTI timeslot devices associated with the trunk as DM3 type.
4. Close DTI device with **dt_close( )**.
5. Repeat steps *1* to *4* for each trunk.
6. Open first voice channel device on the first voice board in the system with **dx_open( )**.
7. Call **dx_getctinfo( )** and check **CT_DEVINFO.ct_devfamily** value.
8. If **ct_devfamily** is CT_DFDM3, then flag all the voice channel devices associated with the board as DM3 type.
9. Close voice channel with **dx_close( )**.
10. Repeat steps *6* to *9* for each voice board.
11. For those voice and DTI devices **not** identified as DM3 devices, proceed with the standard initialization process for earlier-generation boards as performed in the original application. (If using GlobalCall with these earlier-generation boards, follow the initialization in *Section GlobalCall API with Earlier-Generation Boards Only*.)

12. For those voice and DTI devices identified as DM3 devices, proceed with the initialization as described in *Section GlobalCall API with DM3 Boards Only (Fixed Routing)*.

12.5.3. Porting the PANSR Demonstration Program to R4 for DM3 with Fixed Routing

The Dialogic Answering Machine Demonstration program (PANSR) is a sample demonstration program that uses the Dialogic R4 DTI and Voice APIs to simulate the operation of a simple household answering machine. This section describes the steps taken to port the existing PANSR demonstration program so that it can work with both earlier-generation and DM3 devices. The most important considerations when porting the code are included.

The PANSR demonstration program (hereafter referred to as “the demo”) is an MFC GUI application that simulates a simple answering machine. The demo receives a call, plays an introduction prompt, and based on caller response, it either records a message or else plays a previously recorded message to the caller.

The original PANSR demo supports Analog, T-1 E&M robbed-bit, as well as a CAS E-1 protocol, and supports Span Cards and D/4x boards.

Convert Analog Call Control to GlobalCall

The main differences between standard R4 and R4 for DM3 are call control. The following table gives a brief explanation of how to convert an analog network interface call control to use GlobalCall.

Table 20. Porting Analog Call Control to GlobalCall (Fixed Routing)

Call State	R4 Functions	GlobalCall Functions
System initialization	**dx_open( )**	**gc_Start( )** **gc_Open( ) or gc_OpenEx( )** **gc_GetNetworkH( )** **gc_GetVoiceH( )** **Note**: After **gc_Open( )** or **gc_OpenEx( )**, the application must wait for a GCEV_UNBLOCKED event before proceeding with a call (make call or wait call).
Wait for inbound call	Enable CST event mask DE_RINGS by calling **dx_setevtmsk( )** Wait for TDX_CST event	**gc_WaitCall( )** Wait for event GCEV_OFFERED
Answer inbound call	**dx_sethook( )** with DX_OFFHOOK	**gc_AcceptCall( )** **gc_AnswerCall( )**
Drop a call	**dx_sethook( )** with DX_ONHOOK	**gc_DropCall( )** **gc_ReleaseCall( )**
Make outbound call	**dx_sethook( )** with DX_OFFHOOK **dx_dial( )**	**gc_MakeCall( )**
Wait for disconnect	TDX_CST event of DE_LCOFF	GlobalCall event GCEV_DISCONNECTED

Retrieve Board Family Information

The PANSR demo provides support for both earlier-generation and DM3 devices. For earlier-generation devices, the demo works the same as the previous version, but for DM3 devices, GlobalCall is adopted for call control. In order to determine if a device is an earlier-generation device or a DM3 device, the demo retrieves the device family information for all DTI and voice devices and stores the information in a global structure array for later reference.

The following code segment shows how to retrieve the family information for each board.

```
// Get number of DTI boards in system
if( sr_getboardcnt( DEV_CLASS_DTI, &Dti.NumBrdDevs ) == -1 )
{
     MessageBox(hWnd, "Error in retrieving DTI board count", szAppName,
          MB_OK|MB_APPLMODAL);
     return FALSE;
}

if( Dti.NumBrdDevs )
{
     for ( index = 0; index < Dti.NumBrdDevs; index++ )
     {
          sprintf( Dti.BrdNames[ index ], "dtiB%d\0", index + 1 );
          sprintf( dev_name, "dtiB%dT1\0", index + 1 );
          if(( tsdev = dt_open( dev_name, 0 ) ) == -1 )
          {
               sprintf( tmpbuff, "Cannot open channel %s. errno = %d", dev_name, errno
);
               MessageBox( hWnd, tmpbuff, szAppName, MB_OK|MB_APPLMODAL );
               return FALSE;
          }

          // Get Device Information
          if( dt_getctinfo( tsdev, &ct_devinfo ) == -1 )
          {
               sprintf( tmpbuff, "Error message = %s", ATDV_ERRMSGP( tsdev ) );
               MessageBox( hWnd, tmpbuff, szAppName, MB_OK|MB_APPLMODAL );
               dt_close( tsdev );
               return FALSE;
          }

          if( dt_close( tsdev ) == -1 )
          {
               sprintf( tmpbuff, "Cannot close channel %s. errno = %d", dev_name,
errno);
               MessageBox( hWnd, tmpbuff, szAppName, MB_OK|MB_APPLMODAL );
               return FALSE;
          }

          if ( ct_devinfo.ct_devfamily == CT_DFDM3 )
               Dti.frontend[ index ] = CT_NTDM3;
          else
               Dti.frontend[ index ] = ct_devinfo.ct_nettype;
     }

}

// Get number of voice boards in system
if( sr_getboardcnt( DEV_CLASS_VOICE, &( Vox.NumBrdDevs ) ) == -1)
{
     MessageBox( hWnd, "Error retrieving voice board count", szAppName,
          MB_OK|MB_APPLMODAL );
     return FALSE;
```

```
}

// Make board and channel names for VOX devices
if( Vox.NumBrdDevs )
{
     for ( index = 0; index < Vox.NumBrdDevs; index++ )
     {
          sprintf( Vox.BrdNames[ index ], "dxxxB%d\0", index + 1 );
          sprintf( dev_name, "dxxxB%dC1\0", index + 1 );
          if ( ( chdev = dx_open( dev_name, 0 ) ) == -1 )
          {
               sprintf( tmpbuff, "Cannot open channel %s. errno = %d", dev_name, errno
);
               MessageBox( hWnd, tmpbuff, szAppName, MB_OK|MB_APPLMODAL );
               return FALSE;
          }

          // Get Device Information
          if( dx_getctinfo( chdev, &ct_devinfo ) == -1 )
          {
               sprintf( tmpbuff, "Error message = %s", ATDV_ERRMSGP( chdev ) );
               MessageBox( hWnd, tmpbuff, szAppName, MB_OK|MB_APPLMODAL );
               dx_close( chdev );
               return FALSE;
          }

          if( dx_close( chdev ) == -1 )
          {
               sprintf( tmpbuff, "Cannot close channel %s. errno = %d", dev_name,
errno);
               MessageBox( hWnd, tmpbuff, szAppName, MB_OK|MB_APPLMODAL );
               return FALSE;
          }

          if( ct_devinfo.ct_devfamily == CT_DFDM3 )
               Vox.frontend[ index ] = CT_NTDM3;
          else
               Vox.frontend[ index ] = CT_NTANALOG;
     }
}
```

Based on the family information, the application can be customized to take advantage of the features for the type of board.

Initialize GlobalCall

When the PANSR demo is executed on a system containing DM3 devices, GlobalCall must be initialized before opening the devices. If the PANSR demo is executed on a system containing earlier-generation devices, this step is skipped.

The following code segment shows how to initialize GlobalCall:

```
if( frontend == CT_NTDM3 )
{
     if( gc_Start( NULL ) != GC_SUCCESS )
```

```
     {
          gc_ErrorValue( &gc_error, &cclibid, &cc_error);
          gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
          sprintf( tmpbuff, "Error in gc_Start ErrorValue: %d - %s\n", gc_error, msg);
          disp_msg( tmpbuff );
          QUIT( 2 );
     }
}
```

Open Devices

For earlier-generation devices, the PANSR demo opens voice devices and DTI devices with **dx_open( )** and **dt_open( )**. If SCbus routing was selected as an option at the beginning of the demo, the demo calls the SCbus routing function **nr_scroute( )** to route voice resources to DTI devices.

NOTE: If the PANSR demo finds DM3 devices, then you must initialize GlobalCall before opening those devices.

If the PANSR demo is executed on a system containing DM3 devices, the demo uses GlobalCall **gc_Open( )** with the device naming convention ":N_dtiBxTx:P_protocol" instead of **dx_open( )** and **dt_open( )** to open GlobalCall device handles. The demo calls **gc_GetVoiceH( )** to retrieve the associated voice handle and stores it in a channel-specific structure for later reference. The protocol name is retrieved from the internal protocol setting which the developer specified in application settings. Because SCbus routing of voice resources is not supported in the DM3 fixed routing configuration, the SCbus routing procedure for DM3 devices is skipped.

The following segment shows how to open GlobalCall devices and retrieve associated voice devices.

```
// Open GlobalCall device
sprintf( dxinfo[ channum ].ldevname, ":N_dtiB%dT%d:P_ISDN", dtibdnum, channum );
if( gc_Open( &dxinfo[ channum ].ldev, dxinfo[ channum ].ldevname, 0 ) == -1 )
{
     gc_ErrorValue( &gc_error, &cclibid, &cc_error);
     gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
     sprintf( tmpbuff, "Unable to open DM3 %s\n%s", dxinfo[ channum ].ldevname,msg );
     disp_msg( tmpbuff );
QUIT( 2 );
}

// Retrieve the associated voice device
gc_GetVoiceH( dxinfo[ channum ].ldev, &dxinfo[ channum ].chdev);
```

Process SRL Events

The SRL event processing routine waits for any SRL event by calling **sr_waitevt( )**. For systems containing DM3 devices, the PANSR demo calls **gc_GetMetaEvent( )** to process GlobalCall-specific events. Otherwise, **sr_getevtdev( )** and **sr_getevtdatap( )** are used to retrieve the appropriate information.

The structure METAEVENT, filled by **gc_GetMetaEvent( )**, contains all necessary event data for event processing, including the SRL event type, event handle and additional event data.

The following code segment shows how to process SRL events:

```
// Wait for SRL event
if( sr_waitevt( timeout ) != -1 )
{
     if( frontend == CT_NTDM3 )
     {
          // DM3 devices
          rc = gc_GetMetaEvent( &MetaEvent );
          chtsdev = Metaevent.evtdev;
     }
     else
     {
          // earlier-generation devices
          chtsdev = sr_getevtdev();
     }

     ...
     // Further event process
     ...
}
```

Wait for Incoming Call

For earlier-generation devices, the PANSR demo relies on various interface-specific events to recognize an incoming call. On boards with an analog network interface, the demo waits for a TDX_CST event with DE_RINGS **cst_event** data. On boards with a digital network interface, the demo waits for changes in the E-1 or T-1 digital signal.

For DM3 devices, the demo calls **gc_WaitCall( )** to initialize a “wait for incoming call” state. An incoming call is indicated by a GlobalCall event GCEV_OFFERED.

The following code segment shows how to initialize call waiting in asynchronous mode:

```
if( gc_WaitCall(dxinfo[ channum ].ldev, NULL, NULL, 0, EV_ASYNC) != GC_SUCCESS)
{
     gc_ErrorValue( &gc_error, &cclibid, &cc_error);
     gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
     sprintf( tmpbuff, "gc_WaitCall Error%s\n%s", lineDevName,msg );
     disp_msg( tmpbuff );
     QUIT( 2 );
}

dxinfo[ channum ].state = ST_WTRING;
```

The incoming call is reported by the GlobalCall event GCEV_OFFERED.

Answer Inbound Call

On boards with an analog network interface, the demo calls **dx_sethook( )** to go off hook. On earlier-generation boards with a digital network interface, the demo calls **dt_settssigsim( )** to set digital signals indicating that the incoming call is answered.

For DM3 devices, the demo does the following:

1. Calls **gc_GetMetaEvent( )** to fill the METAEVENT structure with data.
2. Calls **gc_GetCRN( )** to retrieve the CRN (Call Reference Number).
3. Optionally, calls **gc_AcceptCall( )** to accept the incoming call, waits for the GlobalCall event GCEV_ACCEPT to indicate the completion of call acceptance. This indicates that the line is ringing.
4. Calls **gc_AnswerCall( )** to answer the incoming call. The completion of call answering is indicated by GCEV_ANSWERED.

The following code segment shows how to answer an incoming call in asynchronous mode:

```
rc = gc_GetMetaEvent( &dxinfo[ channum ].ch_Metaevent);
if( rc != GC_SUCCESS)
{
      gc_ErrorValue( &gc_error, &cclibid, &cc_error);
     gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
     sprintf( tmpbuff, " gc_GetMetaEvent Error%s\n%s", lineDevName,msg );
     disp_msg( tmpbuff );
     QUIT( 2 );
}
```

```
rc = gc_GetCRN( &dxinfo[ channum ].crn, &dxinfo[ channum ].ch_Metaevent );
if( rc != GC_SUCCESS)
{
     gc_ErrorValue( &gc_error, &cclibid, &cc_error);
     gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
     sprintf( tmpbuff, "gc_GetCRN Error%s\n%s", lineDevName,msg );
     disp_msg( tmpbuff );
QUIT( 2 );
}

rc = gc_AnswerCall( dxinfo[ channum ].crn, 0, EV_ASYNC);
if( rc != GC_SUCCESS)
{
     gc_ErrorValue( &gc_error, &cclibid, &cc_error);
     gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
     sprintf( tmpbuff, "gc_AnswerCall Error%s\n%s", lineDevName,msg );
     disp_msg( tmpbuff );
     QUIT( 2 );
}
```

Monitor Call Disconnection

After the call is answered, the call may be disconnected at any time. For earlier-generation devices, depending upon the network interface of the device, when a call is disconnected, the PANSR demo is notified. On boards with an analog network interface, the notification is a TDX_CST event in which the **cst_event** is DE_LCOFF. On earlier-generation boards with a digital network interface, the notification is through digital signaling.

For DM3 devices, the call disconnection is notified by the GlobalCall event GCEV_DISCONNECTED.

Drop Call

For earlier-generation devices, the PANSR demo drops a call in according to the network interface of the device. On boards with an analog network interface, the demo calls **dx_sethook( )** to go on hook. On boards with a digital network interface, the demo calls **dt_settssigsim( )** to transmit digital signals to drop the call.

For DM3 devices, the demo calls **gc_DropCall( )** to drop the call. The completion of call dropping is indicated by the GlobalCall event GCEV_DROPCALL. After the call is dropped, the demo calls **gc_ReleaseCall( )** to release the call completely.

The following code segment shows how to receive a call disconnect event and drop the call in asynchronous mode.

```
switch( event )
{
     ...
case GCEV_DISCONNECTED :
if( gc_DropCall( dxinfo[ channum ].crn, GC_NORMAL_CLEARING, EV_ASYNC )!= 0)
     {
          gc_ErrorValue( &gc_error, &cclibid, &cc_error);
          gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
          sprintf( tmpbuff, "The linedev %s call cannot be dropped!\nERROR: %s",
               lineDevName, msg );
          disp_msg( tmpbuff );
          QUIT( 2 );
     }
     break;

case GCEV_DROPCALL:
     if( gc_ReleaseCall( dxinfo[ channum ].crn ) != 0 )
     {
          gc_ErrorValue( &gc_error, &cclibid, &cc_error);
          gc_ResultMsg( LIBID_GC, (long) gc_error, &msg);
          sprintf( tmpbuff, "The linedev %s call cannot be released!\nERROR: %s",
               lineDevName, msg );
          disp_msg( tmpbuff );
          QUIT( 2 );
     }
     break;
     ...
}
```

Clean-up

In application clean-up, the PANSR demo closes all opened devices. If GlobalCall was started, the demo calls **gc_Stop( )** to stop GlobalCall.

Make Outbound Call

This is a topic not covered in the implementation of the PANSR demo, but is presented here to give information about how to make outbound calls using GlobalCall for DM3 devices.

For earlier-generation devices, analog network interface devices use **dx_sethook( )** and **dx_dial( )** with or without call analysis to make an outbound call, while digital network interface devices use **dt_settssigsim( )** and corresponding call processing signaling to make an outbound call.

For DM3 devices, the program should use **gc_MakeCall( )** to make a call. After a call is established or has failed to connect, the program will be informed by the GlobalCall events such as GCEV_CONNECTED, GCEV_ALERTING, or GCEV_DISCONNECTED.

Appendix A

CPA Default Tone Templates

The following are Call Progress Analysis (CPA) default tone templates for DM3. SpringWare tone IDs are listed for reference. Amplitudes are given in dBm, frequencies in Hz, and duration in 10 ms units.

NOTE: The Dm3_DIALTONE2 name must be specified as shown, with the mix of uppercase and lowercase characters. A dash in a table cell means not applicable. A space in a table cell means not available on DM3 boards.

Table 21. Default Call Progress Analysis Tone Definitions

SpringWare Tone ID	DM3 ID	Freq1	Freq2	On Time	Off Time	Reps
TID_BUSY1	BUSY	480 ± 30	620 ± 30	650 ± 350	650 ± 350	2
TID_BUSY2	BUSY_FAST	480 ± 30	620 ± 30	250 ± 150	250 ± 150	2
TID_DIAL_LCL	DIALTONE1	350 ± 40	440 ± 40	100	-	1
TID_DIAL_INTL	Dm3_DIALTONE2	340 ± 40	440 ± 40	100	-	1
TID_DIAL_XTRA						
TID_DISCONNECT						
TID_FAX1	FAX_CNG1	1100 ± 50		350 ± 250	-	1
TID_FAX2						
TID_RNGBK1	RINGING1	450 ± 150	450 ± 150	1875 ± 1125	4000 ± 4000	1
TID_RNGBK2	RINGING2 TS0	450 ± 150	450 ± 150	600 ± 400	600 ± 400	1
	RINGING2 TS1	450 ± 150	450 ± 150	600 ± 400	3500 ± 2500	1

Index

Index

B

C

D

Index

Index

H

I

Index

R

S

NOTES

NOTES

NOTES

NOTES