Dialogic®
Making Innovation Thrive™

Application Note

Migrating an Application from Media Boards with Dialogic® Springware Architecture to Media Products with Dialogic® DM3 Architecture

Migrating an Application from Media Boards with Dialogic® Springware Architecture to Media Products with Dialogic® DM3 Architecture

Application Note

Executive Summary

This document provides architectural guidelines for migrating an existing application written for media boards with Dialogic® Springware Architecture to media products with Dialogic® DM3 architecture. It also aids in the initial design of an application for a system using boards with DM3 architecture, and serves as a reference for finding DM3-architecture-related information that can be used in the development process.

Migrating an Application from Media Boards with Dialogic® Springware Architecture to Media Products with Dialogic® DM3 Architecture

Application Note

Table of Contents

Migrating an Application from Media Boards with Dialogic® Springware Architecture to Media Products with Dialogic® DM3 Architecture

Application Note

Introduction

This application note provides architectural guidelines for migrating an existing application written for media boards with Dialogic® Springware Architecture to one using media products with Dialogic® DM3 Architecture. It does not fully cover all aspects of using and programming products with DM3 architecture, but rather focuses on areas to consider when migrating to DM3 architecture. The information provided here applies to telephony applications that run under the Windows® or Linux operating systems.

Important Note Regarding Terminology: For simplicity and clarity, "DM3" is used in this application note to refer to Dialogic® DM3 Architecture, and "Springware" is used to refer to Dialogic® Springware Architecture. Dialogic® JCT Boards use the Springware architecture. References to "DM3" and "DM3 architecture" are equally applicable to APIs for media boards with DM3 architecture and APIs for Dialogic® Host Media Processing (HMP) Software because the APIs for both are enabled through the same libraries and drivers. An application using a media board with DM3 architecture should not be noticeably different from an application using Dialogic HMP Software.

Dialogic® DM3 Architecture Overview

Figure 1 shows a high-level view of the host-based DM3 software stack, from the operating system drivers up to the telephony application itself. This diagram and the definition of the layers that follow are presented as background information, because an application's only access to media functionality on DM3 is through the Dialogic® R4 API, and its only access to DM3 call control is through the Dialogic® Global Call API.

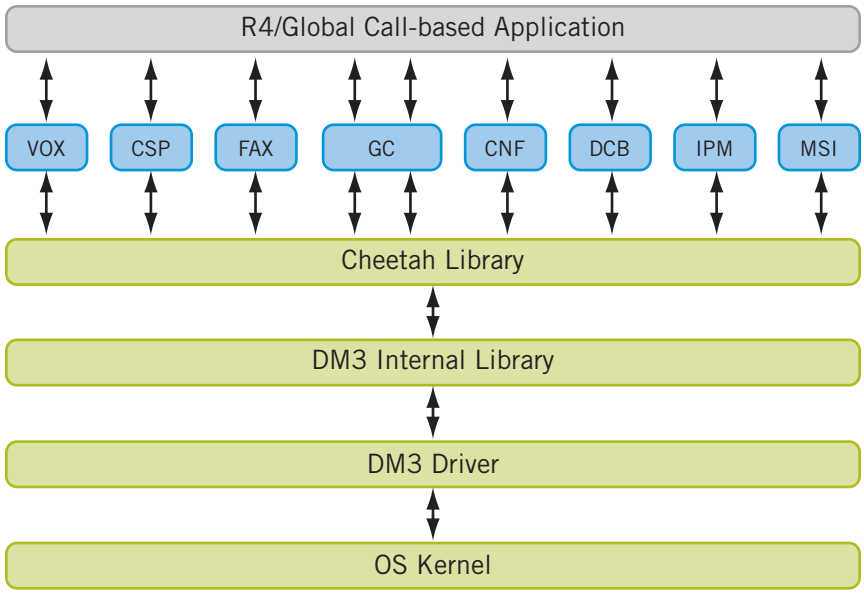

Figure 1. Dialogic® DM3 Architecture Diagram

The layers in the DM3 Architecture Diagram are described below:

- **R4/Global Call-Based application** — The user-written telephony application that contains Dialogic® API calls to DM3 hardware.
- **VOX, CSP, FAX, GC, CNF, DCB, IPM, and MSI** — R4 API module groupings that provide the highest level of R4 to DM3 translation.
- **Cheetah Library** — The layer containing translations from R4 to the native low-level messages used with Windows® or Linux DM3 implementations. Because the Cheetah layer abstracts the internal native DM3 API, you can use one unified API (R4/Global Call) to control both Springware-based Dialogic® media boards and DM3-based media products.
- **DM3 Driver** — An OS-level device driver that controls the DM3 PCI/cPCI hardware.
- **OS Kernel** — Windows or Linux kernel-level constructs that underlie the DM3 device drivers.

Application Architectures

Successful application architectures under DM3 are more restrictive than architectures that work well with Springware. Older, non-multithreaded operating systems were prevalent when Springware was first designed. Therefore, Springware allows the ability of multiple processes to access the same channel on a board.

Under DM3, multi-process device sharing is generally not allowed or recommended. There are documented exceptions, however, if the application architecture requires a multiprocessor approach. For information, refer to the *Dialogic® Voice API Programming Guide*.

While this type of restriction may at first seem burdensome, it can lead to a more efficient application that can handle the higher channel densities possible in a DM3 system. Newer multithreaded operating systems and the DM3 architecture itself require a different approach than with Springware, leading to an architecture that is easier to build and maintain, and that results in a lower cost of ownership. For example:

- An asynchronous architecture leads to the use of state machines for application and device logic. This is more desirable and maintainable in the long run than logic implemented using the nested switch statements that are often found with synchronous programming.
- With a multithreaded application, much of the inter-process communications needed to move data between processes and coordinate uncoupled logic is eliminated. Relying on an operating system's optimized thread scheduling is more efficient than relying on process scheduling.

Programming Models

Although the subject of programming models has been covered in Dialogic® System Release documentation and elsewhere, it is described here as it relates to DM3.

Synchronous Programming Model

Using Dialogic® API calls in a synchronous mode blocks the process or thread execution until the function completes. For a lengthy operation such as a file play, blocking can occur for many seconds, preventing the process/thread from doing useful work for that time. This means that an architecture that handles multiple simultaneous channels becomes difficult to use, and leads to one where each device requires a separate application or process/thread. As individual processes/threads sleep, the operating system continues with other unblocked processes/threads. When the synchronous function completes, the process/thread wakes up and processing continues.

While a synchronous approach neatly segments each channel into a single process/thread, it can be very inefficient. This approach consumes significant system resources for process/thread scheduling and context switching, making it unsuitable for a high density DM3 system. For example, E1 configurations running R2MF have experienced performance issues when the application uses a single process per channel for 60 channels. These issues can prevent an application from reaching a common application density goal of 8 T1 spans (192 channels).

Tip: Use the synchronous programming model only for low-density demos or very simple applications that are not intended for deployment.

Asynchronous Programming Model

When Dialogic API calls are made in an asynchronous mode, the process/thread continues while the API function completes. There is a short delay (on the order of milliseconds) before the API call returns and the process/thread can continue executing. The short delay confirms that execution started. When the operation started by the API call finishes, the application receives the completion event asynchronously.

Asynchronous operation allows multiple channels to be easily handled by a single application process/thread, and it moves the application toward an event-driven architecture that requires the use of a state machine. This reduces the system overhead required for inter-process communication and thread scheduling, which results in the ability to have higher channel densities. Because the DM3 architecture is inherently asynchronous by design, asynchronous application architecture fits well with the way that the DM3 media boards and software function.

An asynchronous programming model is either a polled model or a callback model. In the polled model, the application polls (waits) for events using an event loop to retrieve and dispose of events. In the callback model, the SRL layer maintains the event loop through the use of callback functions for specific events on specific devices and event handlers. While the polled model allows a little more flexibility in specifying polling timeouts, the callback model is easier to use.

Extended Asynchronous Programming Model

The extended asynchronous programming model is similar to the asynchronous model, except that in this model, you can also:

- Assign different groups of devices to different event handlers.
- Create multiple threads, each of which can control multiple devices.

For a more in-depth discussion of programming models and their performance characteristics, see the *Dialogic® Standard Runtime Library API Programming Guide* or the *Dialogic® Global Call API Programming Guide*.

Programming Model for Dialogic® DM3 Architecture

A DM3 application should be a single process that uses the asynchronous (ASYNC) model and that limits the use of threads when possible. Since both callback and polled ASYNC models are acceptable, application design using either model should work equally well.

Overview of the Dialogic® Global Call API for Call Control

The Global Call API is a call control API that uses the Dialogic® Standard Runtime Library (SRL) API to deliver response events to its API commands. The Global Call API and other Dialogic® APIs form a family of APIs that use the underlying services provided by the SRL API.

Migrating an Application from Media Boards with Dialogic® Springware Architecture to Media Products with Dialogic® DM3 Architecture

The Global Call API provides a collection of functions that support call control operations, operation tasks, administration tasks, and maintenance tasks. It attempts to unify call control into a unified state machine, which allows for code to be portable between technologies and products. For information about Global Call API functions, refer to the *Dialogic® Global Call API Library Reference*.

Tip: DM3 uses the Global Call API for call control. If your application uses Analog (dx_), CAS /Rob Bit (dt_), or ISDN (cc_) libs, you must port your application to Global Control for call control.

Figures 2 and 3 show the Global Call API high level state machine diagrams for inbound and outbound calls. For more information, refer to the *Dialogic® Global Call API Programming Guide and the Global Call technology guides.*

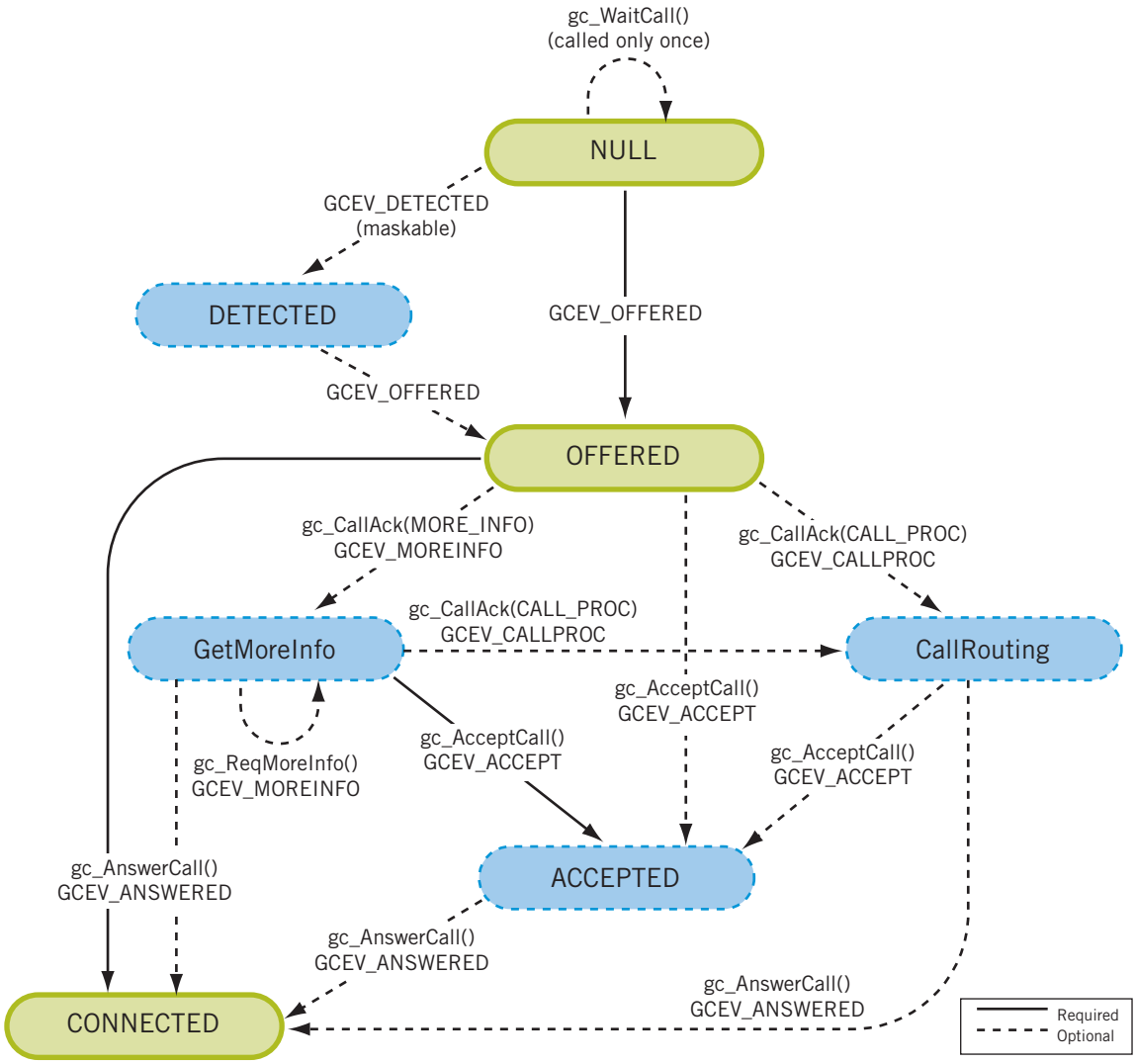

Figure 2. Inbound Call State Machine Diagram

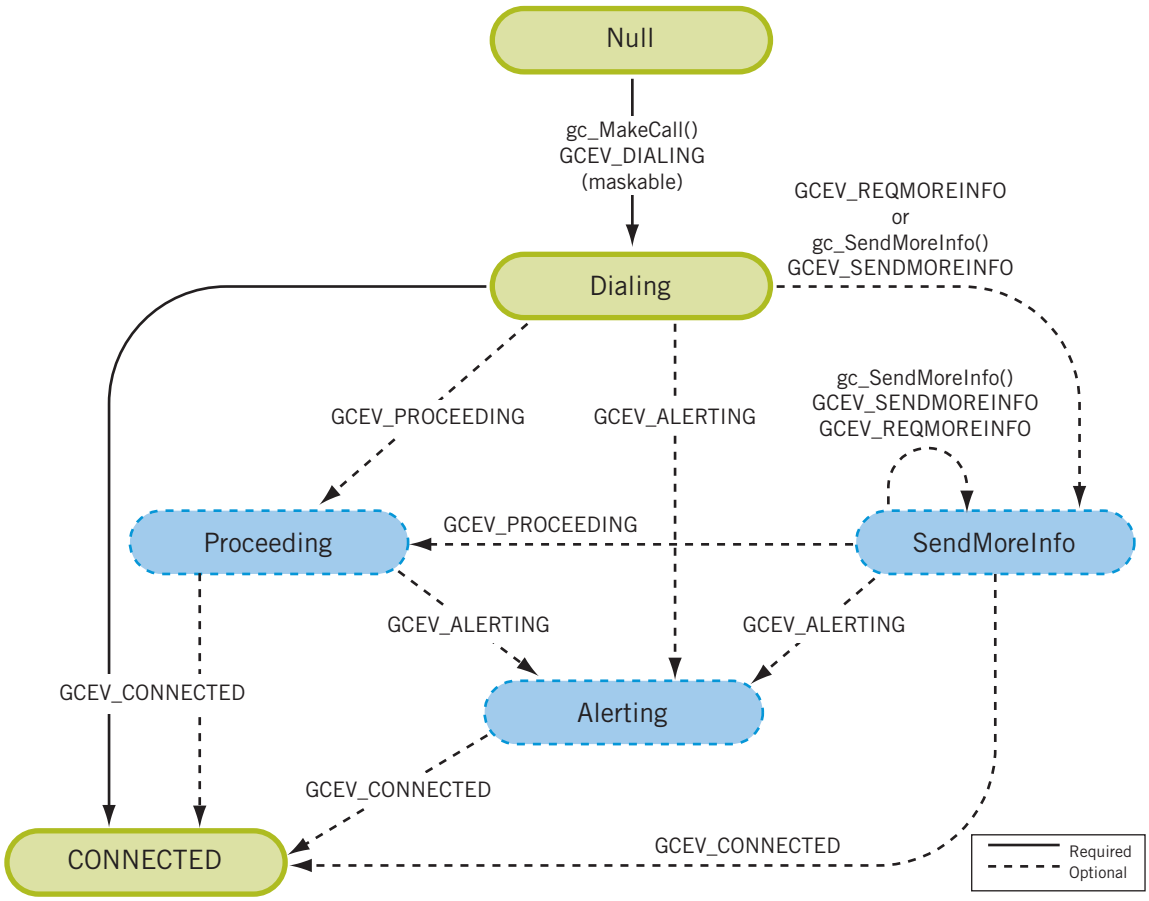

Figure 3. Outbound Call Call State Machine Diagram

Migrating an Application to the Dialogic® Global Call API

The Dialogic® Global Call API Programming Guide provides in-depth information and expands on several of the topics in this application note. It can be used as a reference when migrating an application to the Global Call API.

Initializing Call Control Libraries

Each supported network technology requires a call control library to provide the interface between the network and the Global Call library.

Call control libraries must be started before they can be used by the Global Call functions. To start a call control library, call the gc_Start() function. This function allows the selective starting of call control libraries, so that you can specify whether to start all call control libraries or only specified libraries. You can also start a custom call control library that is not supported by Global Call. For more information about the gc_Start() function, refer to the *Dialogic® Global Call API Library Reference*.

Tip: When first working with gc_Start() and the corresponding start structure, it is helpful to read the documentation and review the sample demos before using gc_Start.

Dialogic® Global Call API Objects

The Global Call API is call-oriented - that is, each call initiated by the application or network is assigned a Call Reference Number (CRN) for call control and tracking purposes. Call handling is independent of the line device over which the call is routed. Each line device or device group is assigned a Line Device Identifier (LDID) that enables the application to address a resource or group of resources using a single device identifier.

Line Device Identifier

A Line Device Identifier (LDID) is a unique logical number assigned to a specific resource (for example, a network device) or group of resources within a Global Call library process. When an application calls the gc_OpenEx() function, an LDID number is assigned to represent one or more physical or logical devices that will handle a call. This identification number assignment remains valid until the application calls the gc_Close() function to close the line device.

When an event arrives, the application can use the gc_GetMetaEvent() or gc_GetMetaEventEx() function to retrieve the LDID number associated with the event function. The Global Call API places the LDID value in the linedev field of the METAEVENT structure.

Tip: Formatting an open string for the gcOpenEx() function is an important part of correctly initializing the Global Call system. The contents of this string are based on the technology used. For more information, refer to the *Dialogic® Global Call API Library Reference* and the technology guide for the technology type being used. Reviewing the sample demos can also help in understanding the gcOpenEx() fuction.

Call Reference Numbers

A Call Reference Number (CRN) identifies a call on a specific line device. The Global Call library creates a CRN when a call is requested by a process, thread, or network. With the CRN approach, a process or thread can access and control the call without referring to a specific physical port or line device. CRNs are assigned to both inbound and outbound calls:

- For inbound calls, the gc_WaitCall() function assigns the CRN.
- For outbound calls, the the gc_MakeCall() or gc_SetUpTransfer() function assigns the CRN.

The CRN has a single LDID associated with it; for example, an LDID that represents the line device on which the call was made. Because multiple calls can exist on a single line, an LDID can be associated with multiple CRNs, up to a maximum of 20. At a given instant, each CRN is a unique number within a process. After a call is terminated and the application calls the gc_ReleaseCallEx() function to release the resources used for the call, the CRN is no longer valid.

For more information about CRNs, see the *Dialogic® Global Call API Library Reference*.

Using METAEVENT

Use the gc_GetMetaEvent() function and the METAEVENT structure to replace SRL API event retrieval functions (sr_getevttype, sr_getevtdatap, and so forth). You can optionally use the METAEVENT structure for processing all events instead of just the older SRL API functions.

Migrating from the Dialogic® Digital Network Interface API to the Dialogic® Global Call API

The Digital Network Interface (DTI) API contains several functions that allow the implementation of custom protocols or the viewing and modifying of the lower level signaling bits of a CAS/Rob Bit protocol. In addition, the DTI API provides Time Division Multiplex (TDM) bus routing, network interface alarms, and time slot signaling control.

The Global Call API presents a higher level of abstraction at the API level, hiding the internals of the DTI specific protocol. For example, to use the DTI API to make a call under a T1 robbed-bit trunk, you must flip the A & B signaling bits. To do the same under an ISDN PRI protocol, a specific HDLC packet must be sent over the ISDN data channel. The Global Call API hides these complicated operations, and a Global Call application can be designed so that the same application can run with an E1 CAS trunk or an E1 ISDN trunk without requiring changes.

The Global Call API is well suited for migrated DM3 applications because it:

- Presents a high level of abstraction for flexible digital interface telephony application deployment.
- Provides a unified interface across product and technology lines, allowing application vendors to maintain a single code base for multiple technologies such as robbed-bit, ISDN, analog, and IP.

One of the challenges of migrating an application that uses Springware and the DTI API is the lack of support for much of the DTI API functionality when using DM3. The Global Call API more than makes up for this shortcoming and simplifies application development by providing a level of abstraction that allows seamless support for telephony interfaces, including T1, E1, ISDN, or analog. Once an application is designed to use the Global Call API, minimal changes (if any) are required for the same application to run on various Dialogic® products based on Springware and DM3.

Call States

The Global Call API consolidates the bit/tone definitions for each call state into a set of universal events. Upon receiving these events, the application must make the appropriate Global Call API function call for each state (Answer, Drop, and so forth) Therefore, the application needs to maintain just the high level call state, and it should monitor the Global Call state events rather than the DTEV_SIGEVENT.

A useful starting point for migrating to the Global Call API is to lay out each of the call states and bit diagrams, and then use Figures 2 and 3 to map Global Call API functions and events to the current protocol. For example, to end a call, the original application receives a DTEV_SIGEVENT with AOFF + BOFF. Then it calls dt_settssig(AOFF+BOFF). In the new Global Call API application, the application receives a GCEV_DISCONNECTED event when a call is dropped. Then it drops the call by issuing a gc_DropCall().

Once the actual state machines are mapped out, start to implement the new Global Call API state machines as described in the *Dialogic® Global Call API Programming Guide*.

Call Reference Number

When migrating from the DTI API, keep in mind that the Global Call API uses a Call Reference Number (CRN) to track each of the calls. A CRN is what a majority of the functions take as an argument, rather than just the device handle. Because the CRN is unique for each call, use a map to maintain an association between the CRN and the device handle.

Tip: The sr_setparm() function should be used to store the CRN in the USERCONTEXT location. For more information, see the *Dialogic® Standard Runtime Library API Library Reference*.

Release each CRN by issuing a gc_ReleaseCallEx(crn) at the end of each call, usually after the application receives a GCEV_DROPCALL event.

Migrating from the Dialogic® ISDN API to the Dialogic® Global Call API

Many existing applications make use of the rich Dialogic® ISDN API. This API has evolved over time, and provides access to two levels of abstraction, known as layer 3 and layer 2. The ISDN API is not supported when using DM3; however, much of its layer 3 functionality can be achieved at a similar level of abstraction using the Global Call API.

The migration of an ISDN library-based application to a Global Call API-based application is often easy because the states selected for the Global Call API were originally based on the Q.931 protocol. Because this is the same specification that ISDN uses, the two APIs share a large amount of similarity.

Tip: In most cases, you can successfully migrate an application from the ISDN API to the Global Call API by replacing cc_ API calls with gc_ equivalents and changing CCEV_ processing to GCEV_ events.

Migrating from an Existing Voice Library to Dialogic® DM3

Springware-based applications and DM3-based applications use the same voice library for their voice processing needs. However, there are slight variations in the implementations of these libraries. The following table discusses functions that, as of the publication of this note, behave slightly differently in the Springware and DM3 architectures:

Functions	Notes
ATDX_ANSRSIZ()	Not supported in DM3.
ATDX_BUFDIGS()	Supported in DM3, but must be manually enabled. Enable this function as follows before the application is loaded in memory: • In Linux, add SupportForSignalCounting = 1 in /usr/dialogic/cfg/cheetah.cfg. To subsequently disable this function, remove this line from the .cfg file. • In Windows®, set the SupportForSignalCounting parameter to 1 in Key HKEY_LOCAL_MACHINE\SOFTWARE\Dialogic\Cheetah\CC. To subsequently disable this function, set this parameter to 0.
CPA Helper functions: • ATDX_DTNFAIL() • ATDX_FRQDUR() • ATDX_FRQDUR2() • ATDX_FRQDUR3() • ATDX_FRQHZ() • ATDX_FRQHZ2() • ATDX_FRQHZ3() • ATDX_FRQOUT() • ATDX_LONGLOW() • ATDX_SHORTLOW() • ATDX_SIZEHI() • dx_chgdur() • dx_chgfreq() • dx_chgrepcnt()	Not supported in DM3.
ATDX_FWVER()	Not supported in DM3.
ATDX_HOOKST()	Not supported in DM3.
ATDX_LINEST()	Not supported in DM3.

Functions	Notes
ATDX_PHYADDR()	Not supported in DM3.
dx_addspddig() dx_addvoldig() dx_setsvcond()	In DM3, digits used for play adjustment can also be used as a terminating condition. If a digit is defined for both actions, both actions are applied upon detection of that digit. In Springware, digits used for play adjustment are not used as a terminating condition. If a digit is defined for both actions, the play adjustment takes priority.
dx_clrdigbuf()	In DM3, digits are not always cleared by the time the dx_clrdigbuf() function returns. For this reason, give careful consideration when using this function before or in a code section that requires digit detection or digit termination. The digits can only be cleared after the function has returned; possibly during the next function call.
dx_dial()	The ',' for a pause character is set to 2.5 seconds on DM3. In addition, DM3 does not support '&', 'P', 'L','I','X' as a character in the dialstring.
dx_gtcallid() dx_gttextcallid() dx_wtcallid()	Not supported in DM3.
dx_initcallp()	In DM3, call progress analysis is enabled directly through the dx_dial() function. Therefore, there is no need to explicitly call dx_initcallp().
dx_playtone(), dx_playtoneEx	In DM3, the DX_MAXTIME termination condition is not supported by tone generation functions.
dx_reciottdata() dx_recvox() dx_recwav()	In DM3, voice channels must be listening to a TDM bus time slot in order for voice recording functions to work. Therefore, you must call the dx_listen() function call on the device handle before calling a voice recording function for that device handle. If this is not done, the voice channel remains in a suspended state and can be cleared only by calling dx_stopch() or dx_listen(). The actual recording operation starts only after the voice channel is listening to the proper external time slot.
dk_sendevt()	Not supported in DM3.
dx_setdevuio() dx_setuio()	In DM3, user-defined I/O functions installed by dx_setdevuio() are called in a different thread than the main application thread. If data is being shared among these threads, the application must carefully protect access to this data using appropriate synchronization mechanisms (such as mutex) to ensure data integrity.
dx_setdigbuf()	Not supported in DM3.
dx_sethook	Not supported in DM3.
dx_setparm()	Refer to the *Dialogic® Voice API Library Reference* for a list of parameters that are supported in each technology.
dx_setsvmt()	In DM3, if an application calls dx_close() to close a device after calling dx_setsvmt() to modify speed and value tables, the dx_getcursv() function might return incorrect speed and volume settings for the device. This is because the next dx_open() resets the speed and volume tables to their default values. Therefore, do not issue a dx_close() after a making a function call that modifies speed and volume table values.
dx_wink	Not supported in DM3.
dx_wtring	Not supported in DM3.

Table 1. Functions that behave differently in the Springware and DM3 architectures

Tip: If a function is not working as expected, refer to the *Dialogic® Voice API Library Reference* to check that the function is supported and has no cautions.

Migrating from other Libraries to Dialogic® DM3

DM3 supports many of the other commonly used programming libraries that are supported on Springware (Fax, CSP, DCB). Refer to the programming guides and library references for each of these APIs for information about differences among the various platforms.

Tip: Because Dialogic HMP Software only supports DM3, Dialogic HMP Software documentation does not provide comparisons to Springware. Therefore, it can be useful to look up differences among APIs in documentation written for releases that support both Springware and DM3.

For More Information

Voice

Dialogic® Voice API Library Reference —
http://www.dialogic.com/manuals/docs/voice_api_hmp_v5.pdf
Provides reference information for building computer telephony applications that require call processing functionality, such as voice recording and playback, voice encoding, tone detection and signaling, and much more.

Dialogic® Voice API Programming Guide —
http://www.dialogic.com/manuals/docs/voice_programming_hmp_v5.pdf
Provides instructions for developing applications on Linux and Windows® operating systems using the Dialogic® Voice API that is supplied with the Dialogic HMP Software.

Call Control

Digital Network Interface Software Reference for Linux and Windows —
http://www.dialogic.com/manuals/docs/digital_network_api_v5.pdf
Provides an overview of the digital telephony interface (DTI) and a DTI function reference.

Dialogic® Global Call API Programming Guide —
http://www.dialogic.com/manuals/docs/globalcall_programming_hmp_v4.pdf
Provides guidelines for building computer telephony applications that require call control functionality. Such applications include, but are not limited to, call routing, enhanced services, unified messaging, voice messaging, LAN telephony services, computer telephony services, switching, PBX, interactive voice response, help desk and work flow applications.

Dialogic® Global Call API Library Reference —
http://www.dialogic.com/manuals/docs/globalcall_api_v8.pdf
Provides reference information for building computer telephony applications that require call control functionality.

Global Call E1/T1 CAS/R2 Technology Guide —
http://www.dialogic.com/manuals/docs/globalcall_for_e1t1_v1.pdf
Provides information about developing Global Call applications that use E1 CAS or T1 robbed bit technology.

Dialogic® Global Call IP Technology Guide —
http://www.dialogic.com/manuals/docs/globalcall_for_ip_hmp_v10.pdf
Provides information about developing Global Call applications that implement host-based H.323/SIP call control.

Global Call ISDN Technology Guide —
http://www.dialogic.com/manuals/docs/globalcall_for_isdn_v7.pdf
Provides information about developing Global Call applications that use ISDN technology.

Dialogic® Global Call SS7 Technology Guide —
http://www.dialogic.com/manuals/docs/globalcall_for_ss7_v5.pdf
Provides information about developing Global Call applications that use SS7 technology.

Standard Runtime Library

Dialogic® Standard Runtime Library API, Programming Guide —
http://www.dialogic.com/manuals/docs/srl_programming_win_v5.pdf
Provides guidelines for using the Standard Runtime Library (SRL) API when developing computer telephony applications. The SRL API provides a common interface for event handling and other functionality common to all devices, such as network interface, voice, and fax devices.

Dialogic® Standard Runtime Library API, Library Reference —
http://www.dialogic.com/manuals/docs/srl_api_v5.pdf
Provides reference information for the Standard Runtime Library (SRL) API when developing computer telephony applications.

Installation and Configuration

Dialogic® Global Call Country Dependent Parameters (CDP) for PDK Protocol, Configuration Guide —
http://www.dialogic.com/manuals/docs/globalcall_cdp_config_v8.pdf
Provides detailed information about configuring the Country Dependent Parameters (CDP) files included in the Global Call Protocols package. Configuration procedures are included, as well as a description of each configuration parameter.

Dialogic® Host Media Processing Software Release 3.0WIN, Software Installation Guide —
http://www.dialogic.com/manuals/hmp30win/release_install.pdf
Explains how to install and uninstall this release.

Dialogic® Host Media Processing Software for Windows Configuration Guide —
http://www.dialogic.com/manuals/docs/config_hmp_win_v1.pdf
Provides information for configuring Dialogic HMP Software and digital network interface boards in a Windows® system.

Dialogic®
Making Innovation Thrive™

www.dialogic.com

Dialogic Corporation
9800 Cavendish Blvd., 5th floor
Montreal, Quebec
CANADA H4M 2V9

06/09 11169-01