Dialogic®

Dialogic® Standard Runtime Library API

Programming Guide

November 2007

05-1881-004

Copyright © 1992-2007, Dialogic Corporation. All Rights Reserved. You may not reproduce this document in whole or in part without permission in writing from Dialogic Corporation.

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

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

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

It is possible that the use or implementation of any one of the concepts, applications, or ideas described in this document, in marketing collateral produced by or on web pages maintained by Dialogic may infringe one or more patents or other intellectual property rights owned by third parties. Dialogic does not provide any intellectual property licenses with the sale of Dialogic products other than a license to use such product in accordance with intellectual property owned or validly licensed by Dialogic and no such licenses are provided except pursuant to a signed agreement with Dialogic. More detailed information about such intellectual property is available from Dialogic’s legal department at 9800 Cavendish Blvd., 5th Floor, Montreal, Quebec, Canada H4M 2V9. **Dialogic encourages all users of its products to procure all necessary intellectual property licenses required to implement any concepts or applications and does not condone or encourage any intellectual property infringement and disclaims any responsibility related thereto. These intellectual property licenses may differ from country to country and it is the responsibility of those who develop the concepts or applications to be aware of and comply with different national license requirements.**

Dialogic, Diva, Eicon, Eicon Networks, Dialogic Pro, EiconCard and SIPcontrol, among others, are either registered trademarks or trademarks of Dialogic. Dialogic's trademarks may be used publicly only with permission from Dialogic. Such permission may only be granted by Dialogic’s legal department at 9800 Cavendish Blvd., 5th Floor, Montreal, Quebec, Canada H4M 2V9. Any authorized use of Dialogic's trademarks will be subject to full respect of the trademark guidelines published by Dialogic from time to time and any use of Dialogic’s trademarks requires proper acknowledgement. Windows is a registered trademark of Microsoft Corporation in the United States and/or other countries. Other names of actual companies and products mentioned herein are the trademarks of their respective owners.

Publication Date: November 2007

Document Number: 05-1881-004

Contents

Tables

Revision History

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

Document No.	Publication Date	Description of Revisions
05-1881-004	November 2007	Global changes: Made global changes to reflect Dialogic brand. Programming Models chapter: Added information about using the Extended Asynchronous Model and **sr_waitevtEx( )** function (IPY00039620).
05-1881-003	June 2005	Programming Models chapter: Removed obsolete note about SR_INTERPOLLID in Synchronous Model section. Added new section on Performance Considerations (PTR 34119). Device Handling chapter: Added separate definition for physical board (brdBn) in Device Concepts section. Added new section on Device Naming and Numbering for Physical Boards (brdBn). Added multimedia (mm) devices in Board-Level Names, Channel-Level Names, and Technology-Specific Device Information sections. (Multimedia devices are supported on Dialogic® Host Media Processing (HMP) software only.) Getting Information About the Structure of a System chapter: Updated code in Device Mapper API Code Example.
05-1881-002	October 2004	Programming Models chapter: Revised Model Combinations section; combined Valid Model Combinations and Invalid Model Combinations into Model Combinations section. Using the Asynchronous Model chapter: Added guideline in Implementing the Asynchronous Model with Event Handlers section stating that Linux signals are no longer used to deliver events. (PTR #28969)
05-1881-001	September 2002	Initial version of document. Much of the information contained in this document was previously published in the *Voice Software Reference - Standard Runtime Library for Linux*, document number 05-1455-003.

About This Publication

The following topics provide information about this publication:

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

Purpose

This publication contains general programming guidelines for the Dialogic® Standard Runtime Library (SRL), which provides a common interface for event handling and other functionality common to all devices (such as network interface, voice, and fax resource devices) provided by Dialogic® Boards.

This publication is a companion to the *Dialogic® Standard Runtime Library API Library Reference*, which provides details on the functions and parameters used by the SRL software.

Applicability

This document is applicable to Dialogic® Host Media Processing (HMP) Software for Linux and to Dialogic® System Release Software for Linux.

Check the Release Guide for your software release to determine whether this document is supported.

Intended Audience

This publication is written for the following audience:

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

How to Use This Publication

This publication assumes that you are familiar with your operating system software and the C programming language.

The information in this guide is organized as follows:

- Chapter 1, "Product Description" provides an overview of the SRL software.
- Chapter 2, "Programming Models" describes the supported programming models in the Linux environment.
- Chapter 3, "Device Handling" describes the concept of a device, the various types of devices, how they are named, and how to access information about devices.
- Chapter 4, "Event Handling" describes the event handling mechanisms provided by the SRL software.
- Chapter 5, "Error Handling" describes the error handling facilities provided by the SRL software including information on how to implement event handlers.
- Chapter 6, "Application Development Guidelines" provides guidelines for selecting and implementing one of the supported programming models.
- Chapter 7, "Using the Synchronous Model" provides guidelines for implementing the Synchronous programming model.
- Chapter 8, "Using the Asynchronous Model" provides guidelines for implementing the Asynchronous programming model.
- Chapter 9, "Using the Extended Asynchronous Model" provides guidelines for implementing the Extended Asynchronous programming model.
- Chapter 10, "Getting Information About the Structure of a System" describes the Device Mapper API that can be used to retrieve information about the structure of a system, such as the number of physical boards, virtual boards, and devices.
- Chapter 11, "Building Applications" provides guidelines for building applications that use the SRL software.
- A Glossary provides a definition of terms used in this guide.

Related Information

This publication is a companion to the *Dialogic® Standard Runtime Library API Library Reference*, which describes the functions and parameters used by the SRL.

Refer to the following documents and websites for more information:

- Release Guide and Release Update for your Dialogic® software release
- *http://www.dialogic.com/support/* (for Dialogic technical support)
- *http://www.dialogic.com/* (for Dialogic® product information)

Product Description 1

This chapter describes the purpose of the Dialogic® Standard Runtime Library (SRL) software.

The primary function of the SRL is to provide a common interface for event handling and other functionality common to all devices. The SRL serves as the centralized dispatcher for events that occur on all devices. Through the SRL, events are handled in a standard manner.

The SRL is a library that contains C functions and a data structure to support application development. Using the SRL, an application can perform the following tasks:

Manage events associated with devices
: The SRL includes a set of event management functions that provide application program control for devices and events, providing the foundation for implementing the supported programming models.

Retrieve information about devices
: The SRL includes a set of standard attribute functions (prefixed ATDV_) that return general information about a device, such as device name, board type, and the error that occurred on the last library function call. Also associated with the SRL is a special device called the SRL_DEVICE that has attributes and can generate events in the same way as other Dialogic® devices. Parameters for the SRL_DEVICE can be set within the application program.

Set and retrieve user-specific context
: The SRL includes two functions, **sr_setparm( )** and **sr_getparm( )**, that enable an application to set up and retrieve user-specific context on a device-by-device basis. An example of user context is an index (or pointer) to a per-device application table.

Retrieve information about the structure of the system
: The SRL includes a set of functions called *Device Mapper* (functions prefixed SRL) that are a subset of the SRL software and return information about the structure of the system, such as a list of all the virtual boards on a physical board.

Specify termination conditions for devices
: The SRL includes the DV_TPT data structure that specifies termination conditions for multitasking functions on devices. For example, you can set the Dialogic® Voice library function **dx_rec( )** to terminate on any digit by setting the tp_termno field in the DV_TPT structure to a value of DX_MAXDTMF and tp_length field to a value of 1.

You can use the SRL interface to simplify application development. The SRL enables you to do the following:

- Write applications using any of the supported programming models
- Write common event handlers to be used by all devices
- Configure devices
- Handle events that occur on the devices
- Return device information
- Create user-defined device information (application-specific information per device)

The SRL software consists of the following files:

- *srllib.h*
- *libsrl.so*

The multithreaded SRL shared object library supports all SRL programming models. See Chapter 2, "Programming Models" for an overview of the supported programming models and Chapter 6, "Application Development Guidelines" for more information about choosing a programming model for your application.

Programming Models 2

This chapter provides an overview of the programming models supported by the Dialogic® Standard Runtime Library (SRL) software in a Linux environment. Topics include:

2.1 Synchronous Versus Asynchronous Programming

Using synchronous programming models, developers can scale an application by simply instantiating more threads or processes (one per channel). This programming model may be easy to encode and manage but it relies on the system to manage scalability. Applying the synchronous programming model can consume large amounts of system overhead, which reduces the achievable densities and negatively impacts timely servicing of both hardware and software interrupts. Using this model, a developer can only solve system performance issues by adding memory or increasing CPU speed or both. The synchronous programming models may be useful for testing or for very low-density solutions.

Asynchronous programming models enable a single program to control multiple devices within a single process. This allows the development of complex applications where multiple tasks must be coordinated simultaneously. Generally, when building applications that use any significant density, you should use the asynchronous programming model to develop field solutions. Asynchronous models:

- Achieve a high level of resource management by combining multiple devices in a single thread.
- Provide better control of applications that have high channel density.
- Provide several extended mechanisms that help you port applications from other operating systems.
- Reduce system overhead by minimizing thread context switching.
- Simplify the coordination of events from many devices.

2.2 Synchronous Model

The Synchronous model is the least complex programming model. Typically, you can use this model to write code for a voice-processing device, then simply create a thread for each device that needs to run this code. You do not need event-driven state machine processing because each function runs uninterrupted to completion.

When using the Synchronous model, each function blocks thread execution until the function completes. The operating system can put individual device threads to sleep while allowing threads that control other devices to continue their actions unabated. When a function completes, the operating system wakes up the function's thread so that processing continues. For example, if the application is playing a file as a result of a **dx_play( )** function call, the calling thread does not continue execution until the play has completed and the **dx_play( )** function has terminated.

Since application execution is blocked by a function in the Synchronous model, a separate application or process is needed for each channel and the operating system allocates execution time for each process.

An application that uses the Synchronous model may have a requirement to service unsolicited events on Dialogic® devices. To service these events the application can use event handlers, also known as *callback* functions. The application polls or waits for events using the **sr_waitevt( )** function. When an event occurs, the SRL calls event handlers automatically, within the context of **sr_waitevt( )**.

See Chapter 7, "Using the Synchronous Model" for more information about implementing the model and Section 4.2, "Using Event Handlers", on page 30 for more information about implementing event handlers.

2.3 Asynchronous Model

In the Asynchronous programming model, after the application issues an asynchronous function, it uses the **sr_waitevt( )** function to wait for events on devices. If there is no event, other processing may take place. If an event is available, information about the event can be accessed (upon successful completion of **sr_waitevt( )**) using event management functions.

Note: The Asynchronous model is also known as the Asynchronous Polled model.

In Asynchronous programming models, the calling thread performs further operations while the function completes. At completion, the application receives event notification. Asynchronous models are recommended for applications that require coordination of multiple tasks and have large numbers of devices. Asynchronous models use system resources more efficiently because they control multiple devices in a single thread.

Due to concurrent processing requirements, a thread cannot block execution while waiting for functions, such as **dx_play( )** or **dx_rec( )**, to finish; this would interfere with the processing requirements of other devices being managed by the thread. In this case, the SRL lets you create an event-driven state machine for each device. Instead of each function blocking until completion, it returns immediately and allows thread processing to continue. Subsequently, when an event is returned through the SRL, signifying the completion of the operation, state machine processing can continue. You can also place user-defined events into the event queue to get single-point state processing control of non-Dialogic application states.

The application can include event handlers (also known as *callback* functions) to service events on Dialogic® devices. The application polls or waits for events using the **sr_waitevt( )** function. When an event occurs, the SRL calls event handlers automatically, within the context of **sr_waitevt( )**.

See Chapter 8, “Using the Asynchronous Model” for more information about implementing the model and Section 4.2, “Using Event Handlers”, on page 30 for more information about implementing event handlers. See also Section 2.6, “Performance Considerations”, on page 16.

2.4 Extended Asynchronous Model

The Extended Asynchronous model is a variation of the Asynchronous model, except that the application can control groups of devices with separate threads. When using the Extended Asynchronous model, you can create multiple threads, each of which controls multiple devices. In such an application, each thread has its own specific state machine for the devices that it controls.

Applications should create threads that manage a **unique** group of devices/channels. For example, you can have one group of devices that provides fax services and another group that provides interactive voice response (IVR) services, and so on. Once a thread is created to service a unique group of devices/channels, it should remain in scope as long as the thread has to retrieve events on that device/channel group. The threads created should not be destroyed until all the devices referred by that thread have been closed. Creating a thread per span or a thread per board is recommended.

Note: Application models that create a thread per channel to retrieve all the events on that channel and/or constantly create and kill threads for every channel and every event are highly discouraged. This results in too much context switching and may not only negatively impact system performance but may also lead to undesired behavior in the application.

The SRL software supports two variants of the Extended Asynchronous model:

sr_waitevtEx( ) Variant

The **sr_waitevtEx( )** function is used to wait for events on certain devices determined by passing an array of the device handles to wait for events on. It offers developers a different approach for event retrieval across multiple threads. This function is merely an extension of the standard **sr_waitevt( )** function and provides the convenience to the calling thread to wait for events on certain devices. Using **sr_waitevtEx( )** does **not** result in any performance improvements over using **sr_waitevt( )**. **sr_waitevt( )** will likely yield higher performance as it bypasses the extra work performed by **sr_waitevtEx( )** in matching the events to devices. As mentioned above, the same device should not be used across multiple threads.

Device Grouping API Variant

The Device Grouping API allows the SRL to make direct associations between threads and devices. Since the Device Grouping functions can be used to separate the functionality of **sr_waitevtEx( )** into two sub-functions (grouping devices together and waiting for events from a group), existing applications that use **sr_waitevtEx( )** can easily be modified to use the Device Grouping API. If an application requires more sophisticated functionality, the Device Grouping API can be used to manipulate a device group after it has been created.

See Chapter 9, “Using the Extended Asynchronous Model” for implementation guidelines. See also Section 2.6, “Performance Considerations”, on page 16.

2.5 Model Combinations

Valid model combinations are listed below:

Synchronous/Event Handlers
: In this combination, the application generally uses synchronous functions with unsolicited events managed by event handlers. Typically, these unsolicited events are exceptions such as hang-up, which are dealt with via handlers. With this combination, the main thread is uncluttered with exception-handling code.

  Using this combination, it is possible to control multiple devices within the same program and still maintain most of the ease in coding. For example, when a voice board is used with a digital network interface board, the voice board handles the user, and the hang-up is received on the digital network interface board.

Synchronous/Polled
: In this combination, the application is written in the Synchronous model, but at various stages, the application polls using **sr_waitevt( )** to verify that a given condition is satisfied which allows synchronization or detection of events that are not time critical.

Synchronous/Polled/Event Handlers
: This combination is similar to the Synchronous/Polled combination except event handlers manage unsolicited events.

Polled/Event Handlers
: This combination uses some asynchronous functions in the main thread, but primarily waits for their termination also in the main thread. Occasional unsolicited events are dealt with via handlers; for example, a hang-up may occur at any time during the application that an event handler can deal with, and the process remains ready for the next call.

Polled/Synchronous
: In this combination, most calls are asynchronous and the main thread waits for termination but, occasionally, synchronous calls are made.

Polled/Synchronous/Event Handlers
: With this combination, the main thread uses **sr_waitevt( )** to wait for termination and uses some synchronous calls, and also deals with some unsolicited events via event handlers.

When an application is written to use event handlers, be aware of the following cautions:

- It is **not** possible to wait for events while in event handlers.
- It is **not** possible to call synchronous functions from within event handlers.

2.6 Performance Considerations

This section applies to Dialogic® DM3 Boards only.

To build scalable applications for higher densities, it is strongly recommended that developers design applications to use a single process for one or more boards and a single thread per span/trunk or a single thread per board. This enables the underlying Dialogic® DM3 libraries to use system resources more efficiently. Using one process per channel or even one thread per channel can have a negative impact on system performance, where systems may become overloaded due to thread and/or process context switching.

Designing applications to use a single process for one or more boards and a single thread per span/trunk or a single thread per board is strongly recommended for the following reasons:

- When Dialogic® DM3 libraries are loaded, they initialize many objects and create threads to enable the asynchronous behavior of the API. At the time of the process shutdown, the libraries de-initialize these objects and threads. This operation requires system-wide resources. If the application is architected by creating one process per channel, the number of processes required would be equivalent to the channel density. The system resource requirement for the initialization/de-initialization and steady state operation would increase accordingly. As the density of the channels increases, the regular operation of the Dialogic® DM3 library stack is impacted. The performance impact is a result of increased CPU scheduling; this in turn affects CPU availability per process, causing processes to starve due to the long line of processes waiting in the scheduler queue. The behavior can cause delayed events, gaps in media play/record, and latency in user-defined callbacks from the Dialogic® DM3 libraries.
- Dialogic® DM3 libraries also create threads per process internally; as the density of the channels increases with the one process per channel model, an increased number of threads runs in the system. Thread context switching is a performance issue with any operating system and severely affects system performance, which may cause delayed events and delayed callbacks.

Therefore, the ideal design to scale applications with higher densities is to use one board per process and one thread per span/trunk or one thread per board.

Device Handling 3

This chapter describes the concept of a Dialogic® device, how devices are named and used, and how to retrieve information about devices. Topics include:

3.1 Device Concepts

The following concepts are key to understanding Dialogic® devices and device handling:

device
: A computer component controlled through a software device driver. A Dialogic® resource board, such as a voice resource, fax resource, and conferencing resource, and network interface board contain one or more logical board devices. Each channel or time slot on the board is also considered a device.

device channel
: A data path that processes one incoming or outgoing call at a time (equivalent to the terminal equipment terminating a phone line). The first two numbers in the product naming scheme identify the number of device channels for a given product. For example, there are 24 voice device channels on a Dialogic® D/240JCT-T1 Board, 30 on a Dialogic® D/300JCT-E1 Board.

device name
: A literal reference to a device, used to gain access to the device via an xx_open( ) function, where "xx" is the prefix defining the device to be opened. The "xx" prefix is "dx" for voice device, "fx" for fax device, and so on. For more information on device names, see Section 3.2, "Device Names", on page 20.

device handle
: A numerical reference to a device, obtained when a device is opened using xx_open( ), where "xx" is the prefix defining the device to be opened. The device handle is used for all operations on that device. For more information on device handles, see Section 3.3, "Opening and Using Devices", on page 26.

physical and virtual boards
: Dialogic® API functions distinguish between physical boards and virtual boards. The device driver views a single physical voice board with more than four channels as multiple emulated Dialogic® D/4x Boards. These emulated boards are called virtual boards. For example, a Dialogic® D/120JCT-LS Board with 12 channels of voice processing contains three virtual boards. A Dialogic® DM/V480A-2T1 Board with 48 channels of voice processing and two T1 lines contains 12 virtual voice boards and two virtual network interface boards.

physical board
A single piece of hardware that fits in a single slot in the computer. A physical board device handle (of the form brdB*n*) is a concept introduced in Dialogic® System Release 6.0. Previously there was no way to identify a physical board but only the virtual boards that make up the physical board. Having a physical board device handle enables Dialogic® API functions to act on all devices on the physical board.

3.2 Device Names

The Dialogic® software assigns device names. The following topics describe how the device names are assigned:

- Overview of Device Names
- Dividing Boards Among Device Types
- Sorting Devices on Dialogic® DM3 Boards
- Sorting Devices on Dialogic® Springware Boards
- Constructing Device Names

3.2.1 Overview of Device Names

The Dialogic® software creates standard device and channel names for boards. These names are input as the **namep** parameter to, for example, the **dx_open( )** and **fx_open( )** functions, which return the device handles necessary for many essential API calls, such as **dx_play( )** and **dx_rec( )**.

3.2.2 Dividing Boards Among Device Types

The Dialogic® software designates devices by type. Some examples of devices types are:

Voice and fax
Device names for this type are prefixed **dxxx**.

Digital network interface
Device names for this type are prefixed **dti**.

Modular station interface
Device names for this type are prefixed **msi**.

Audio conferencing
Device names for this type are prefixed **dcb**.

IP network interface
Device names for this type are prefixed **ipt**.

IP media (for example, Dialogic® IPT Boards and Dialogic® DM/IP Boards)
Device names for this type are prefixed **ipm**.

Voice boards with an integrated digital network interface are assigned both voice devices and one or two digital network interfaces.

3.2.3 Sorting Devices on Dialogic® DM3 Boards

Once the devices are divided by device type, the Dialogic® software sorts the devices within each division. The sort order determines how the device names are constructed. All Dialogic® DM3 Board devices are numbered in sequential order **before** any Dialogic® Springware Board devices have been numbered. For example:

DM3: dtiB1 to dtiB4 / dxxxB1 to dxxxB24
Springware: dtiB5 / dxxxB25 to dxxxB30

3.2.4 Sorting Devices on Dialogic® Springware Boards

Once the devices are divided by device type, the Dialogic® system software sorts the devices within each division. The sort order determines how the device names are constructed. The following topics describe the sorting rules for Dialogic® Springware Boards:

- BLT Boards Only
- PCI Boards Only
- BLT and PCI Boards

3.2.4.1 BLT Boards Only

Board Locator Technology (BLT) boards are sorted in ascending order of the rotary switch setting. Table 1 shows an example.

Table 1. Device Sorting Example for BLT Boards

Sort Order	Board	Address	Rotary Switch	Slot Number
1	Dialogic® VFX/40ESC Board	N/A	0	N/A
2	Dialogic® D/240SC-T1 Board	N/A	1	N/A
3	Dialogic® D/41ESC Board	N/A	1F	N/A

3.2.4.2 PCI Boards Only

The way in which PCI boards are sorted depends on how the rotary switches on the various boards are set:

- Rotary switch settings are unique: In this case, the PCI boards are sorted in ascending order of rotary switch setting.
- Rotary switches are set to zero: In this case, the boards are sorted by bus and slot number.

Note: Both of these methods may be used in the same system.

Table 2 shows an example.

Table 2. Device Sorting Example for PCI Boards

Sort Order	Board	Address	Rotary Switch	Slot Number
1	Dialogic® VFX/PCI Board	N/A	0	2
2	Dialogic® D/41EPCI Board	N/A	0	3
3	Dialogic® D/240PCI-T1 Board	N/A	1	1

3.2.4.3 BLT and PCI Boards

When BLT and PCI boards are used together in a system, the order in which the boards are sorted depends on how the PCI rotary switches are set:

- All BLT and PCI rotary switches are set to unique values: The BLT and PCI boards are all sorted together in ascending order of rotary switch setting.
- PCI rotary switches are set to zero and BLT rotary switches are set to unique values: The PCI boards as a group are ordered before the BLT boards; within the group, PCI boards are sorted by bus and slot number, and BLT boards follow in order of ascending rotary switch setting.
- BLT and PCI rotary switches are set to zero: The PCI boards are ordered before the BLT boards.

Table 3 shows an example.

Table 3. Device Sorting Example for BLT and PCI Boards

Sort Order	Board	Address	Rotary Switch	Slot Number
1	Dialogic® VFX/PCI Board	N/A	0	2
2	Dialogic® D/41EPCI Board	N/A	0	3
3	Dialogic® VFX/40ESC Board	N/A	0	N/A
4	Dialogic® D/240PCI-T1 Board	N/A	1	1

3.2.5 Constructing Device Names

Once the Dialogic® software sorts the devices, it assigns names to both devices and channels within devices. The following topics describe how to construct device names:

- Overview of Device Naming
- Board-Level Names
- Channel-Level Names
- Device Naming and Numbering for Dialogic® DM3 Boards
- Device Naming and Numbering for Physical Boards (brdBn)

3.2.5.1 Overview of Device Naming

Although there is a great deal of 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. A programmer can achieve a great degree of backward compatibility among boards by making the device mapping in the application program hardware independent.

3.2.5.2 Board-Level Names

A board name is assigned to a physical or virtual board in the system. The following board devices are used:

- **dxxxBn**, where **n** is the board device number assigned in sequential order down the list of sorted voice boards. A board device corresponds to a group of two or four voice channels.

  For example, a Dialogic® D/240JCT-T1 Board employs 24 voice channels; the Dialogic® software therefore divides the D/240JCT Board into six voice board devices, each board device consisting of four channels. Boards with an E1 interface, such as the Dialogic® D/600JCT-E1 Board, employ 60 voice channels. The Dialogic® software divides the Dialogic® D/320SC-E1 Board into seven board devices consisting of four channels each and one board device consisting of two voice channels.

  Examples of board device names for voice boards are dxxxB1 and dxxxB2.
- **dtiBn**, where **n** is the board number assigned in sequential order down the list of sorted digital network interface boards. A board device consists of one digital network interface. A Dialogic® DTI/240SC Board contains one dti board device. A Dialogic® DM/V480A-2T1 Board contains two dti board devices. Note that the Dialogic® DM/V480A-2T1 Board also contains 12 dxxx board devices.

  Examples of board device names for digital network interface boards are dtiB1 and dtiB2.
- **msiBn**, where **n** is the board device number assigned in sequential order down the list of sorted modular station interface boards.
- **dcbBn**, where **n** is the board device number assigned in sequential order down the list of sorted audio conferencing boards.
- **iptBn**, where **n** is the logical board number that corresponds to a NIC or NIC address when using IP technology. These devices are used by the Dialogic® Global Call API.
- **ipmBn**, where **n** is the board device number assigned to a media board. These devices are used by the Dialogic® Global Call API and the Dialogic® IP Media Library API.
- **mmBn**, where **n** is the board device number assigned to a media board (multimedia devices are supported in Dialogic® Host Media Processing software only).
- **brdBn**, where **n** is a physical board name assigned to each board in the system. Given the opaque identifier (AUID) for a board, the **SRLGetPhysicalBoardName( )** function can be used to retrieve the physical board name.

3.2.5.3 Channel-Level Names

A board device name can be appended with a channel or component identifier. The following channel-level devices are used:

- **dxxxBnCy**, where **y** corresponds to one of the voice channels. Examples of channel device names for voice boards are dxxxB1C1, dxxxB1C2.
- **dtiBnTy**, where **y** corresponds to one of the digital time slots. Examples of channel device names for digital network interface boards are dtiB1T1, dtiB1T2.
- **msiBnCy**, where **y** corresponds to one of the conferencing channels.
- **dcbBnDy**, where **y** corresponds to 1 (DCB/320), 2 (DCB/640), or 3 DSPs (DCB/960).
- **iptBnTy**, where **y** corresponds to the logical channel number over which call signaling is transmitted when using IP technology. These devices are used by the Dialogic® Global Call API.
- **ipmBnCy**, where **y** corresponds to a media resource on a media board and is used to control media streaming and related functions when using IP technology. These devices are used by the Dialogic® Global Call API and the Dialogic® IP Media Library API.
- **mmBnCy**, where **y** corresponds to one of the multimedia channels (multimedia devices are supported in Dialogic® Host Media Processing software only). Examples of multimedia channel device names are mmB1C1, mmB1C2.

3.2.5.4 Device Naming and Numbering for Dialogic® DM3 Boards

The following conventions apply to Dialogic® DM3 Board naming and numbering:

- All Dialogic® DM3 Board devices are assigned standard device names, for example, dxxxB1, dxxxB2, dtiB1, dtiB2.
- All Dialogic® DM3 channel and timeslot devices are assigned standard device names, for example, dxxxB1C1, dxxxB1C2, dtiB1T1, dtiB1T2.
- A single physical Dialogic® DM3 Board device can contain multiple virtual boards that are each numbered in sequential order; for example, a Dialogic® DM/V960-4T1 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.

  Note: See also Section 3.2.5.5, "Device Naming and Numbering for Physical Boards (brdBn)", on page 25 for information about brdBn physical board devices.
- All Dialogic® DM3 Board devices are numbered in sequential order based on the logical board ID assigned by the Dialogic® DM3 driver (the board with the lowest logical board ID will be assigned the next board number, and so on).

The SRL device mapper functions can be used to return information about the structure of the system including the number of boards in the system and so on. See the *Dialogic® Standard Runtime Library API Library Reference* for more information.

Table 4 provides an example of the device naming and numbering conventions used for Dialogic® DM3 Boards.

Table 4. Device Naming and Numbering Example for Dialogic® DM3 Boards

Hardware	Resource Type	Device Type	Logical Device Names and Numbers
Dialogic® D/480SC-2T1Board (BLT board ID 5)†	Voice	Board Channels ... Channels	dxxxB25 to dxxxB36 dxxxB25C1 to dxxxB25C4 to dxxxB36C1 to dxxxB36C4
	Digital Network Interface	Board Timeslots Timeslots	dtiB5 to dtiB6 dtiB5T1 to dtiB5T24 ‡ dtiB6T1 to dtiB6T24 ‡
Dialogic® DMV/V960-4T1Board (logical board ID 1)†	Voice	Board Channel ... Channel	dxxxB1 to dxxxB24 dxxxB1C1 to dxxxB1C4 to dxxxB24C1 to dxxxB24C4
	Digital Network Interface	Board Timeslots Timeslots Timeslots Timeslots	dtiB1 to dtiB4 dtiB1T1 to dtiB1T24 ‡ dtiB2T1 to dtiB2T24 ‡ dtiB3T1 to dtiB3T24 ‡ dtiB4T1 to dtiB4T24 ‡
† All Dialogic® DM3 Board devices are assigned device numbers (for example, dxxxB1) before all Dialogic® Springware Board devices. ‡ T23 when using ISDN.			

For a given physical board, devices are enumerated sequentially, but there are differences in the way devices are enumerated for Dialogic® Springware Boards and Dialogic® DM3 Boards. For example:

For a Dialogic® Springware D/600JCT Board, devices are enumerated as follows:

- dxxxB1C1-dxxxB8C2 (span 1) then
- dxxxB9C1-dxxxB16C2 (span 2)

For a Dialogic® DM3 DM/V600A Board, devices are enumerated sequentially without any skips as follows:

- dxxxB1C1-dxxxB8C2 then
- dxxxB8C3-dxxxB15C4

3.2.5.5 Device Naming and Numbering for Physical Boards (brdBn)

The following conventions apply to physical board naming and numbering for Dialogic® DM3 Boards:

- All physical board devices are assigned standard device names, such as brdB1, brdB2, brdB3.
- In a single board start and stop, physical board devices are numbered in sequential order based on the order of the board start. The board that is started first is assigned 1, namely brdB1.
- In a system start (all boards in the system are started), the order of the individual board start is based on the logical board ID assigned by the Dialogic® DM3 driver; the board with the lowest

logical board ID is started first and is assigned 1, namely brdB1. The board with the next lowest logical board ID is assigned the next number, namely brdB2, and so on.

The SRL device mapper functions can be used to return information about the structure of the system including the number of physical boards in the system and so on. See the *Dialogic® Standard Runtime Library API Library Reference* for more information.

3.3 Opening and Using Devices

When you open a file in a Linux environment, it returns a unique file descriptor for that file. The following is an example of a file descriptor:

```
int file_descriptor;
file_descriptor = open(filename, mode);
```

Any subsequent action you wish to perform on that file is accomplished by identifying the file using the **file_descriptor**. No action can be performed on the file until it is first opened.

Dialogic® Boards and channels work in a similar manner. You must first open a voice device using **dx_open( )** before you can perform any operation on it. Keep in mind that Dialogic® Springware Boards such as the Dialogic® D/240JCT-T1 Board and Dialogic® D/300JCT-E1 Board comprise both voice resources (**dx_open( )**) and digital interface resources (**dt_open( )**), and that these resources must be opened separately.

When you open a channel or a device connected to the time division multiplexing (TDM) bus using **dx_open( )** or **dt_open( )**, the value returned is a unique Dialogic® device handle for that particular open process on that channel. Typically, the channel device handle is referred to as **chdev**:

```
int chdev;
chdev = dx_open(dxxxBnCy,mode)
```

The channel device name is dxxxBnCy, where B is followed by the board number and C is followed by the number of the voice device channel. An example is dxxxB1C2 for board 1, channel 2.

The device handle for a digital network interface device is referred to as **dtih**:

```
int dtih;
dtih = dt_open(dtiBxTx,mode)
```

The device name is dtiBxTx, where B is followed by the unique board number and T is followed by the number of the time slot (digital channel), 1 to 24 for T1 or 1 to 30 for E1.

For more information on device naming, see Section 3.2, “Device Names”, on page 20.

To use a Dialogic® Voice library function on the channel, you must identify the channel with its channel device handle, **chdev**. The channel device name is used only when opening a channel, and all actions after opening must use the handle **chdev**.

Board devices are opened by following the same procedure, where **bddev** refers to the board device handle. If you use the cached prompt management feature, the concept of a physical board device handle, **brdBn**, is introduced. See the *Dialogic® Voice API Programming Guide* for more information.

Note: Boards and channels are considered separate devices. It is possible to open and use a channel without ever opening the board it is on. There is no board-channel hierarchy imposed by the driver.

In applications that spawn child processes from a parent process, device handles are not inheritable from the parent process to the child process. Make sure that devices are opened in the child process.

Note: When using Dialogic® DM3 Boards, two processes cannot open and access the same device.

To enable you to control the boards and channels in a Linux environment, Dialogic provides libraries of C language functions. For details on opening and closing board and channel devices, see the documentation provided for each library.

Caution: Do not open Dialogic® devices using the Linux **open( )** command.

3.4 Getting Device Information

The SRL provides several ways of retrieving information about devices. Device information is categorized as follows:

- Common Device Information
- Technology-Specific Device Information
- User-Defined Device Information
- SRL-Specific Device Information

The Device Mapper API can also be used to retrieve device information. See Chapter 10, "Getting Information About the Structure of a System" for more information.

3.4.1 Common Device Information

General information exists for all devices, such as the device name and the error that occurred on the last library call. This information can be obtained through SRL standard attribute functions, such as **ATDV_LASTERR( )**. Standard attribute functions return general information about a device, such as device name, board type, and the error that occurred on the last library call.

3.4.2 Technology-Specific Device Information

Technology-specific devices communicate through the SRL and are addressable entities:

- Voice channel and board devices
- Analog or digital time slot and network interface board devices
- Fax channels and board devices
- Modular station interface sets and board devices
- IP network interface channel and board devices

- IP media channel and board devices
- Multimedia channel and board devices (multimedia devices are supported in Dialogic® Host Media Processing software only)

Technology-specific device information can be obtained through the API using technology-specific, extended attribute functions, such as **ATDX_BDNAMEP( )** for voice and **ATFX_BADIOTT( )** for fax. The APIs also may provide functions to get and set technology-specific parameters, such as **dx_getparm( )** for voice and **fx_getparm( )** for fax.

3.4.3 User-Defined Device Information

An application programmer can set up and get application-specific information on a device-by-device basis. Two examples are:

- An index to a per-device application array
- A pointer to a per-device application structure

To set user-specific context, use the **sr_setparm( )** function with the **parmno** parameter set to SR_USERCONTEXT. To get user-specific context, use the **sr_getparm( )** function with the **parmno** parameter set to SR_USERCONTEXT.

3.4.4 SRL-Specific Device Information

Associated with the SRL is a special device called SRL_DEVICE, which has attributes and can generate errors and events similar to any technology-specific device. The SRL_DEVICE is a predefined virtual device handle for the SRL. The SRL provides functions to get and set SRL device information using the SRL_DEVICE parameter in the **sr_getparm( )** and **sr_setparm( )** functions.

Event Handling 4

This chapter describes the event handling facilities provided by the Dialogic® Standard Runtime Library (SRL). Topics include:

4.1 Event Management

The SRL includes event management functions to provide an interface for managing events on devices and handling the program flow associated with the different programming models.

The event management functions include:

sr_dishdlr( )
: disable an event handler

sr_enbhdlr( )
: enable an event handler

sr_putevt( )
: add an event to the SRL event queue

sr_waitevt( )
: wait for next event

sr_waitevtEx( )
: wait for events on certain groups of devices

Application programmers can use event management functions to do the following:

- Utilize asynchronous and/or synchronous functions. An asynchronous function returns immediately to the calling application and returns event notification at some future time. EV_ASYNC is specified in the function's mode. This allows the calling thread to perform further operation while the function completes. A synchronous function blocks the thread until the function completes. EV_SYNC is specified in the function's mode argument.
- Write one program to handle events on several devices.
- Enable or disable application-defined event handlers for a device.

See the *Dialogic® Standard Runtime Library API Programming Guide* for detailed information about each event management function.

4.2 Using Event Handlers

The Synchronous and Asynchronous models can use event handlers (also known as *callback* functions) to act as application-level interrupt service routines that are triggered by the detection of events associated with devices. The following topics provide more information on event handlers:

- Event Handler Overview
- Event Handler Guidelines
- Event Handler Hierarchy
- Using an Application Handler Thread

4.2.1 Event Handler Overview

An event handler is a user-defined function called by the SRL to handle an event or events associated with one or more devices. You can set up event handlers to be invoked for:

- A single event on any device.
- Any event on a specified device.
- Combinations of events on combinations of devices. Where overlap occurs, the most specific event handler is called.

Event notification is implemented using the **sr_waitevt( )** function. The application defines the function(s) that will be called when an event occurs on a device. Events are not received by the process until the **sr_waitevt( )** function is called. When an event occurs (or has previously occurred) on the device, the appropriate event handler for the event is called before **sr_waitevt( )** returns.

In the Asynchronous model that uses event handlers, the main thread typically consists of a single call to **sr_waitevt( )**. If the event handler returns a zero, the SRL will remove the event from the queue so no further processing for that event is possible. If the event handler returns a non-zero value, the SRL will not remove the event from the queue. This will cause **sr_waitevt( )** to return so further processing of that event is possible.

4.2.2 Event Handler Guidelines

The following guidelines apply to event handlers:

- You can enable more than one handler for any event. The SRL calls the most specific handler for an event.
- You can enable general handlers that handle all events on a specified device.
- You can enable a handler for any event on any device.
- You can **not** call synchronous functions in a handler.

4.2.3 Event Handler Hierarchy

The SRL calls event handlers in a hierarchy determined by how device- and event-specific a handler is. The order in which the SRL calls event handlers is listed below:

1. Device/event-specific handlers. Handlers enabled for a specific event on a specific device are called when the event occurs on the device.
2. Device specific/event non-specific handlers. Handlers enabled for any event on a specific device are called only if no device/event specific handlers are enabled for the event.
3. Device non-specific/event non-specific or device non-specific/event-specific handlers (also called *backup* or *fallback* handlers). Handlers enabled for any event, or for a specific event on any device, are called only if no higher-ranked handler has been called. This allows these handlers to act as contingencies for events that might not have been handled by device/event-specific handlers.

The function prototype for user-supplied event handler functions is as follows (shown in ANSI C format):

```
long usr_hdlr(void *parm)
```

4.2.4 Using an Application Handler Thread

An application that uses event handlers can use an event handler thread to wait for events on Dialogic devices and when an event is detected, invoke the appropriate event handler. To create your own application handler thread, with which you can distribute your workload and gain more control over program structure, you can use the application handler thread to make calls to the **sr_waitevt( )** function and execute event handlers. The thread must not call any synchronous functions.

After initiation of the asynchronous function, the application thread can perform other tasks but cannot receive solicited or unsolicited events until the **sr_waitevt( )** function is called.

If a handler returns a non-zero value, the **sr_waitevt( )** function returns in the application thread.

Note: A *solicited event* is an expected event specified using an asynchronous function contained in the device library, such as a “play complete” after issuing a **dx_play( )** function. An *unsolicited event* is an event that occurs without prompting, such as a silence-on or silence-off event in a device.

Error Handling 5

This chapter describes the error handling functionality provided by the Dialogic® Standard Runtime Library (SRL). Topics include:

5.1 SRL Function Error Indication

Most SRL event management functions return a value that indicates success or failure:

- Success is indicated by a return value other than -1.
- Failure is indicated by a return value of -1.

Note: The exception is **sr_getevtdatap( )**, which returns a NULL to indicate that there is no data associated with the current event.

5.2 Retrieving Error Information Using Standard Attribute Functions

If a function fails, the error can be retrieved using the **ATDV_LASTERR( )** or **ATDV_ERRMSGP( )** SRL standard attribute functions. See the *Dialogic® Standard Runtime Library API Library Reference* for more information. If an SRL function fails, retrieve the error by using the **ATDV_LASTERR( )** function with SRL_DEVICE as the argument. To retrieve a text description of the error, use the **ATDV_ERRMSGP( )** function.

For example, if the SRL function **sr_getparm( )** fails, the error can be found by calling the **ATDV_LASTERR( )** function with SRL_DEVICE as the argument.

The *Dialogic® Standard Runtime Library API Library Reference* includes a list of the errors that can occur for each function.

If the error returned by **ATDV_LASTERR( )** is ESR_SYSTEM, an error from the operating system has occurred. The global variable **errno** should be checked.

The error codes are defined in *srllib.h*. See the *Dialogic® Standard Runtime Library API Library Reference* for a list of valid error codes.

Application Development Guidelines

6

This chapter provides guidelines for selecting the most appropriate Dialogic® Standard Runtime Library programming model for your application. Topics include:

6.1 Summary of SRL Programming Model Selections

Select a programming model according to the criteria shown in Table 5. See also Section 2.6, "Performance Considerations", on page 16 for information that impacts system performance.

Table 5. Guidelines for Selecting an SRL Programming Model

Application Requirements	Recommended Programming Model	Threading and Event Handling Considerations
• Few devices	Synchronous model †	Create a separate thread to execute processing for each Dialogic® device.
• Few devices • Needs to service unsolicited events	Synchronous model with event handlers †	Create a separate thread to execute processing for each Dialogic® device. Create your own handler thread that calls the **sr_waitevt( )** function to receive the unsolicited events. See Section 4.2, "Using Event Handlers", on page 30 for more information.
• Many devices • Multiple tasks	Asynchronous model	Call **sr_waitevt( )** to wait for events. Create a single thread to execute processing for all Dialogic® devices.
• Many devices • Multiple tasks • Needs user-defined event handlers	Asynchronous model with event handlers	Create a single thread to execute processing for all Dialogic® devices. Create your own handler thread that calls the **sr_waitevt( )** function to receive the unsolicited events. See Section 4.2, "Using Event Handlers", on page 30 for more information.

Table 5. Guidelines for Selecting an SRL Programming Model (Continued)

Application Requirements	Recommended Programming Model	Threading and Event Handling Considerations
• Many devices • Multiple tasks • Needs to wait for events on more than one group of devices • High availability	Extended Asynchronous model	Create multiple threads to execute processing on groups of Dialogic® devices. Call **sr_waitevtEx( )** for each group of devices, to wait for events on that group. OR Use the Device Grouping API to create groups of devices and wait for events on each group. This model is more efficient than the **sr_waitevtEx( )** alternative. High availability is supported; if one thread dies, other threads can continue processing calls.
† The Synchronous model is not recommended for production applications. However, it can be used for demo or proof of concept applications.		

6.2 Selecting the Synchronous Model

Choose the Synchronous programming model when developing applications that have:

- Only a few devices.
- Simple and straight flow control with only one action per device occurring at any time.

Advantages

The advantages are:

- The Synchronous programming model is the easiest to program, and typically allows fast deployment.
- The model can easily be adapted to handle notification of some unsolicited asynchronous events on Dialogic® devices.

Disadvantages

The disadvantages are:

- A high level of system resources is required since the main thread creates a separate thread for each device. This can limit maximum device density; thus, the Synchronous programming model provides limited scalability for growing systems.
- When a thread is executing a synchronous function, it cannot perform any other processing since a synchronous operation blocks thread execution.
- Unsolicited events are not processed until the thread calls a function such as **dx_getevt( )** or **dt_getevt( )**.
- If you are using event handlers, you may need to set up a way for each event handler to communicate events to another thread. For example, an event handler might need to stop a multitasking function that is active in another thread.

6.3 Selecting the Asynchronous Model

Choose the Asynchronous model for any application that:

- Requires a state machine.
- Needs to wait for events on multiple devices in a single thread.

See also Section 2.6, "Performance Considerations", on page 16 for information that impacts system performance.

Advantages

The advantages are:

- A lower level of system resources than the Synchronous model is required since the Asynchronous model uses one thread for all devices; therefore, the Asynchronous model allows for greater scalability in growing systems
- The Asynchronous model lets you use a single thread to run the entire Dialogic portion of the application.
- When using event handlers, even if the application's non-Dialogic threads block on non-Dialogic functions, the event handlers can still handle Dialogic events. This model ensures that events on Dialogic® devices can be serviced when an event occurs and when the thread is scheduled for execution.

Disadvantages

The disadvantages are:

- The Asynchronous model requires the development of a state machine, which is typically more complex to develop than a Synchronous application.
- When using event handlers, you may need to set up a way for the event handler to communicate events to other threads.

6.4 Selecting the Extended Asynchronous Model

Choose the Extended Asynchronous model for any application that:

- Requires a state machine.
- Needs to wait for events on more than one group of devices.
- Needs to support high availability.

See also Section 2.6, "Performance Considerations", on page 16 for information that impacts system performance.

Advantages

The advantages are:

- A lower level of system resources than the Synchronous model is required since the Extended Asynchronous model uses only a few threads for all Dialogic® devices.
- The Extended Asynchronous model lets you use a few threads to run the entire Dialogic portion of the application.

Disadvantages

The main disadvantage is that the Extended Asynchronous model requires the development of a state machine that is typically more complex to develop than a Synchronous application.

Using the Synchronous Model 7

This chapter provides information on using the Synchronous programming model of the Dialogic® Standard Runtime Library (SRL). Topics include:

7.1 Implementing the Synchronous Model

The following guidelines apply:

- You should use the Synchronous model only for simple and straight flow control, with only one action per device occurring at any time.
- Because each function in the Synchronous model blocks execution in its thread, the main thread in your application must create a separate thread for each device.

Note: The Synchronous model is *not* recommended for production applications. It can be used for demo or proof of concept applications.

7.2 Implementing the Synchronous Model With Event Handlers

The following guidelines apply:

- Create your own event handler thread that calls **sr_waitevt( )** to wait for events on Dialogic® devices. You can set up separate event handlers for the various devices and event types. When an event occurs, the SRL calls event handlers automatically, within the context of the **sr_waitevt( )** function.
- The event handlers must not call **sr_waitevt( )** or any synchronous function. For example, you can use this model to wait for inbound calls synchronously, then service those calls through telephony functions, such as play and record. You could use the event handlers to receive notification of unsolicited hang-up events.

Note: The Synchronous model is *not* recommended for production applications. It can be used for demo or proof of concept applications.

Using the Asynchronous Model 8

This chapter provides information on using the Asynchronous programming model of the Dialogic® Standard Runtime Library (SRL). Topics include:

8.1 Implementing the Asynchronous Model

The following guidelines apply:

- The application uses the **sr_waitevt( )** function to wait for events on Dialogic® devices.
- If an event is available, you can use the following functions to access information about the event:
  - **sr_getevtdev( )** to get the device handle for the current event.
  - **sr_getevttype( )** to get the event type for the current event.
  - **sr_getevtdatap( )** to get a pointer to additional data for the current event.
  - **sr_getevtlen( )** to get the number of bytes of additional data that are pointed to by **sr_getevtdatap( )**.
- Use the **sr_getevtdatap( )** function to extract the event-specific data. Use the other functions to return values about the current event. The values returned are valid until **sr_waitevt( )** is called again.
- After the event is processed, the application determines what asynchronous function should be issued next; the decision to issue a function depends on what event has occurred, and on the last state of the device when the event occurred.

Example Code

The following pseudo code shows a framework for the Asynchronous model.

```
main( )
{
    /* Set up */
    ...
    call_async_fn1( ddd, ... );
    while( 1 ){
        sr_waitevt( tmout );
        switch( sr_getevttype( )){
            case EVT1:
                ...
                break;
            case EVT2:
                ...
                break;
            ...
            ...
            default:
```

```
                ...
        } /* switch */
    } /* while */
...
} /* main */
```

8.2 Implementing the Asynchronous Model with Event Handlers

The Asynchronous model lets your application execute event handlers via an application handler thread. When using an application handler thread, the following guidelines apply:

- To create your own application thread, with which you can distribute your workload and gain more control over program structure, you can use the application handler thread to make calls to the **sr_waitevt( )** function and execute event handlers. The thread must not call any synchronous functions.
- After initiation of the asynchronous function, the application thread can perform other tasks but cannot receive solicited or unsolicited events until the **sr_waitevt( )** function is called.
- If a handler returns a non-zero value, the **sr_waitevt( )** function returns in the application thread.
- In earlier releases, the **sr_hold( )** and **sr_release( )** functions were used to prevent the delivery of events through signal handlers while executing a particular section of code. In Dialogic® System Release 5.1 and later versions, the SRL library no longer uses Linux signals to deliver events; therefore the **sr_hold( )** and **sr_release( )** functions are no longer required and should not be used.

Using the Extended Asynchronous Model 9

This chapter provides information on using the Extended Asynchronous programming model of the Dialogic® Standard Runtime Library (SRL). Topics include:

9.1 Extended Asynchronous Model Variants

The SRL supports two variants of the Extended Asynchronous model. The first variant uses the *Device Grouping API* (a subset of the SRL API) to perform the same basic function as the **sr_waitevtEx( )** variant. The second variant is the traditional variant that uses the **sr_waitevtEx( )** function to control groups of devices with separate threads.

Note: Developers are strongly encouraged to use the Device Grouping API variant since it provides superior performance over the **sr_waitevtEx( )** variant.

The following topics provide more detail on each variant:

- Implementing the Device Grouping API Variant
- Implementing the sr_waitevtEx( ) Variant

9.2 Implementing the Device Grouping API Variant

The Device Grouping API variant of the Extended Asynchronous model was introduced in Dialogic® System Release 6.0 to provide a more efficient alternative to the **sr_waitevtEx( )** variant described in Section 9.3, "Implementing the sr_waitevtEx( ) Variant", on page 46. This variant allows the SRL to make some internal assumptions about the application's behavior and binds these assumptions to the thread context, thus allowing the SRL to perform at levels equivalent to the basic Asynchronous model. The Device Grouping API includes the following functions:

sr_CreateThreadDeviceGroup( )
specifies a list of devices to poll for events

sr_WaitThreadDeviceGroup( )
waits for events on devices in the group

sr_AddToThreadDeviceGroup( )
adds specified devices to the group

sr_RemoveFromThreadDeviceGroup( )
: removes specified devices from the group

sr_GetThreadDeviceGroup( )
: retrieves all devices from the group

sr_DeleteThreadDeviceGroup( )
: removes all devices from the group

See the *Dialogic® Standard Runtime Library API Library Reference* for detailed information about each function.

The following guidelines apply when using the Device Grouping API variant of the Extended Asynchronous model:

How is a device group defined?
: Use the **sr_CreateThreadDeviceGroup( )** function to define a device group. The SRL uses the information to create a queue to store events for these devices.

How is a device group modified?
: Use the **sr_RemoveFromThreadDeviceGroup( )** function to remove devices from a group. The devices in the list passed to the function are removed from the internal list of devices that the thread is managing events for. Use the **sr_AddToThreadDeviceGroup( )** function to add devices to the group.

What happens if a single device is added to multiple thread device groups?
: The device becomes a member of the thread device group of the last thread to call **sr_AddToThreadDeviceGroup( )**. Any events for this device that were in the original device thread group queue are moved to the new one.

How are the devices in a device group tracked?
: The SRL internally stores the devices that are members of a device group. The application does not need to track this information. If the application requires knowledge of which devices are members of a particular thread's device group, the application should call the **sr_GetThreadDeviceGroup( )** function.

What happens to events pending for a device that is removed from a device group?
: As an example, what happens if the dxxxB1C1 device is part of a device group, it has an event pending, and the **sr_RemoveFromThreadDeviceGroup( )** function is called to remove the device from the group? When the **sr_RemoveFromThreadDeviceGroup( )** function is called, the SRL detects that there is an event still pending on the thread's event queue for this device. This event is removed from the thread event queue and placed on the main event queue. The situation is now identical to that described in the next bullet, that is, an event is pending for a device that is not part of any device group. The **sr_waitevt( )** function can be used to retrieve this event.

How are events for a device that is not part of a device group captured?
: To retrieve events for devices that are not part of a device group, the application should call **sr_waitevt( )**. Otherwise, the events accumulate in the event queue and consume memory. This is an improvement over the **sr_waitevtEx( )** implementation, since calling **sr_waitevt( )** in a separate thread is not an option when **sr_waitevtEx( )** is used. When using the Device Grouping API, **sr_waitevt( )** can be used as a fallback to handle events from devices not managed in a group.

What happens to events pending for a device that is added to a device group?

Suppose that the dxxxB1C1 device is **not** part of a device group and that it has an event pending, what happens to the event when the device is **added** to a device group by calling **sr_AddToThreadDeviceGroup( )**? When **sr_AddToThreadDeviceGroup( )** is called, the SRL scans the main event queue to determine if there are any events pending for this device. If events are pending, they are removed from the main queue and placed on the thread event queue.

How is a device group removed?

To remove an entire grouping, the function **sr_DeleteThreadDeviceGroup( )** is used. Any events pending for devices in this group are moved to the main event queue.

How are events for a device group captured?

To capture events for a device group, the function **sr_WaitThreadDeviceGroup( )** is used. It is important that this function be called from the same thread that created the device grouping, that is, the thread that called **sr_CreateThreadDeviceGroup( )**. This is because the SRL internally associates the data passed via **sr_CreateThreadDeviceGroup( )** with the thread context. This is consistent with the behavior of **sr_waitevtEx( )** since it is passed both the device array and the timeout value.

How are events for a device group retrieved?

Upon successful termination of **sr_WaitThreadDeviceGroup( )**, use the Event Data Retrieval functions **sr_getevtdev( )**, **sr_getevttype( )**, **sr_getevtlen( )**, and **sr_getevtdatap( )** to retrieve the associated event information.

How does an application use the Device Grouping API to use a multithreaded model to service the event queue?

With the device groups, this is not necessary. When using device groups, the SRL creates a separate event queue for each group. This is much more efficient than using a single event queue because no synchronization is required to access it.

Example Code for the Device Grouping API Variant

The following pseudo code shows how to implement the Device Grouping API variant of the Extended Asynchronous model.

```
main()
{
    int iNumTrunks = GetNumberOfTrunks();
    for each trunk{
        beginthread (EventPollThread (trunkNumber);
    }
    WaitUntilDone();
}

EventPollThread (TrunkNumber)
{
    int Devices [NumTimeslotsPerTrunk];

    for each device on the trunk{
        Devices [DevNum] = dx_open(...);
    }

    sr_CreateThreadDeviceGroup (Devices, NumTiemslotsPerTrunk);
```

```
    while (1){
        sr_WaitThreadDeviceGroup (-1);
        // do something with the event
    }
}
```

9.3 Implementing the sr_waitevtEx( ) Variant

The **sr_waitevtEx( )** variant of the Extended Asynchronous model is the traditional variant that has been supported in many previous Dialogic® software releases. The following guidelines apply:

- This variant uses multiple threads and calls **sr_waitevtEx( )**.
- If an event is available, you can use the following functions to access information about the event:
  - **sr_getevtdev( )** to get the device handle for the current event.
  - **sr_getevttype( )** to get the event type for the current event.
  - **sr_getevtdatap( )** to get a pointer to additional data for an event.
  - **sr_getevtlen( )** to get the number of bytes of additional data that are pointed to by **gc_getevtdatap( )**.
- Use the **sr_getevtdatap( )** function to extract the event-specific data; use the other functions to return values about the current event. The values returned are valid until **sr_waitevtEx( )** is called again.
- After the event is processed, the application determines what asynchronous function should be issued next depending on what event has occurred and the last state of the device when the event occurred.
- Do not use any Dialogic® device in more than one grouping. Otherwise, it is impossible to determine which thread receives the event.
- Do not use the **sr_waitevtEx( )** function in combination with either the **sr_waitevt( )** function or event handlers.

Getting Information About the Structure of a System 10

This chapter describes the *Device Mapper API*, a subset of the Dialogic® Standard Runtime Library (SRL) API, which can be used to retrieve information about the structure of the system such as the number of physical and virtual boards in a system, and the number of devices on a board.

The SRL Device Mapper API operates on a hierarchy of entities described in the following rules:

- A **physical board** owns zero or more virtual boards.
- A **virtual board** owns zero or more **subdevices**.
- A virtual board is an R4 device.
- A subdevice is an R4 device.
- One or more **jacks** can be associated with one or more R4 devices.

The SRL Device Mapper API consists of the following functions:

SRLGetAllPhysicalBoards( )
Retrieves a list of all physical boards in a system.

SRLGetJackForR4Device( )
Retrieves the jack number for an R4 device.

SRLGetPhysicalBoardName( )
Retrieves the physical board name for the specified AUID.

SRLGetSubDevicesOnVirtualBoard( )
Retrieves a list of all subdevices on a virtual board.

SRLGetVirtualBoardsOnPhysicalBoard( )
Retrieves a list of all virtual boards on a physical board.

Note: The SRL Device Mapper API provides a set of atomic transforms, such as a list of all virtual boards on a physical board. For more complicated transforms, such as information about all the subdevices on a physical board, combine multiple SRL Device Mapper API functions.

Device Mapper API Code Example

The following code demonstrates the use of the Device Mapper API to determine all physical boards, virtual boards, virtual channels, and virtual channel types in the system. It displays the information to the screen and also writes to the *devinfo.log* file.

```
// devinfo.c
// Illustrates use of SRL Device Mapper API
//
// Program Pseudocode
// Start
// Open logfile devinfo.log
// For each Physical Board in system {
//    Get Virtual Boards on associated AUID {
//       Get Virtual Channels on Virtual Board {
//          - Display AUID, Virtual Board and Virtual Device Type for
//            each Virtual Channel
//          - Log information to devinfo.log
//       }
//    }
// }
// Close logfile
// End

// System Header Files
#ifdef WIN32
#include <windows.h>
#endif

#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>

// Dialogic Header Files
#include <srllib.h>

// Function Prototypes
void Shutdown();

// Global Variables
FILE *g_pLogFile;             // Handle for device log file

int main(int argc, char *argv[])
{
    int nPhysBoards;        // Number of Physical Boards in system
    int nVirtualBoards;     // Number of Virtual Boards on a Physical Board
    int nVirtualChans;      // Number of Virtual Channels on a Virtual Board

    AUID *pAuidList = NULL;             // AUID list
    SRLDEVICEINFO *pSrlBoardList = NULL;// SRL Device Info for virtual board
    SRLDEVICEINFO *pSrlChanList = NULL; // SRL Device Info for virtual channel

    AUID *pAuid = NULL;                 // AUID pointer for iterating list
    SRLDEVICEINFO *pSrlBoard = NULL;    // Board pointer for iterating list
    SRLDEVICEINFO *pSrlChan = NULL;     // Channel pointer for iterating list

    int indxPhys;           // Loop index for physical boards
    int indxBoard;          // Loop index for virtual boards
    int indxChan;           // Loop index for virtual channels

    long rc;                // Return code from device mapper functions

    // Open device log file
    g_pLogFile = fopen("devinfo.log", "w");
    if (g_pLogFile == NULL)
    {
        printf("Unable to open devinfo.log, errno = 0x%X\n", errno);
        Shutdown();
    }
```

```
// Determine number of physical boards by
// passing 0 Physical Board count and NULL Auid device info
// Function will fail but number of Physical Boards will
// be returned
nPhysBoards = 0;
pAuidList = NULL;

rc = SRLGetAllPhysicalBoards(&nPhysBoards, pAuidList);
if (rc != ESR_INSUFBUF)
{
    // If error other than ESR_INSUFBUF then either no virtual boards
    // or other unexpected error
    printf("SRLGetAllPhysicalBoards() failed, error = 0x%X\n", rc);
    Shutdown();
}

// Allocate memory for array of AUIDs
pAuidList = (AUID *) malloc(nPhysBoards * sizeof(AUID));
if (pAuidList == NULL)
{
    printf("malloc() failed, unable to allocate memory for AUIDs\n");
    Shutdown();
}

// Retrieve physical board info
rc = SRLGetAllPhysicalBoards(&nPhysBoards, pAuidList);
if (rc != ESR_NOERR)
{
    printf("SRLGetAllPhysicalBoards failed, error = 0x%X\n", rc);
    free(pAuidList);
    Shutdown();
}

// Display header for AUID/board/chan/type display
printf("AUID\tBoard\t\tChan\t\tDev Type\n");
fprintf(g_pLogFile, "AUID\tBoard\t\tChan\t\tDev Type\n");
fflush(g_pLogFile);

// For each Physical Board AUID, determine associated
// virtual board
for (indxPhys = 0; indxPhys < nPhysBoards; indxPhys++)
{
    // Set the AUID pointer to the current item
    pAuid = &pAuidList[indxPhys];

    // Determine number of virtual boards by
    // passing 0 Virtual Board count and NULL SRL device info
    // Function will fail but number of virtual boards will
    // be returned
    nVirtualBoards = 0;
    pSrlBoardList = NULL;

    rc = SRLGetVirtualBoardsOnPhysicalBoard(    *pAuid,
                                                &nVirtualBoards,
                                                pSrlBoardList);
    if (rc != ESR_INSUFBUF)
    {
        // If error other than ESR_INSUFBUF then either no virtual boards
        // or other unexpected error
        printf( "SRLGetVirtualBoardsOnPhysicalBoard() failed, "
                "error = 0x%X\n",
                rc);
        free(pAuidList);
        Shutdown();
    }
```

```
// Allocate memory for number of virtual boards found
pSrlBoardList = (SRLDEVICEINFO *) malloc(
                            nVirtualBoards * sizeof(SRLDEVICEINFO));
if (pSrlBoardList == NULL)
{
    printf("Unable to allocate memory for pSrlBoardList");
    free(pAuidList);
    Shutdown();
}

// Now retrieve all virtual boards on physical board
rc = SRLGetVirtualBoardsOnPhysicalBoard(    *pAuid,
                                            &nVirtualBoards,
                                            pSrlBoardList);
if (rc != ESR_NOERR)
{
    printf( "SRLGetVirtualBoardsOnPhysicalBoard() failed, "
            "error = 0x%X\n",
            rc);
    free(pSrlBoardList);
    free(pAuidList);
    Shutdown();
}

// For each virtual board, determine associated virtual channels
for (indxBoard = 0; indxBoard < nVirtualBoards; indxBoard++)
{
    // Set the Board pointer to the current item
    pSrlBoard = &pSrlBoardList[indxBoard];

    // Determine number of virtual channels by
    // passing 0 Virtual Channel count and NULL SRL device info
    // Function will fail but number of virtual channels will
    // be returned
    nVirtualChans = 0;
    pSrlChanList = NULL;

    rc = SRLGetSubDevicesOnVirtualBoard(    pSrlBoard->szDevName,
                                            &nVirtualChans,
                                            pSrlChanList);
    if (rc != ESR_INSUFBUF)
    {
        // If error other than ESR_INSUFBUF then either no virtual
        // channels or other unexpected error
        printf("SRLGetSubDevicesOnVirtualBoard() failed, "
                "error = 0x%X\n",
                rc);
        free(pSrlBoardList);
        free(pAuidList);
        Shutdown();
    }

    // Allocate memory for number of virtual channels found
    pSrlChanList = (SRLDEVICEINFO *) malloc(
                            nVirtualChans * sizeof(SRLDEVICEINFO));
    if (pSrlChanList == NULL)
    {
        printf("Unable to allocate memory for pSrlChanList");
        free(pSrlBoardList);
        free(pAuidList);
        Shutdown();
    }

    // Now retrieve all virtual channels on physical board
    rc = SRLGetSubDevicesOnVirtualBoard(    pSrlBoard->szDevName,
                                            &nVirtualChans,
                                            pSrlChanList);
```

```
            if (rc != ESR_NOERR)
            {
                printf( "SRLGetSubDevicesOnVirtualBoard() failed, "
                        "error = 0x%X\n",
                        rc);
                free(pSrlChanList);
                free(pSrlBoardList);
                free(pAuidList);
                Shutdown();
            }

            // For each virtual channel, display associated AUID,
            // virtual board and device type
            for (indxChan = 0; indxChan < nVirtualChans; indxChan++)
            {
                // Set the Channel pointer to the current item
                pSrlChan = &pSrlChanList[indxChan];

                // Display AUID, virtual board, virtual channel and
                // device type
                // device type values found in devmapr4.h
                printf( "%d\t%s\t\t%s\t\t%d\n",
                        *pAuid,
                        pSrlBoard->szDevName,
                        pSrlChan->szDevName,
                        pSrlChan->iDevType);
                fprintf(g_pLogFile,
                        "%d\t%s\t\t%s\t\t%d\n",
                        *pAuid,
                        pSrlBoard->szDevName,
                        pSrlChan->szDevName,
                        pSrlChan->iDevType);
                fflush(g_pLogFile);
            } // end virtual channel loop

            // Free memory previously malloc'd
            free(pSrlChanList);

        } // end virtual board loop

        // Free memory previously malloc'd
        free(pSrlBoardList);

    } // end physical board loop

    // Cleanup and terminate application
    free(pAuidList);
    Shutdown();
    return(0);
} // end main()

/*=========================================================================*/
/* NAME: Shutdown() */
/* DESCRIPTION: Terminate the application after cleaning up. */
/* INPUT: None */
/* OUTPUT: Application state when Shutdown() was invoked */
/* RETURN: None */
/*=========================================================================*/
void Shutdown()
{
    fclose(g_pLogFile);
    exit(0);
}
```

Building Applications 11

This chapter provides information on building applications that use the Dialogic® Standard Runtime Library (SRL) software. Topics include:

11.1 Compiling and Linking

Applications that use the SRL software must include references to the SRL header file and must be linked with the appropriate library file. This information is provided the following topics:

- Include Files
- Required Libraries
- Compiling and Linking Guidelines
- Variables for Compiling and Linking Commands

11.1.1 Include Files

The SRL uses one header file, *srllib.h*, that contains the equates required by each application that uses the SRL software.

The following lines of code show where the *srllib.h* file should be included relative to other header files that may be used by the application:

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

where:

- *srllib.h* must be included in code **before** all other Dialogic header files.
- *XXXXlib.h* represents the header file for the device being used. For example, if using a voice device, include the *dxxxlib.h* file. Depending upon the application, you may need to include more than one Dialogic header file for the devices being used.

11.1.2 Required Libraries

Applications developed using the SRL software should be linked with the *libsrl.so* shared object library file. By default, shared object library files are located in the directory given by the INTEL_DIALOGIC_LIB environment variable.

11.1.3 Compiling and Linking Guidelines

Applications using the Dialogic® Digital Network Interface (dti) and Dialogic® Voice (dx) software should be linked with the following libraries in the order shown:

```
/usr/lib/libdti.so;
/usr/lib/libdxxx.so;
/usr/lib/libsrl.so;
```

When compiling an application, Dialogic® library link flags must be listed before all other link flags, for example, Linux library link flags. The following example shows the correct way to list the flags:

```
cc -c application.c

cc application.o -ldti -ldxxx -lsrl -lcurses
```

In the above example, `-ldti -ldxxx -lsrl` are the entries for the Dialogic® library link flags, and `-lcurses` is an entry for OS flags.

Caution: To avoid a core dump, the Dialogic® library link flags must appear **before** the OS flags.

11.1.4 Variables for Compiling and Linking Commands

The following variables provide a standardized way of referencing the directories that contain header files and shared object libraries:

INTEL_DIALOGIC_INC
: Variable that points to the directory where header files are stored.

INTEL_DIALOGIC_LIB
: Variable that points to the directory where shared object library files are stored.

These variables are automatically set at login and should be used in compiling and linking commands.

Note: It is strongly recommended that developers begin using these variables when compiling and linking applications since they will be required in future releases. The name of the variables will remain constant, but the values may change in future releases.

Glossary

asynchronous function: A function that returns immediately to the application and returns event notification at some future time. EV_ASYNC is specified in the function's mode argument. This allows the current thread of code to continue while the function is running.

backup handlers: Handlers that are enabled for all events on one device or all events on all devices.

device: Any object, for example, a board or a channel, that can be manipulated via a physical library.

device grouping functions: Functions that allow a direct association between threads and devices. The Device Grouping APIs can be used to group devices together and wait for events from one of the devices.

device handle: Numerical reference to a device, obtained when a device is opened using **xx_open( )**, where xx is the prefix defining the device to be opened. The device handle is used for all operations on that device.

device mapper functions: Functions that are contained in the device mapper API, a subset of the Dialogic® Standard Runtime Library. They return information about the structure of the system, such as a list of all the virtual boards on a physical boards. The device mapper API works for any component that exposes R4 devices.

device name: Literal reference to a device, used to gain access to the device via an **xx_open( )** function, where xx is the prefix defining the device type to be opened.

event: Any message sent from the device.

event data retrieval functions: Dialogic® Standard Runtime Library functions that retrieve information about the current event, allowing data extraction and event processing.

event handling functions: Dialogic® Standard Runtime Library functions that connect and disconnect events to application-specified event handlers, allowing the user to retrieve and handle events when they occur on a device.

handler: A user-defined function called by the Dialogic® Standard Runtime Library when a specified event occurs on a specified event.

solicited event: An expected event. It is specified using one of the device library's asynchronous functions. For example, for **dx_play( )**, the solicited event is "play complete".

Standard Attribute functions: Dialogic® Standard Runtime Library functions that return general information about the device specified in the function call. Standard Attribute information is applicable to all devices that are supported by the Dialogic® Standard Runtime Library.

Standard Runtime Library (SRL): Device-independent library that contains functions that provide event handling and other functionality common to Dialogic® devices.

Standard Runtime Library parameter functions: Functions that are used to check the status of and set the value of Dialogic® Standard Runtime Library parameters.

subdevice: Any device that is a direct child of another device, for example, a channel is a subdevice of a board device. Since “subdevice” describes a relationship between devices, a subdevice can be a device that is a direct child of another subdevice.

synchronous function: A function that blocks the application until the function completes. EV_SYNC is specified in the function’s mode argument.

unsolicited event: An event that occurs without prompting, for example, silence-on or silence-off events on a channel.

Index