

# **Voice Software Reference - Standard Runtime Library for Linux**

**Copyright © 2001 Dialogic Corporation**

05-1455-003



## COPYRIGHT NOTICE

Copyright © 2001 Dialogic Corporation. All Rights Reserved.

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

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

Publication Date: November, 2001

Part Number: 05-1455-003

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

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

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



# Table of Contents

---

About This Guide.....	vii
<b>1. Overview.....</b>	<b>1</b>
1.1. Event Management Functions.....	2
1.2. Standard Attribute Functions.....	2
1.3. SRL Device Mapper Functions.....	2
1.4. Termination Parameter Table.....	2
<b>2. Application Development Models.....</b>	<b>3</b>
2.1. Synchronous Programming.....	3
2.2. Asynchronous Programming.....	3
2.3. Synchronous Model.....	4
2.4. Polled Model (Asynchronous).....	5
2.5. Callback Model (Asynchronous).....	7
2.5.1. Event Handlers.....	7
2.5.2. Non-Signal Callback Model.....	8
2.5.3. Signal Callback Model.....	9
2.6. Model Combinations.....	12
2.6.1. Valid Model Combinations.....	12
2.6.2. Invalid Model Combinations.....	13
2.7. Compiling and Linking Applications.....	13
<b>3. Event Management Functions.....</b>	<b>17</b>
3.1. Event Handling Functions.....	17
3.2. Event Data Retrieval Functions.....	20
3.3. SRL Parameter Functions.....	20
3.4. Error Handling.....	20
3.5. Include Files.....	20
3.6. Event Management Function Reference.....	21
sr_dishdlr() - disables the handler.....	22
sr_enbhdlr() - enables the handler function.....	25
sr_getevtdatap() - returns the address of the variable data block.....	30
sr_getevtdev() - returns the Dialogic device handle.....	32
sr_getevtlen() - returns the length of the variable data.....	34
sr_getevttype() - returns the event type for the current event.....	36
sr_getfdcnt() - returns the total number of Linux file descriptors.....	38
sr_getfdinfo() - populates the fdarray argument with Linux file descriptors.....	40
sr_hold() - holds off automatic notification of all events.....	43

## ***Voice Software Reference - Standard Runtime Library for Linux***

sr_release() - automatic notification of events resume.....	46
sr_waitvt() - allows events to be handled using the Callback model.....	50
<b>4. Standard Attribute Functions .....</b>	<b>55</b>
4.1. Include Files .....	55
4.2. Standard Attribute Function Reference.....	56
ATDV_ERRMSGP() - returns a pointer to an ASCIIZ string .....	57
ATDV_IOPORT() - returns the base port address .....	59
ATDV_IRQNUM() - returns the interrupt number (IRQ) .....	61
ATDV_LASTERR() - returns a long that indicates the error that occurred .....	63
ATDV_NAMEP() - returns a pointer to an ASCIIZ string.....	65
ATDV_SUBDEVS() - returns the number of subdevices for the device.....	67
<b>5. SRL Device Mapper Functions.....</b>	<b>69</b>
5.1. Include Files .....	70
5.2. SRL Device Mapper Function Reference .....	70
SRLGetAllPhysicalBoards() - returns a list of all the physical boards .....	71
SRLGetJackForR4Device() - returns the jack number .....	74
SRLGetSubDevicesOnVirtualBoard() - returns a list of sub devices .....	76
SRLGetVirtualBoardsOnPhysicalBoard() - returns a list of virtual boards .....	79
<b>Appendix A - Related Publications .....</b>	<b>83</b>
<b>Glossary .....</b>	<b>85</b>
<b>Index .....</b>	<b>87</b>

# List of Tables

---

Table 1. Event Management Function Errors .....	20
---	----

*Voice Software Reference - Standard Runtime Library for Linux*

# About This Guide

---

This guide is used in conjunction with the appropriate Software Reference for Linux for the Dialogic device(s) being used.

This guide contains general programming instructions and functional descriptions for the Dialogic Standard Runtime Library (SRL) for Linux, which consists of the following types of functions:

- Event Management functions
- Standard Attribute functions

This guide also describes the functions provided under the SRL Device Mapper Library Interface (SDM API), a subset of the SRL.

Chapter 1 provides an overview of the Dialogic Standard Runtime Library for Linux.

Chapter 2 provides a description of synchronous and asynchronous programming, including guidelines for choosing either the Synchronous model or Asynchronous Polled model. A description of Asynchronous Callback models and event handlers used with the Callback model is also contained in this chapter. Chapter 2 also describes the general procedure for linking application programs.

Chapter 3 provides a functional description, cautions, and examples for each Event Management function.

Chapter 4 defines Standard Attribute functions and provides a functional description, cautions, and examples for each function.

Chapter 5 defines the functions provided under the SDM API and provides a functional description, cautions, and examples for each function.

This guide assumes users have a working knowledge of C programming in a Linux environment.

***Voice Software Reference - Standard Runtime Library for Linux***

# 1. Overview

---

The Standard Runtime Library (SRL) for Linux is provided as part of the Dialogic voice software for Linux. The major function of the SRL is to provide a common interface for event handling and other common functions to all Dialogic devices. The SRL serves as the centralized dispatcher for events that occur on all Dialogic devices. Through the SRL, events are handled in a standard manner.

The SRL is a library of the following types of C functions and data structures that support application development:

- Event Management Functions
- Standard Attribute Functions
- Termination Parameter Table

The SRL Device Mapper Library Interface (SDM API), is a subset of the SRL which provides a set of atomic transforms for R4 devices in a system.

## 1.1. Event Management Functions

Event Management functions provide an interface to manage events on devices. Most Dialogic functions can be specified to execute in an asynchronous or synchronous mode. The Event Management functions are used to handle the program flow associated with these different modes of application architecture. With Event Management functions, application programmers can perform the following tasks:

1. Retrieve information about an event.
2. Control how events are reported or handled.
3. Control program flow by waiting for events that are generated by a device.
4. Write one program to handle events on several devices in a single process.

*Chapter 2. Application Development Models* contains detailed instructions for using the models in application development. Refer to *Chapter 3. Event*

*Management Functions* for a description and complete reference for each of the Event Management functions.

## **1.2. Standard Attribute Functions**

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.

**NOTE:** Associated with the SRL is a special device called `SRL_DEVICE`, which has attributes and can generate events just as any Dialogic device. Parameters for `SRL_DEVICE` can be set within the application program.

*Chapter 4. Standard Attribute Functions* describes the Standard Attribute functions in detail.

## **1.3. SRL Device Mapper Functions**

The SRL Device Mapper Library Interface (SDM API), is a subset of the SRL which provides C functions and data structures that support programmatic determination of:

1. All the Physical Boards in a Node
2. All the Virtual Boards on a Physical Board
3. All the Sub Devices on a Physical Board
4. The Jack Number of a Device

*Chapter 5. SRL Device Mapper Functions* describes the SRL Device Mapper functions in detail.

## **1.4. Termination Parameter Table**

The SRL also includes the Termination Parameter Table (TPT). The TPT is contained in the *srlib.h* file and consists of `DV_TPT` data structures that are used to specify termination conditions for Dialogic devices. The `DV_TPT` structure is described in the *Voice Programmer's Guide for Linux*.

## 2. Application Development Models

---

The following information is provided in this chapter:

- A description of synchronous and asynchronous programming
- A description and guidelines for choosing the Synchronous model
- A description and guidelines for choosing the Asynchronous Polled model
- A description and guidelines for choosing the Asynchronous Callback model as well as information on event handlers that are used with the Callback model
- Guidelines for using model combinations when developing applications

The Dialogic Standard Runtime Library (SRL) for *Linux* provides synchronous and asynchronous models for use in application development.

### 2.1. Synchronous Programming

Synchronous programming is characterized by functions that block application execution until the function completes. For example, if the application is playing a file as a result of a **dx\_play()** function call, the application will not continue execution until the play is complete 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. See *Section 2.3. Synchronous Model* for a description of the Synchronous model and guidelines for choosing this model for your application.

### 2.2. Asynchronous Programming

Asynchronous programming provides an easier method than the synchronous method for programming more complex applications that may require the coordination of multiple tasks. In asynchronous programming, multiple voice channels can be handled in a single process rather than in separate processes as

## **Voice Software Reference - Standard Runtime Library for Linux**

required in synchronous programming. Handling multiple channels in a single process results in more efficient use of system resources.

Two types of asynchronous models provided for development include:

- Callback
- Polled

The Asynchronous Callback model may be run in signal or non-signal mode.

Choose one of the Asynchronous models to easily program complex applications and to do the following:

- achieve a high level of resource management in your application by combining voice channels in a single process. A single process reduces system overhead required for interprocess communication and simplifies the coordination of events from many devices.
- "fine tune" your programming by selecting one of the Asynchronous models described in *Section 2.4. Polled Model (Asynchronous)* and *Section 2.5. Callback Model (Asynchronous)*.

See the following sections for a description of each Asynchronous model and guidelines for choosing the most appropriate model for your application:

- Polled model - *Section 2.4. Polled Model (Asynchronous)*
- Callback model - *Section 2.5. Callback Model (Asynchronous)*

Guidelines for combining models when developing applications are provided in *Section 2.6. Model Combinations*.

### **2.3. Synchronous Model**

The Synchronous model is the least complex of the programming models and the least complex to program. Since synchronous programming is characterized by functions that run uninterrupted to completion (unlike the Asynchronous models which allow other processing to take place while the function executes), event notification is not necessary when using the Synchronous model.

## 2. Application Development Models

Use the Synchronous model when you are programming an application with a low level of complexity. An application has a low level of complexity if:

- it is more costly to develop and maintain a more complex Asynchronous model for that application
- it does not require the high level of resource management provided by the Asynchronous models
- the number of channels in the application will not require too much system time to support a separate process for each channel

The Synchronous model is useful when an application is used for one telephone line with only one event occurring at a time. You can also use this model for a multi-line system application to handle multiple calls at the same time by writing the application as a single-line application, and then spawning a process for each line that is required.

**NOTE:** For performance considerations when using the Synchronous model, see the `SR_INTERPOLLID` parameter in the `sr_setparm( )` function in *Section 3.6. Event Management Function Reference*.

### 2.4. Polled Model (Asynchronous)

In the Polled model, after an asynchronous function is issued, the application polls for or waits for events using the `sr_waitevt( )` function. If there is no event, other processing may take place between polls. If an event is available, information about the event can be accessed (upon successful completion of `sr_waitevt( )`) using the following functions:

- |                               |   |
|-------------------------------|---|
| <code>sr_getevtdev( )</code>  | Get Dialogic device handle for the current event.   |
| <code>sr_getevttype( )</code> | Get event type for the current event.   |
| <code>sr_getevtdata( )</code> | Get a pointer to additional data for the current event.   |
| <code>sr_getevtlen( )</code>  | Get the number of bytes of additional data that are pointed to by <code>sr_getevtdata( )</code> . |

Use the `sr_getevtdata( )` function to extract the event-specific data. Use the other functions to return values about the current event. The values returned are

## Voice Software Reference - Standard Runtime Library for Linux

valid until `sr_waitevt()` is called again. Event commands can be executed from the main thread via switch statements, and events are processed immediately.

After the event is processed the application determines what asynchronous function should be issued next, depending on:

- the event that occurred
- the last state of the channel when the event occurred

An example of the Polled model is shown below:

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

Choose the Polled model for an application that requires a more complex programming model than the Synchronous model, but does not need to use event handlers to process events.

For further information, refer to the *pansr* program included on the voice software for *Linux* distribution media. The *pansr* program is a polled answer demo that runs in polled mode only. NO handlers are enabled for this program. *pansr* is state driven and runs asynchronously.

### 2.5. Callback Model (Asynchronous)

The Callback model runs in two modes:

- Non-Signal
- Signal

With the Callback model, event handlers can be enabled or disabled for specific events on specific devices.

#### 2.5.1. Event Handlers

An event handler is a user-defined function called by the SRL to handle a specified event that occurs on a specified device. The following guidelines apply to event handlers:

- More than one handler can be enabled for an event. The SRL calls all the specified handlers when the event is detected in the Non-Signal Callback model. In the Signal Callback model, event handlers are called from a central *Linux* signal handler within the SRL.

**NOTE:** In the Signal Callback model, event handlers must follow *Linux* signal handler rules since event handlers are called by a *Linux* signal handler.

- General handlers can be enabled that handle any event on a specific device.
- Handlers can be enabled to handle any event on any device.
- Synchronous functions cannot be called in a handler.
- In the Signal Callback model the handler return value is meaningless. In the Non-Signal Callback model, the handler return value carries meaning, as explained in *Section 2.5.2. Non-Signal Callback Model*.

**NOTE:** In the signal mode, the SRL removes all events before control is returned to the main thread.

## ***Voice Software Reference - Standard Runtime Library for Linux***

The SRL calls handlers according to a hierarchy that depends on how specific or general the handler is. Handlers are listed below in the order the SRL will call them:

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. Handlers enabled for any event or a specific event on any device (also called "backup" or "fallback" handlers) are called only if no higher-ranked handler has been called. This allows these handlers to act as contingencies for events that may not have been handled by device/event-specific handlers.

### **2.5.2. Non-Signal Callback Model**

The Non-Signal Callback model uses the non-signal mode of event notification. In non-signal mode, the application polls for 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( )**.

After initiation of the asynchronous function, the process can perform other tasks but cannot receive solicited or unsolicited events until **sr\_waitevt( )** is called. A solicited event is an expected event specified using an asynchronous function contained in the device library (e.g., a "play complete" after issuing a **dx\_play( )** function). An unsolicited event is an event that occurs without prompting (e.g., silence-on or silence-off events on a channel).

With the pure Non-Signal Callback model, 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

## 2. Application Development Models

that event is possible. For details, refer to the description of `sr_waitevt()` in *Chapter 3. Event Management Functions*.

An example of the Non-Signal Callback model is shown below.

```
main( )
{
    /* set up */
    .
    .
    /* enable handlers */
    .
    .
    /*
     * Handlers catch all events
     */
    (void)sr_waitevt( -1 );
    ...
    /* cleanup */
    ...
}
```

Choose the Non-Signal Callback model for an application that:

- requires a more complex programming model than the Synchronous model
- will benefit from the use of event handlers
- will not use *Linux* signals for event notification

### 2.5.3. Signal Callback Model

The Signal Callback model uses *Linux* signal handlers for event notification. A benefit of using *Linux* signals for event notification is a fast response time to unsolicited events. In signal mode, the process receives notification of events via a central *Linux* signal handler within the SRL.

#### CAUTION

Since event handlers are called from a *Linux* signal handler in this model, event handlers must follow *Linux* signal handler rules.

After initiation of the asynchronous function, the process can perform other tasks and receive solicited or unsolicited events.

When an event occurs, a Linux SIGPOLL signal is generated by the device. This signal interrupts the process and control is passed to a central signal handler within SRL. From the central signal handler, the SRL calls all event handlers that have been enabled for that event on that device using the `sr_enbhdr()` function. After all event handlers are called, control returns to the main thread at the place where the interrupt occurred and the main thread continues until notification of the next event.

### CAUTION

Dialogic functions cannot be called within a Linux signal handler. Doing so can cause unexpected results such as a Dialogic function lockup.

In the pure Signal Callback model all processing (after initial setup) takes place in handlers. The main thread will typically consist of an endless sleep loop (nothing happens in the main thread while the handlers are being called). An example of this type of program is shown below:

```
main( )
{
    /* set up */
    .
    .
    /* enable handlers */
    .
    .
    /* Call first async function */
    .
    /* Sit in endless loop dealing with events */
    while( 1 ){
        sleep( -1 );
    }
}
```

When the SRL operates in signal mode, it is possible to reenter a function within a handler that was in mid-execution when the signal occurred which may cause unexpected results. For example, `malloc()` on some systems may not be reentrant and if it is reentered, the heap may become corrupted. This problem can also occur if a call within a handler modifies the same global data as an interruptible call in the main thread. For example, `malloc()` and `free()` manipulate the same global data so unexpected results may occur if a handler containing a `free()` interrupts a call to `malloc()`.

## 2. Application Development Models

All non-reentrant areas of code should be protected against reentry with `sr_hold()/sr_release()` pairs as shown below:

**NOTE:** Function libraries provided by third parties may contain non-reentrant code that is not readily apparent. When using functions contained in these libraries in the Signal Callback model, always check that the functions have no adverse effect.

```
void handler( )
{
    ...
    ...malloc( )...;
    ...
}

main( )
{
    ...
    ...
    /* enable handlers */
    if( sr_enbhdr( ..., ..., handler ) == -1 ){
        do_Error( );
    }
    ...
    /* Call 1st async function which kicks everything off */
    ...
    /* Sit in loop in main thread receiving events while doing further
    * processing */
    while( 1 ){
        ...
        /* Protect a critical region */
        if( sr_hold( ) == -1 ){
            do_error( );
        }
        ...malloc( ... ) ...;
        if( sr_release( ) == -1 ){
            do_error( );
        }
        ...
    }
    ...
}
```

Choose the Signal Callback model for an application that:

- requires a more complex programming model than the Synchronous model
- will benefit from the use of event handlers
- will use Linux signals for event notification

For further information, refer to the *cbansr* program included on the voice software for Linux distribution media. The *cbansr* program is a callback answer demo that runs on both signal call back and non-signal call back. The program is state driven but uses event handlers and runs asynchronously.

By default, *cbansr* compiles for non-signal mode. If signal mode is desired, signal should be defined in a compile-time flag found in the *makefile* supplied with the demo. An example of compiling for signal mode is shown below:

```
CFLAGS=-O -DSIGNAL
```

## **2.6. Model Combinations**

Applications can be developed using a combination of the models. This section describes valid and invalid model combinations.

### **2.6.1. Valid Model Combinations**

Valid model combinations are listed below:

- **Synchronous/Callback**  
In this combination, the application generally uses synchronous functions with exceptions (unsolicited events) handled by a Callback model. Typically, these exceptions are unsolicited events 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 D/4x board is used with a DTI/xxx board, the D/4x board handles the user, and the hang-up is received on the DTI/xxx board.
- **Synchronous/Polled**  
In this combination, the application is written in the Synchronous model, but at various stages, the application polls using `sr_waitvt( )` to verify that a given condition is satisfied which allows synchronization or detection of events that are not time critical.
- **Synchronous/Polled/Non-Signal Callback**  
This combination is similar to the Synchronous/Polled combination except Non-Signal Callback handlers handle exceptions.
- **Polled/Non-Signal Callback**  
This combination uses some asynchronous functions in the main thread, but primarily waits for their termination also in the main thread.

## 2. Application Development Models

Occasional exceptions are dealt with via handlers (e.g., 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/Callback**  
With this combination, the main thread uses `sr_waitevt( )` to wait for termination and uses some synchronous calls, and also deals with some exceptions (unsolicited events) via handlers.

### 2.6.2. Invalid Model Combinations

When an application is written using one of the Callback models as the dominant model, it is illegal to combine the Polled and Synchronous models in the application for the following reasons:

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

Instead of using synchronous functions, use the asynchronous versions of the functions and wait for terminations in the main thread.

## 2.7. Compiling and Linking Applications

Application programs developed using the Voice Library for UNIX should be linked with the following libraries:

```
/usr/lib/libdti.a;  
/usr/lib/libdxxx.a;  
/usr/lib/libsr1.a;
```

They should be linked in the order specified above.

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:

## ***Voice Software Reference - Standard Runtime Library for Linux***

```
cc -c application.c
```

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

In the above example, `-ldti -ldxxx -lsrl` are the entries for the Dialogic library link flags, and `-ldl -lLiS -lcurses` are the entries for the OS flags (including Linux Streams).

## **2. *Application Development Models***



## 3. Event Management Functions

---

The Event Management functions of the SRL are device independent and provide applications with asynchronous event management capabilities. The functions can be divided into the following categories:

- |                      |   |
|----------------------|---|
| Event handling       | • enable and disable event handlers or wait a specified amount of time for the next event |
| Event data retrieval | • obtain information about the current event  |
| SRL parameter        | • set and request SRL parameters  |

The Event Management functions are listed below. *Section 3.6. Event Management Function Reference* describes each of these functions in detail.

### 3.1. Event Handling Functions

<b>sr_dishdlr()</b>	• disable an event handler
<b>sr_enhdlr()</b>	• enable an event handler
<b>sr_hold()</b>	• hold off automatic event notification
<b>sr_release()</b>	• release held events
<b>sr_waitevt()</b>	• wait for next event

Event handling functions are used to enable or disable event handlers, hold events while other processing takes place, or specify the amount of time to wait for the next event. **sr\_hold()** and **sr\_release()** are used for signal mode only to protect non-reentrant areas of code against reentry.

Event handlers can be enabled and disabled for specific events on specific devices. Backup event handlers can also be enabled to serve as contingencies for events that have not been specifically enabled.

## ***Voice Software Reference - Standard Runtime Library for Linux***

The SRL calls handlers according to a specific hierarchy. The ranking of a handler in the hierarchy depends on how specific the handler is. The following is the SRL handler hierarchy in descending order:

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

**NOTE:** Device non-specific/event specific handlers are treated as item 3 above and are not recognized as a different category.

Two modes of event notification are possible:

- signal via Linux signals
- non-signal via **sr\_waitevt( )**

In signal mode, a SIGPOLL signal is generated by the device whenever an event occurs. The current process is interrupted and control is passed to the central signal handler within SRL. From that point, the SRL calls all handlers that are enabled for that event on that device via **sr\_enbhdlr( )**. Control then returns to the point in the thread of code from which the interruption occurred.

In the models using pure non-signal modes (Polled and/or Callback), events are not received by the process until the function **sr\_waitevt( )** is called. When an event occurs (or has previously occurred) on the device in **sr\_waitevt( )**, all appropriate event handlers for the event are called before **sr\_waitevt( )** returns. Typically with the Polled model, no event handlers are enabled and the event can be processed upon return of **sr\_waitevt( )**. Data is extracted via the event data extraction functions **sr\_getevtdev( )**, and **sr\_getevttype( )**. See *Section 2.4. Polled Model (Asynchronous)* for details on using the Polled model for application development.

#### 3.2. Event Data Retrieval Functions

<code>sr_getevtdatap( )</code>	• return a pointer to the variable data associated with the current event
<code>sr_getevtdev( )</code>	• get the Dialogic handle for the current event
<code>sr_getevtlen( )</code>	• get the length of variable data associated with the current event
<code>sr_getevttype( )</code>	• get the event type for the current event

Event data retrieval functions are used to retrieve information about the current event allowing data extraction and event processing.

#### 3.3. SRL Parameter Functions

<code>sr_getparm( )</code>	• get an SRL parameter
<code>sr_setparm( )</code>	• set an SRL parameter

SRL parameter functions are used to check the status of and set the value of SRL parameters, for example, specify the mode of event notification (signal or non-signal).

#### 3.4. Error Handling

Most SRL Event Management functions return a value that indicates success or failure:

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

**NOTE:** The exception is the function `sr_getevtdatap( )` which returns a NULL to indicate an error.

If a function fails, the error can be retrieved using `ATDV_LASTERR( )` and `ATDV_ERRMSGP( )` functions described in *Chapter 4. Standard Attribute*

*Functions.* To get an error on a Standard Attribute function, retrieve the last error by specifying `SRL_DEVICE` as the device on `ATDV_LASTERR()`. For example, if the SRL function `sr_getparm()` fails, the error can be found by calling `ATDV_LASTERR()` and `open()` (`SRL_DEVICE`).

Each function description in this chapter includes a list of the errors that can occur for that function. If the error returned by `ATDV_LASTERR()` is `ESR_SYSTEM`, a Linux system error has occurred; the global variable `errno` contained in `errno.h` must be checked.

The error codes are defined in `srllib.h` and are listed in *Table 1*.

**Table 1. Event Management Function Errors**

<b>Error Define</b>	<b>Error String</b>
<code>ESR_SYSTEM</code>	System error, consult <code>errno</code> in <code>errno.h</code>
<code>ESR_TMOU</code>	Timeout event, the function has timed out

### 3.5. Include Files

The following lines must be included in application code prior to calling any Event Management functions:

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

`DEVICElib.h` is the header file for the device being used. For example, if the device is a D/4x, the `dxxxlib.h` file is included.

**NOTE:** `srllib.h` must be included in code before all other Dialogic header files.

### 3. Event Management Functions

#### 3.6. Event Management Function Reference

This section provides a detailed description of the Event Management functions in alphabetical order.

**NOTE:** The functions described in this section use and return device-specific data. Valid entries for Dialogic devices can be found in the *Voice Software Reference: Programmer's Guide for Linux, Appendix A*.

---

<b>Name:</b>	int sr_dishdlr(dev, evt_type, handler)
<b>Inputs:</b>	int dev <ul style="list-style-type: none"> <li>• Dialogic device handle</li> </ul> int evt_type <ul style="list-style-type: none"> <li>• Event type</li> </ul> int (*handler)() <ul style="list-style-type: none"> <li>• Event handling function</li> </ul>
<b>Returns:</b>	0 if success -1 if failure
<b>Includes:</b>	srllib.h
<b>Type:</b>	Event Control function

---

## ■ Description

The **sr\_dishdlr()** function disables the handler **handler()** that was previously enabled using **sr\_enbhdlr()** on a device/event type/handler triple. **sr\_dishdlr()** can be called from within a handler even if the same handler is being disabled.

The function parameters are described as follows:

Parameter	Description
<b>dev:</b>	specifies the valid device handle obtained when the device was opened using <b>xx_open()</b> , where <i>xx</i> is the prefix identifying the device to be opened. Specify <b>EV_ANYDEV</b> to be notified of an event on any device.
<b>evt_type:</b>	specifies the event for which the application is waiting, for example: <ul style="list-style-type: none"> <li>• specify the specific event. Refer to <i>Appendix A, Standard Runtime Library: Entries and Returns</i>, in the appropriate software reference for <i>Linux</i> for a list of possible event types.</li> <li>• specify <b>EV_ANYEVT</b> in <b>evt_type</b> and the device in the <b>dev</b> parameter to be notified of any event on a specific device.</li> <li>• specify <b>EV_ANYEVT</b> in <b>evt_type</b> and <b>EV_ANYDEV</b> in the <b>dev</b> parameter to be notified of any event on any device.</li> </ul>
<b>handler:</b>	points to an application-defined event handler that will process the event. See <b>sr_enbhdlr()</b> for more information on the application-defined event handlers.

## ■ Cautions

Only one handler is disabled by this function and must be disabled under the same conditions as the handler was enabled. To disable device non-specific and/or event non-specific handlers, specify `EV_ANYDEV` for the device and/or `EV_ANYEVT` for the event type.

When a handler is disabled while an event is being dealt with, the handler is not called if it has not been called previously.

## ■ Example

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

int dx_handler()
{
    printf( "dx_handler() called, event is 0x%x\n", sr_getevtttype());
    return( 0 );
}
/* tell SRL to dispose of the event */

main()
{
    int dxxxdev;
    int mode = SR_POLLMODE;

    /* set SRL to run in non-signal mode */
    if( sr_setparm( SRL_DEVICE, SR_MODEID, &mode ) == -1 ){
        printf( "Failed to set SRL mode\n" );
        exit( 1 );
    }

    /* open dxxx channel device */
    if(( dxxxdev = dx_open( "dxxxB1C1", 0 ) ) == -1 ){
        printf( "dx_open failed\n" );
        exit( 1 );
    }

    /* enable handler dx_handler on device dxxxdev .... */
    if( sr_enhdlr( dxxxdev, EV_ANYEVT, dx_handler ) == -1 ){
        printf( "Error: could not enable handler\n" );
        exit( 1 );
    }

    /* Disable the handler */
    if( sr_dishdlr( dxxxdev, EV_ANYEVT, dx_handler ) == -1 ){
        printf( "Error: could not disable handler\n" );
        exit( 1 );
    }
}
```

**■ Errors**

If the function returns a -1 to indicate an error, use **ATDV\_LASTERR()** to determine the reason for the failure. If the value returned is **ESR\_SYSTEM**, consult **errno** in *errno.h* for the following possible value:

**EINVAL**      • The dev/evt/handler triple has not already been registered.

**■ See Also**

- **sr\_enbhdlr()**
- *Voice Software Reference: Programmer's Guide for Linux, Appendix A*

---

**Name:** int sr\_enbhdr(dev, evt\_type, handler)  
**Inputs:** int dev                   • Dialogic device handle  
int evt\_type                   • Event type  
int (\*handler)( )           • Event handling function  
**Returns:** 0 if success  
-1 if failure  
**Includes:** srllib.h  
**Type:** Event Control function

---

## ■ Description

The **sr\_enbhdr()** function enables the handler function **handler()** for the device/event pair. A handler is a user-defined function called by the SRL to handle a specified event that occurs on a specified device.

The function parameters are described as follows:

Parameter	Description
<b>dev:</b>	specifies the valid device handle obtained when the device was opened using <b>xx_open()</b> , where <i>xx</i> is the prefix identifying the device to be opened. Specify EV_ANYDEV to be notified of an event on any device.
<b>Evt_type:</b>	specifies the event for which the application is waiting; for example: <ul style="list-style-type: none"> <li>• specify the specific event. Refer to the <i>Voice Software Reference: Programmer's Guide for Linux, Appendix A</i>, for a list of possible event types.</li> <li>• specify EV_ANYEVT in <b>evt_type</b> and the device in the <b>dev</b> parameter to be notified of any event on a specific device.</li> <li>• specify EV_ANYEVT in <b>evt_type</b> and EV_ANYDEV in the <b>dev</b> parameter to be notified of any event on any device.</li> </ul>
<b>handler:</b>	points to an application-defined event handler, as described previously, that will process the event.

If more than one handler is enabled for an event, the SRL calls all the specified handlers when the event is detected. Also, more general handlers can be enabled

that handle **any** event on a given device or handlers can be enabled to handle **any** event on **any** device.

The following rules apply to handlers:

- Synchronous functions cannot be called in a handler.
- **sr\_waitevt()** cannot be called in a handler.
- Handlers must return a 1 to advise the SRL to keep the event or 0 to advise the SRL to release the event. If a handler returns 0 and the event is released, **sr\_waitevt()** will not return for that event.

**NOTE:** In signal mode, the SRL removes all events before control is returned to the main thread.

The SRL calls handlers according to a hierarchy that depends on how specific the handler is. The SRL handler hierarchy is listed below in descending order:

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

**NOTE:** Device non-specific/event specific handlers are treated as item 3 above and are not recognized as a different category.

When a handler is called in signal mode, control is passed to the handler from the main thread except for regions protected by **sr\_hold()** or **sr\_release()** pairs. Handlers themselves cannot be interrupted by further events.

In non-signal mode, the handler is called from within the function **sr\_waitevt()** or from within a device-specific synchronous function if the event occurs within the specified timeout. For details, see the *Voice Software Reference: Programmer’s Guide for Linux, Appendix A*.

---

A handler may be enabled from within another handler since SRL's event handling is fully reentrant. For the same reason, opening and closing of devices is permitted inside handlers. However, handlers cannot be called from within handlers since it is not possible to call **sr\_waitevt()** from within a handler. Control must return to the main thread before **sr\_waitevt()** can be called again.

Handlers return 0 if the event is to be removed and will not be returned by **sr\_waitevt()**.

### ■ Cautions

If two handlers are enabled for the same event on the same device, the order in which they will be called is undetermined.

If a second handler is enabled for the current event from within the first handler, the second handler will also be called before **sr\_waitevt()** returns.

If a device with outstanding events is closed within a handler, none of its handlers are called. All handlers enabled on that device are disabled and no further events are serviced on that device.

If more than one handler is enabled for a given event and the first handler is released, all handlers for the event are called before the event is deleted.

When the SRL operates in signal mode, it is possible to reenter a function within a handler that was in mid-execution when the signal occurred. This can cause unexpected results. For example, **malloc()** on some systems may not be reentrant and if it is reentered, the heap may become corrupted. This problem can also occur if code executed within a handler modifies the same global data as interruptible code in the main thread. For example, **malloc()** and **free()** manipulate the same global data. Therefore, it is dangerous if a handler containing a **free()** can interrupt a section of main thread containing a **malloc()**.

**■ Example**

```
#include <srllib.h>

#include <dxxxlib.h>

int dx_handler()
{
    printf( "dx_handler() called, event is 0x%x\n", sr_getevtttype());
    return( 0 );
    /* tell SRL to dispose of the event */
}

main()
{
    int dxxxdev;
    int mode = SR_POLLMODE;

    /* set SRL to run in non-signal mode */
    if( sr_setparm( SRL_DEVICE, SR_MODEID, &mode ) == -1 ){
        printf( "Failed to set SRL mode\n" );
        exit( 1 );
    }

    /* open dxxx channel device */
    if( ( dxxxdev = dx_open( "dxxxB1C1", 0 ) ) == -1 ){
        printf( "dx_open failed\n" );
        exit( 1 );
    }

    /* Enable a handler for all events on dxxxdev */
    if( sr_enbhdr( dxxxdev, EV_ANYEVT, dx_handler ) == -1 ){
        printf( "Error: could not enable handler\n" );
        exit( 1 );
    }
}
```

**■ Errors**

If the function returns a -1 to indicate an error, use **ATDV\_LASTERR()** to determine the reason for the failure. If the value returned is **ESR\_SYSTEM**, consult **errno** in *errno.h* for the following possible values:

- |        |   |
|--------|---|
| ENOMEM | • The Event Library has run out of space when allocating memory for internal data structures. |
| EINVAL | • The dev/evt/handler triple has already been registered.                                     |

■ **See Also**

- `sr_dishdr()`
- *Voice Software Reference: Programmer's Guide for Linux, Appendix A*

***sr\_getevtdatap()***

***returns the address of the variable data block***

---

**Name:** void \*sr\_getevtdatap()  
**Inputs:** none  
**Returns:** Address of variable data block  
NULL if no additional data  
**Includes:** srllib.h  
**Type:** Event Data Retrieval function

---

## ■ Description

The **sr\_getevtdatap()** function returns the address of the variable data block associated with the current event. Use this data pointer and the event length **sr\_getevtlen()** to extract the event data. The function returns a value of NULL if no additional data is associated with the current event.

**NOTE:** Information about the data that can be returned by this function can be found in the *Voice Software Reference: Programmer's Guide for Linux, Appendix A*.

## ■ Cautions

If the program is executing a handler, the value returned is valid throughout the scope of the handler. If the program is not in a handler, the value returned is valid only until **sr\_waitevt()** is called again.

## ■ Example

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

int dx_handler()
{
    printf( "Got event with event data 0x%x\n", *(sr_getevtdatap()));

    /* Tell SRL to keep the event */
    return( 1 );
}

main()
{
    int dxxxdev;
    int mode = SR_POLLMODE;

    /* Set SRL to run in polled mode */
    if( sr_setparm( SRL_DEVICE, SR_MODEID, &mode ) == -1 ){
        printf( "Cannot set SRL to polled mode\n" );
        exit( 1 );
    }
}
```

```

/* open the device */
if( ( dxdev = dx_open( "dxxxB1C1", 0 ) ) == -1 ){
    printf( "failed to open device\n" );
    exit( 1 );
}

/* Enable handlers */
if( sr_enbhdlr( dxdev, EV_ANYEVT, dx_handler ) == -1 ){
    printf( "Could not enable handler: error = %s\n",
           ATDV_ERRMSGP( SRL_DEVICE ) );
    exit( 1 );
}

/* Generate events via async calls */
if( dx_sethook( dxdev, DL_ONHOOK, EV_ASYNC ) == -1 ){
    printf( "dx_sethook failed: error = %s\n", ATDV_ERRMSGP( dxdev ) );
    exit( 1 );
}

/*
 * Wait forever while handlers deal with events
 * All handlers return 0 except the one for the last
 * event returns 1 telling SRL to leave the event to wake up
 * sr_waitevt().
 */
(void)sr_waitevt( -1 );

/* Cleanup */
}

```

## ■ Errors

None.

## ■ See Also

- `sr_getevtlen()`
- *Voice Software Reference: Programmer's Guide for Linux, Appendix A*

**Name:** int sr\_getevtdev()  
**Inputs:** none  
**Returns:** Dialogic device handle if successful  
-1 if no current event  
**Includes:** srllib.h  
**Type:** Event Data Retrieval function

---

## ■ Description

The **sr\_getevtdev()** function returns the Dialogic device handle associated with the current event. If no current event exists, a value of -1 is returned. This function returns SRL\_DEVICE if a timeout occurs on SRL\_DEVICE. See also **sr\_waitevt()**.

**NOTE:** Information about the device handles that can be returned by this function can be found in the *Voice Software Reference: Programmer's Guide for Linux, Appendix A*.

## ■ Cautions

If the program is executing a handler, the value returned is valid throughout the scope of the handler. If the program is not in a handler, the value returned is valid only until **sr\_waitevt()** is called again.

## ■ Example

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

int dx_handler()
{
    printf( "Got event on device %s\n", ATDV_NAMEP( sr_getevtdev()));

    /* Tell SRL to keep the event */
    return( 1 );
}
```

```

main()
{
    int dxxxxdev;
    int mode = SR_POLLMODE;

    /* Set SRL to run in polled mode */
    if( sr_setparm( SRL_DEVICE, SR_MODEID, &mode ) == -1 ){
        printf( "Cannot set SRL to polled mode\n" );
        exit( 1 );
    }

    /* open the device */
    if( ( dxxxxdev = dx_open( "dxxxxB1C1", 0 ) ) == -1 ){
        printf( "failed to open device\n" );
        exit( 1 );
    }

    /* Enable handlers */
    if( sr_enbhdr( dxxxxdev, EV_ANYEVT, dx_handler ) == -1 ){
        printf( "Could not enable handler: error = %s\n",
            ATDV_ERRMSGP( SRL_DEVICE ) );
        exit( 1 );
    }

    /* Generate events via async calls */
    if( dx_sethook( dxxxxdev, DL_ONHOOK, EV_ASYNC ) == -1 ){
        printf( "dx_sethook failed: error = %s\n", ATDV_ERRMSGP( dxxxxdev ) );
        exit( 1 );
    }

    /*
     * Wait forever while handlers deal with events
     * All handlers return 0 except the one for the last
     * event returns 1 telling SRL to leave the event to wake up
     * sr_waitevt().
     */
    (void)sr_waitevt( -1 );

    /* Cleanup */
}

```

## ■ Errors

None.

## ■ See Also

- `sr_waitevt()`
- *Voice Software Reference: Programmer's Guide for Linux, Appendix A*

**Name:** int sr\_getevtlen()  
**Inputs:** none  
**Returns:** length of data if successful  
          -1 if no current event  
**Includes:** srllib.h  
**Type:** Event Data Retrieval function

---

## ■ Description

The **sr\_getevtlen()** function returns the length of the variable data associated with the current event. If there is no current event pending, a value of -1 is returned.

**NOTE:** Information about the length of data returned by this function can be found in the *Voice Software Reference: Programmer's Guide for Linux, Appendix A*.

## ■ Cautions

If the program is executing a handler, the value returned is valid throughout the scope of the handler. If the program is not in a handler, the value returned is valid only until **sr\_waitevt()** is called again.

## ■ Example

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

int dx_handler()
{
    printf( "Got event with data length %d\n", sr_getevtlen());

    /* Tell SRL to keep the event */
    return( 1 );
}
```

```

main()
{
    int dxxxxdev;
    int mode = SR_POLLMODE;

    /* Set SRL to run in polled mode */
    if( sr_setparm( SRL_DEVICE, SR_MODEID, &mode ) == -1 ){
        printf( "Cannot set SRL to polled mode\n" );
        exit( 1 );
    }

    /* open the device */
    if(( dxxxxdev = dx_open( "dxxxB1C1", 0 ) ) == -1 ){
        printf( "failed to open device\n" );
        exit( 1 );
    }

    /* Enable handlers */
    if( sr_enbhdr( dxxxxdev, EV_ANYEVT, dx_handler ) == -1 ){
        printf( "Could not enable handler: error = %s\n",
            ATDV_ERRMSGP( SRL_DEVICE ) );
        exit( 1 );
    }

    /* Generate events via async calls */
    if( dx_sethook( dxxxxdev, DL_ONHOOK, EV_ASYNC ) == -1 ){
        printf( "dx_sethook failed: error = %s\n", ATDV_ERRMSGP( dxxxxdev ) );
        exit( 1 );
    }

    /*
     * Wait forever while handlers deal with events
     * All handlers return 0 except the one for the last
     * event returns 1 telling SRL to leave the event to wake up
     * sr_waitevt().
     */
    (void)sr_waitevt( -1 );

    /* Cleanup */
}

```

## ■ Errors

None.

## ■ See Also

- *Voice Software Reference: Programmer's Guide for Linux, Appendix A*

***sr\_getevtttype()***

***returns the event type for the current event***

---

**Name:** long sr\_getevtttype()  
**Inputs:** none  
**Returns:** Event type if successful  
-1 if no current event  
**Includes:** srllib.h  
**Type:** Event Data Retrieval function

---

## ■ Description

The **sr\_getevtttype()** function returns the event type for the current event. If no current event exists, a value of -1 is returned. This function returns SR\_TMOUDEVTT if a timeout occurs on SRL\_DEVICE. See also **sr\_waitevtt()**.

**NOTE:** Information about the device-specific event types that are returned by this function can be found in the *Voice Software Reference: Programmer's Guide for Linux, Appendix A*.

## ■ Cautions

If the program is executing a handler, the value returned is valid throughout the scope of the handler. If the program is not in a handler, the value returned is valid only until **sr\_waitevtt()** is called again.

## ■ Example

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

int dx_handler()
{
    printf( "Got event of type 0x%x\n", sr_getevtttype());

    /* Tell SRL to keep the event */
    return( 1 );
}
```

```

main()
{
    int dxxxxdev;
    int mode = SR_POLLMODE;

    /* Set SRL to run in polled mode */
    if( sr_setparm( SRL_DEVICE, SR_MODEID, &mode ) == -1 ){
        printf( "Cannot set SRL to polled mode\n" );
        exit( 1 );
    }

    /* open the device */
    if( ( dxxxxdev = dx_open( "dxxxxB1C1", 0 ) ) == -1 ){
        printf( "failed to open device\n" );
        exit( 1 );
    }

    /* Enable handlers */
    if( sr_enbhdr( dxxxxdev, EV_ANYEVT, dx_handler ) == -1 ){
        printf( "Could not enable handler: error = %s\n",
            ATDV_ERRMSGP( SRL_DEVICE ) );
        exit( 1 );
    }

    /* Generate events via async calls */
    if( dx_sethook( dxxxxdev, DL_ONHOOK, EV_ASYNC ) == -1 ){
        printf( "dx_sethook failed: error = %s\n", ATDV_ERRMSGP( dxxxxdev ) );
        exit( 1 );
    }

    /*
     * Wait forever while handlers deal with events
     * All handlers return 0 except the one for the last
     * event returns 1 telling SRL to leave the event to wake up
     * sr_waitevt().
     */
    (void)sr_waitevt( -1 );

    /* Cleanup */
}

```

## ■ Errors

None.

## ■ See Also

- `sr_waitevt()`
- the *Voice Software Reference: Programmer's Guide for Linux, Appendix A*

**Name:** int `sr_getfdcnt()`  
**Inputs:** none  
**Returns:** the number of *Linux* file descriptors  
**Includes:** `srlib.h`  
**Type:** Event data retrieval

## ■ Description

The `sr_getfdcnt()` function returns the total number of Linux file descriptors that are currently opened by the application. This number indicates how many valid descriptors exist out of all possible `OPEN_MAX` descriptors.

## ■ Example

```
#include <stdio.h>
#include <limits.h>
#include <malloc.h>

#include <srlib.h>

typedef unsigned long LONG;

main()
{
    int i;
    LONG *tmp;
    LONG *fdarray = (LONG*)NULL;
    LONG fdarray_size;          /* size of the fdarray */
    int nValidFileHandles = 0;

    /* sr_getfdcnt() returns the number of file descriptors that */
    /* have been opened by the application */
    /* the number returned indicates how many valid descriptors */
    /* exist of OPEN_MAX possible array entries */
    nValidFileHandles = sr_getfdcnt();

    /* If there were any valid file descriptors */
    if( nValidFileHandles > 0 )
    {
        /* the sr_getfdinfo() function call assumes the array size */
        /* passed in will be OPEN_MAX so we need to allocate that */
        /* much space (or have a static array of that size) */
        if(( fdarray = (LONG*)malloc( sizeof(LONG)*(OPEN_MAX+1) )) == (LONG*)NULL )
        {
            sr_setlasterr( SRL_DEVICE, ESR_SYS );
            /* errno set by malloc */
            printf("returning -1 on malloc fail\n");
            return( -1 );
        }
        fdarray_size = OPEN_MAX + 1;

        /* Fill in blanks with default i.e. -1 */
        for( i = 0; i < fdarray_size; i++ )
        {
```

```
        fdarray[ i ] = -1;
    }

    sr_getfdinfo( (int *)fdarray );

    /* nValidFileHandles should be printed out; */
    /* if any descriptor is -1, something is wrong */
    for(i=0; i<nValidFileHandles; i++)
        fprintf(stdout, "%d ", fdarray[i]);

    /* At this point, set up an array of pollfd structures */
    /* that are passed to the Linux poll() system call. */
    /* As events occur on the file descriptors, service */
    /* those events using usual callback or poll models in R4 */
}
else
    printf("No file descriptors have been opened!\n");
}
```

■ **See Also**  
**sr\_getfdinfo()**

***sr\_getfdinfo()*** *populates the fdarray argument with Linux file descriptors,*

---

**Name:** void **sr\_getfdinfo**(fdarray[ ])  
**Inputs:** int \* fdarray[ ]                      • file descriptor array  
**Returns:** none  
**Includes:** srllib.h  
**Type:** Event data retrieval

---

## ■ Description

The function **sr\_getfdinfo()** populates the **fdarray** argument with Linux file descriptors, which can then be passed to the Linux **poll()** function. Upon return, only the first *n* entries in **fdarray** are valid, where *n* is the value returned from **sr\_getfdcnt()**.

Parameter	Description
<b>fdarray</b>	Specifies the user allocated file descriptor array to be filled in by this function. Array should be initialized with -1.

## ■ Example

See the example in **sr\_getfdcnt()**.

## ■ See Also

- **sr\_getfdcnt()**



**■ Example**

```
#include <srllib.h>

main()
{
    int mode;

    if( sr_getparm( SRL_DEVICE, SR_MODEID, &mode ) == -1 ){
        printf( "Error: cannot set srl mode\n" );
        exit( -1 );
    }

    printf( "SRL is running in %s mode\n",
           mode == SR_POLLMODE ? "polled" : "signal" );
}
```

**■ Errors**

If the function returns a -1 to indicate an error, use **ATDV\_LASTERR()** to determine the reason for the failure. If the value returned is **ESR\_SYSTEM**, consult **errno** in *errno.h* for the following possible value:

**EINVAL**           • An invalid parameter was specified.

---

**Name:** int sr\_hold()  
**Inputs:** none  
**Returns:** 0 if successful  
**Includes:** srllib.h  
**Type:** Event Control function

---

## ■ Description

In signal mode, the **sr\_hold()** function holds off automatic notification of all events on all devices and subdevices to allow the user to protect critical regions of code by nesting pairs of **sr\_hold()** and **sr\_release()**. Events that occur are queued, but processes will not be interrupted by notification that the event has occurred. Also see **sr\_release()** in this section.

This function does not disconnect any events from event handlers or stop generation of events. Automatic event notification will resume when **sr\_release()** has been called once for each **sr\_hold()** in code, as demonstrated in the following example:

```

sr_hold( ) /* Events no longer reported */
.
.
    sr_hold( )
    .
    .
        sr_hold( )
        .
        .
            sr_release( ) /* All events still held */
            .
            .
                sr_release( ) /* All events still held */
                .
                .
                    sr_release( ) /* Events now reported */
                    .
                    .

```

**NOTES:** 1. **sr\_hold()** uses **sighold()** on SIGPOLL signals; therefore, all rules that apply to **sighold()** apply to this function. Refer to your Linux operating system documentation for information about **sighold()**

**NOTES:** 2. *srllib.h* must be the first header file included in code.

## ■ Cautions

Use **sr\_hold()** in signal mode only.

Take extreme care when using non-reentrant functions within event handlers.

**sr\_hold()** and **sr\_release()** can be used to protect these critical regions of code.

## ■ Example

```
#include <srllib.h>

int my_handler()
{
    ...
    ...malloc()...
    ...
    return( 0 );
}

main()
{
    int mode = SR_SIGMODE;

    /* Set SRL to run in signal mode */
    if( sr_setparm( SRL_DEVICE, SR_MODEID, &mode ) == -1 ){
        printf( "Cannot set SRL to signal mode\n" );
        exit( 1 );
    }
    ...
    ...
    /* enable handlers */
    if( sr_enbhdr( ..., ..., my_handler ) == -1 ){
        printf( "Cannot enable my_handler\n" );
        exit( 1 );
    }
    ...

    /* Call first async function which kicks everything off */
    ...

    /*
     * Sit in loop in main thread receiving events while doing further
     * processing.
     */
    while( 1 ){
        ...
        /* protect a critical region */
        if( sr_hold() == -1 ){
            printf( "sr_hold() failed\n" );
            exit( 1 );
        }
        ...malloc()...
        /* unprotect */
        if( sr_release() == -1 ){
            printf( "sr_release() failed\n" );
            exit( 1 );
        }
    }
    /* while */
}
/* main */
```

■ **Errors**

None.

■ **See Also**

- `sr_release()`

**Name:** int sr\_release()  
**Inputs:** none  
**Returns:** 0 if successful  
**Includes:** srllib.h  
**Type:** Event Control function

---

## ■ Description

If **sr\_hold()** is called only once, automatic notification of events resumes when **sr\_release()** is called. If **sr\_hold()** and **sr\_release()** pairs are nested to protect critical regions of code, automatic notification of events resume when **sr\_release()** has been called once for each **sr\_hold()** in code. Also see **sr\_hold()** in this section.

**NOTES:** 1. An **sr\_release()** must be used for each **sr\_hold()** to release held events.

**NOTES:** 2. **sr\_release()** uses **sigrelse()** on SIGPOLL signals; therefore, all rules that apply to **sigrelse()** apply to this function. Refer to your Linux operating system documentation for information about **sigrelse()**.

**NOTES:** 3. *srllib.h* must be the first header file included in code.

## ■ Cautions

**sr\_release()** is used in signal mode only.

Extreme care should be taken when using non-reentrant functions within event handlers. **sr\_hold()** and **sr\_release()** can be used to protect these critical regions of code.

## ■ Example

```
#include <srllib.h>

int my_handler()
{
    ...
    ...malloc()...
    ...
    return( 0 );
}
```

```

main()
{
    int mode = SR_SIGMODE;

    /* Set SRL to run in signal mode */
    if( sr_setparm( SRL_DEVICE, SR_MODEID, &mode ) == -1 ){
        printf( "Cannot set SRL to signal mode\n" );
        exit( 1 );
    }
    ...
    ...
    /* enable handlers */
    if( sr_enbhdr( ..., ..., my_handler ) == -1 ){
        printf( "Cannot enable my_handler\n" );
        exit( 1 );
    }

    ...

    /* Call first async function which kicks everything off */
    ...

    /*
     * Sit in loop in main thread receiving events while doing further
     * processing.
     */
    while( 1 ){
        ...
        /* protect a critical region */
        if( sr_hold() == -1 ){
            printf( "sr_hold() failed\n" );
            exit( 1 );
        }
        ...malloc()...
        /* unprotect */
        if( sr_release() == -1 ){
            printf( "sr_release() failed\n" );
            exit( 1 );
        }
    }
    /* while */
}
/* main */

```

## ■ Errors

None.

## ■ See Also

- `sr_hold()`

## ***sr\_setparm()***

*allows the application to change the value*

---

<b>Name:</b>	int sr_setparm(dev, parmno, valuep)
<b>Inputs:</b>	int dev <ul style="list-style-type: none"><li>• Device</li></ul> unsigned long parmno <ul style="list-style-type: none"><li>• Parameter number</li></ul> void *valuep <ul style="list-style-type: none"><li>• Parameter value</li></ul>
<b>Returns:</b>	0 if successful -1 if failure
<b>Includes:</b>	srllib.h
<b>Type:</b>	SRL Parameter function

---

### ■ Description

The **sr\_setparm()** function allows the application to change the value of an SRL parameter. The function parameters are described as follows:

<b>Parameter</b>	<b>Description</b>
<b>dev:</b>	The value is always set to SRL_DEVICE when setting SRL parameters.
<b>parmno:</b>	specifies the value for SRL parameter to be changed. Possible values are as follows: <ul style="list-style-type: none"><li>• SR_MODEID set event notification mode parameter to signal (SR_SIGMODE, the default) or non-signal (SR_POLLMODE)</li><li>• SR_INTERPOLLID set the polling granularity parameter (the time between device polls expressed in milliseconds)</li></ul>
	<b>NOTE:</b> If you are using the Synchronous programming model, set SR_INTERPOLLID to MAX_INT for optimal performance.
<b>valuep:</b>	a pointer to an area of memory that contains the value for the specified parameter. For SR_MODEID, the value is expected to point to an integer that contains SR_SIGMODE or SR_POLLMODE. For SR_INTERPOLLID, the value is expected to point to an integer that contains the polling granularity expressed in millisecond (ms) units.

## ■ Cautions

`dev` should always be set to `SRL_DEVICE` when setting SRL parameters.

## ■ Example

```
#include <srllib.h>

main()
{
    int mode = SR_POLLMODE;

    if( sr_setparm( SRL_DEVICE, SR_MODEID, &mode ) == -1 ){
        printf( "Error: cannot set srl mode\n" );
        exit( 1 );
    }
}
```

## ■ Errors

If the function returns a -1 to indicate an error, use `ATDV_LASTERR()` to determine the reason for the failure. If the value returned is `ESR_SYSTEM`, consult **errno** in `errno.h` for the following possible value:

- `EINVAL`      • An invalid parameter was specified.

**sr\_waitevt()** *allows events to be handled using the Callback model*

---

**Name:** long sr\_waitevt(timeout)  
**Inputs:** long timeout      • Timeout in milliseconds (msec)  
**Returns:** time left before timeout  
          -1 if timeout  
**Includes:** srllib.h  
**Type:** Event Control function

---

■ **Description**

The **sr\_waitevt()** function allows events to be handled using the Callback model (Non-Signal or Signal) or the Polled model. With the Callback model, it allows event handlers to be called. With the Polled model, it allows polling for and handling events.

**sr\_waitevt()** returns when next event occurs or a timeout is reached. If an event occurred, the value returned is the number of milliseconds before the timeout. A value of -1 is returned if no event is found before the timeout. If the function times out, a timeout event occurs on the SRL\_DEVICE (i.e. **sr\_getevtdev()** returns SRL\_DEVICE and the event type returned by **sr\_getevttype()** is SR\_TMOUTEVT).

A timeout value of -1 instructs **sr\_waitevt()** to wait indefinitely for the next event.

■ **Cautions**

When an event is received, it must be handled immediately and event-specific information should be retrieved before the next call to **sr\_waitevt()**. This is because **sr\_waitevt()** automatically removes the current event before waiting for the next event. As a result, if the event is not handled immediately or if event-specific information is not retrieved, it is lost.

**sr\_waitevt()** cannot be called from within a handler.

Note that the current implementation of **sr\_waitevt()** uses the Linux **time()** function, which only has a granularity of 1 second.

## ■ Example

```

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

int dx_handler()
{
    printf( "Got event 0x%x on device %s, data length = %d, datap = 0x%x\n",
           sr_getevttype(), ATDV_NAMEP( sr_getevtdev()), sr_getevtlen(),
           sr_getevtdatap() );

    /* Tell SRL to keep the event */
    return( 1 );
}

main()
{
    int dxxxdev;
    int mode = SR_POLLMODE;

    /* Set SRL to run in polled (non-signal) mode */
    if( sr_setparm( SRL_DEVICE, SR_MODEID, &mode ) == -1 ){
        printf( "Error: cannot set srl mode\n" );
        exit( 1 );
    }

    /* Open a dxxx channel device */
    if( ( dxxxdev = dx_open( "dxxxB1C1", 0 ) ) == -1 ){
        printf( "Error: cannot open device\n" );
        exit( 1 );
    }

    /* enable a handler for all events on the device */
    if( sr_enbhdlr( dxxxdev, EV_ANYEVT, dx_handler ) == -1 ){
        printf( "Error: could not enable handler\n" );
        exit( 1 );
    }

    /* Perform an async function on the device */
    if( dx_sethook( dxxxdev, DL_ONHOOK, EV_ASYNC ) == -1 ){
        printf( "dx_sethook failed: error = %s\n", ATDV_ERRMSGP( dxxxdev ) );
        exit( 1 );
    }

    /* Wait 10 seconds for an event */
    if( sr_waitevt( 10000 ) == -1 ){
        printf( "sr_waitevt, %s\n",
              ATDV_ERRMSGP( SRL_DEVICE ) );
        exit( 1 );
    }
}

```

## ***sr\_waitvt()***

***allows events to be handled using the Callback model***

---

```
/* Disable the handler */
if( sr_dishdlr( dxxxdev, EV_ANYEVT, dx_handler ) == -1 ){
    printf( "Error: could not disable handler\n" );
    exit( 1 );
}

if( dx_close( dxxxdev ) == -1 ){
    printf( "Error: could not close dxxxdev\n" );
    exit( 1 );
}

exit( 0 );
}
```

## ■ Errors

If the function returns a -1 to indicate an error, use **ATDV\_LASTERR()** to determine the reason for the failure. If the value returned is **ESR\_SYSTEM**, consult **errno** in *errno.h* for the following possible value:

**EINVAL**           • Invalid timeout value.

*allows events to be handled using the Callback model*

*sr\_waitevt()*

---



## 4. Standard Attribute Functions

---

Standard Attribute functions are contained in the Dialogic Standard Runtime Library for Linux (SRL). They return general information that exists for all Dialogic devices, such as the device name and the error that occurred on the last library call.

All Standard Attribute function names adhere to the following naming conventions:

- The function name is all capital letters.
- The function name is prefixed by “ATDV\_.”
- The name after the underscore describes the attribute.

The Standard Attribute functions and the information they return are listed below.

<b>ATDV_ERRMSGP()</b>	• pointer to string describing error on last library call
<b>ATDV_IOPORT()</b>	• base address of I/O port
<b>ATDV_IRQNUM()</b>	• interrupt being used
<b>ATDV_LASTERR()</b>	• error that occurred on last device library call
<b>ATDV_NAMEP()</b>	• pointer to device name
<b>ATDV_SUBDEVS()</b>	• number of subdevices

### 4.1. Include Files

The following lines must be included in application code prior to calling any Standard Attribute functions:

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

*DEVICElib.h* is the header file for the device being used. For example, if the device is a DTI/240SC board, the *dtilib.h* file is included.

**NOTE:** *srllib.h* must be included in code before all other Dialogic header files.

## **4.2. Standard Attribute Function Reference**

The information returned by the functions described in the following pages is for the device specified in the Standard Attribute function call.

Refer to the *Voice Software Reference: Programmer's Guide for Linux, Appendix A*, for a list of information returned for specific devices.

**Name:** char \* ATDV\_ERRMSGP(dev)  
**Inputs:** int dev • Valid Dialogic device handle  
**Returns:** pointer to string  
**Includes:** srlib.h  
**Category:** Standard Attribute

---

## ■ Description

The **ATDV\_ERRMSGP()** function returns a pointer to an ASCIIZ string that describes the error that occurred on the device during the last device library call. This pointer remains valid throughout the execution of the application. If no error occurred on the device during the last function call, the string pointed to is “No error.”

The function parameters are described as follows:

<b>Parameter</b>	<b>Description</b>
<b>dev:</b>	specifies the valid device handle obtained when the device was opened using <b>xx_open()</b> , where <i>xx</i> is the prefix identifying the device to be opened. Specify <b>EV_ANYDEV</b> to be notified of an event on any device.

The *Voice Software Reference: Programmer's Guide for Linux, Appendix A*, lists further device-specific information about this function.

## ■ Cautions

None.

## ■ Example

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

main()
{
    int dxxxdev;
    int parm = ET_RON;

    /* Open dxxx channel device */
    if(( dxxxdev = dx_open( "dxxxBlCl", 0 )) == -1 ){
        printf( "Error: cannot open device\n" );
        exit( 1 );
    }

    /*Attempt to set a board level parameter on a channel device-will fail */
    if( dx_setparm( dxxxdev, DXBD_R_EDGE, &parm ) == -1 ){
        printf( "The last error on the device was '%s'\n",
            ATDV_ERRMSGP( dxxxdev ) );
    }
}
```

## ■ Errors

This function returns a pointer to the string “Unknown device” if an invalid device handle is specified in **dev**.

## ■ See Also

- *Voice Software Reference: Programmer’s Guide for Linux, Appendix A*

**Name:** long ATDV\_IOPORT(dev)  
**Inputs:** int dev      • Valid Dialogic device handle  
**Returns:** AT\_FAILURE - failure  
Base Port Address of device  
**Includes:** srlib.h  
**Category:** Standard Attribute

---

### ■ Description

The **ATDV\_IOPORT()** function returns the base port address used by the device.

The function parameters are described as follows:

<b>Parameter</b>	<b>Description</b>
<b>dev:</b>	specifies the valid device handle obtained when the device was opened using <b>xx_open()</b> , where <i>xx</i> is the prefix identifying the device to be opened. Specify <b>EV_ANYDEV</b> to be notified of an event on any device.

The *Voice Software Reference: Programmer's Guide for Linux, Appendix A*, lists further device-specific information about this function.

### ■ Cautions

None.

## ■ Example

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

main()
{
    int dtiddd;

    /* Open a dti timeslot */
    if(( dtiddd = dt_open( "/dev/dtiB1T1", 0 )) == -1 ){
        printf( "Error: cannot open dti timeslot device\n" );
        exit( 1 );
    }

    printf( "I/O port is at 0x%x\n", ATDV_IOPORT( dtiddd ));
}
```

## ■ Errors

If the device does not use I/O ports, or if an invalid device handle is specified in **dev**, this function fails and returns the value defined by **AT\_FAILURE**.

## ■ See Also

- *Voice Software Reference: Programmer's Guide for Linux, Appendix A*

**Name:** long ATDV\_IRQNUM(dev)  
**Inputs:** int dev      • Valid Dialogic device handle  
**Returns:** AT\_FAILURE - failure  
           IRQ of device  
**Includes:** srllib.h  
**Category:** Standard Attribute

## ■ Description

The **ATDV\_IRQNUM()** function returns the interrupt number (IRQ) used by the device.

The function parameters are described as follows:

Parameter	Description
<b>dev:</b>	specifies the valid device handle obtained when the device was opened using <b>xx_open()</b> , where <i>xx</i> is the prefix identifying the device to be opened. Specify <b>EV_ANYDEV</b> to be notified of an event on any device.

## ■ Cautions

None.

## ■ Example

```

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

main()
{
    int dxxxdev;

    /* Open a dxxx channel device */
    if(( dxxxdev = dx_open( "dxxxB1C1", 0 )) == -1 ){
        printf( "Error: cannot open device\n" );
        exit( 1 );
    }

    printf( "Device irq is %d\n", ATDV_IRQNUM( dxxxdev ) );
}

```

*returns the interrupt number (IRQ)*

*ATDV\_IRQNUM()*

---

## ■ Errors

This function returns the value defined by `AT_FAILURE` if the device has no IRQ number or an invalid device handle is specified in `dev`.

**ATDV\_LASTERR()**      *returns a long that indicates the error that occurred*

---

**Name:** long ATDV\_LASTERR(dev)  
**Inputs:** int dev      • Valid Dialogic device handle  
**Returns:** EDV\_BADDESC - invalid device handle  
            error number  
**Includes:** srllib.h  
**Category:** Standard Attribute

---

### ■ Description

The **ATDV\_LASTERR()** function returns a long that indicates the error that occurred on the device during the last device library call. The errors are defined in *DEVICELib.h* of the specified device.

If no errors occurred during the last device library call on this device, the return value is 0.

The function parameters are described as follows:

<b>Parameter</b>	<b>Description</b>
<b>dev:</b>	specifies the valid device handle obtained when the device was opened using <b>xx_open()</b> , where <i>xx</i> is the prefix identifying the device to be opened. Specify <b>EV_ANYDEV</b> to be notified of an event on any device.

The *Voice Software Reference: Programmer's Guide for Linux, Appendix A*, lists further device-specific information about this function.

### ■ Cautions

None.

## ■ Example

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

main()
{
    int dxxxdev;
    int parm = ET_RON;

    /* Open dxxx channel device */
    if(( dxxxdev = dx_open( "dxxxBlCl", 0 )) == -1 ){
        printf( "Error: cannot open device\n" );
        exit( 1 );
    }

    /*Attempt to set a board level parameter on a channel device-will fail */
    if( dx_setparm( dxxxdev, DXBD_R_EDGE, &parm ) == -1 ){
        printf( "The last error on the device was 0x%x\n",
                ATDV_LASTERR( dxxxdev ) );
    }
}
```

## ■ Errors

This function returns EDV\_BADDESC if an invalid device handle is specified in **dev**.

## ■ See Also

- *Voice Software Reference: Programmer's Guide for Linux, Appendix A*

**Name:** char \* ATDV\_NAMEP(dev)  
**Inputs:** int dev                   • Valid Dialogic device handle  
**Returns:** pointer to string  
**Includes:** srlib.h  
**Category:** Standard Attribute

---

## ■ Description

The **ATDV\_NAMEP()** function returns a pointer to an ASCIIZ string that specifies the device name contained in the configuration file. The name specified is the name used to open the device.

Examples of device names are:

dxxxBbCc  
dtiBbTt

where

*b* is the number of the board in the system  
*c* is the number of the channel on the real or emulated D/4x board  
*t* is the number of the time slot in the DTI/xxx, D/xxxSC-T1, or D/xxxSC-E1 system

The pointer to this string remains valid only while the device is open.

The function parameters are described as follows:

<b>Parameter</b>	<b>Description</b>
<b>dev:</b>	specifies the valid device handle obtained when the device was opened using <b>xx_open()</b> , where <i>xx</i> is the prefix identifying the device to be opened. Specify <b>EV_ANYDEV</b> to be notified of an event on any device.

The *Voice Software Reference: Programmer's Guide for Linux, Appendix A*, lists further device-specific information about this function.

## ■ Cautions

None.

## ■ Example

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

main()
{
    int dxxxdev;

    /* Open a dxxx channel device */
    if(( dxxxdev = dx_open( "dxxxB1C1", 0 )) == -1 ){
        printf( "Error: cannot open device\n" );
        exit( 1 );
    }

    printf( "Device name is %s\n", ATDV_NAMEP( dxxxdev ) );
}
```

## ■ Errors

This function returns a pointer to the string “Unknown device” if an invalid device handle is specified in **dev**.

## ■ See Also

- *Voice Software Reference: Programmer’s Guide for Linux, Appendix A*

## **ATDV\_SUBDEVS()**

*returns the number of subdevices for the device*

---

**Name:** long ATDV\_SUBDEVS(dev)  
**Inputs:** int dev                   • Valid Dialogic device handle  
**Returns:** AT\_FAILURE - failure  
          number of subdevices  
**Includes:** srllib.h  
**Category:** Standard Attribute

---

### ■ Description

The **ATDV\_SUBDEVS()** function returns the number of subdevices for the device. This number is returned as an integer.

Examples of subdevices are time slots on a DTI/xxx board and channels on a D/4x virtual board.

The function parameters are described as follows:

<b>Parameter</b>	<b>Description</b>
<b>dev:</b>	specifies the valid device handle obtained when the device was opened using <b>xx_open()</b> , where <i>xx</i> is the prefix identifying the device to be opened. Specify <b>EV_ANYDEV</b> to be notified of an event on any device.

The number of subdevices specified for Dialogic devices can be found in the *Voice Software Reference: Programmer's Guide for Linux, Appendix A*.

### ■ Cautions

None.

## ■ Example

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

main()
{
    int dxxxdev;

    /* Open a dxxx channel device */
    if(( dxxxdev = dx_open( "dxxxB1", 0 )) == -1 ){
        printf( "Error: cannot open device\n" );
        exit( 1 );
    }

    printf( "Device has %d subdevices\n", ATDV_SUBDEVS( dxxxdev ));
}
```

## ■ Errors

This function fails and returns the value defined by `AT_FAILURE` if an invalid device handle is specified in `dev`.

## ■ See Also

*Voice Software Reference: Programmer's Guide for Linux, Appendix A*

## 5. SRL Device Mapper Functions

---

SRL Device Mapper functions are contained in the SRL Device Mapper Library Interface (SDM API), a subset of the Dialogic Standard Runtime Library for Linux (SRL). They return information about the structure of the system, such as a list of all the virtual boards on a physical board. The SDM API works for any component that exposes R4 devices.

The SRL Device Mapper functions return information about the structure of the system based upon the following conventions:

- A physical board *owns* zero or more virtual boards.
- A virtual board *owns* zero or more sub devices.
- A virtual board *is* an R4 device.
- A sub device *is* an R4 device.
- One or more jacks *are associated with* one or more R4 devices.

The SRL Device Mapper functions and the information they return are listed below.

<b>SRLGetAllPhysicalBoards( )</b>	• retrieves a list of all physical boards in a node
<b>SRLGetVirtualBoardsOnPhysicalBoard( )</b>	• retrieves a list of all virtual boards on a physical board
<b>SRLGetSubDevicesOnVirtualBoard( )</b>	• retrieves a list of all sub devices on a virtual board
<b>SRLGetJackForR4Device( )</b>	• retrieves the jack number of an R4 device

**NOTE:** The SDM 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 sub devices on a physical board, combine multiple SRL Device Mapper functions.

## **5.1. Include Files**

The following line must be included in application code prior to calling any SRL Device Mapper functions:

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

**NOTE:** *srllib.h* must be included in code before all other Dialogic header files.

## **5.2. SRL Device Mapper Function Reference**

This section provides a detailed description of the SRL Device Mapper functions in alphabetical order.



*returns a list of all the physical boards*

*SRLGetAllPhysicalBoards()*

---

### ■ Cautions

If you issue a call using \*piNum returned from a previous call, this subsequent call may not succeed if the system topology changed between calls.

**■ Example**

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

AUID *pAU;
int iNumPhyBds;
long retVal;

iNumPhyBds = 0;
pAU = 0;
do
{
    free(pAU);
    pAU = iNumPhyBds ? (AUID *)malloc(iNumPhyBds * sizeof(*pAU)) : 0;
    retVal = SRLGetAllPhysicalBoards(&iNumPhyBds, pAU);
} while (ESR_INSUFBUF == retVal);
if (ESR_NOERR != retVal)
{ // do some error handling
    ...
}
```

**■ Errors**

The user of this function is responsible for allocating an array of AUIDs (pPhysicalBoards) of length \*piNum. If \*piNum is insufficient, **SRLGetAllPhysicalBoards()** returns ESR\_INSUFBUF.

If the return code is ESR\_NOERR or ESR\_INSUFBUF, \*piNum on output is the actual number of physical boards.

**■ See Also**

**SRLGetVirtualBoardsOnPhysicalBoard()**

*returns the jack number*

**SRLGetJackForR4Device( )**

---

**Name:** long SRLGetJackForR4Device(**IN** \*szR4Device  
**OUT** \*piJackNum)  
**Inputs:** char \*szR4Device • R4 device  
int \*piJackNum • pointer to jack number  
**Returns:** ESR\_NOERR - indicates success  
ESR\_BADDEV - indicates that this R4 device does not have  
an associated jack  
Other SRL error  
**Includes:** srllib.h  
**Category:** SRL Device Mapper

---

## ■ Description

The **SRLGetJackForR4Device( )** function returns the jack number associated with the R4 device. Each jack number is unique on a physical board.

The user of this function is responsible for allocating the integer value required for writing the jack number.

The function parameters are described as follows:

<b>Parameter</b>	<b>Description</b>
*szR4Device	the R4 device to be used for this function call.
*piJackNum	pointer to the jack number associated with the R4 device.

## ■ Cautions

None.

## ■ Example

```
#include <srllib.h>
...
char szVB[] = "dtiB7";
long retVal;
int iJack;
```

## **SRLGetJackForR4Device()**

*returns the jack number*

---

```
retVal = GetJackForR4Device(szVB, &iJack);
if (retVal != ESR_NOERR)
{ // do some error handling
...
}
```

### ■ Errors

**SRLGetJackForR4Device()** returns `ESR_BADDEV` if the R4 device does not have an associated jack, or if the R4 device is associated with multiple jacks.

### ■ See Also

**SRLGetVirtualBoardsOnPhysicalBoard()**,  
**SRLGetSubDevicesOnVirtualBoard()**



The SRLDEVICEINFO structure is defined as follows:

```
typedef struct _tagSrlDeviceInfo
{
    char szDevName[12]; /* Name of the device */
    int iDevType;      /* Device type - see ATDV_DEVICEYPE */
} SRLDEVICEINFO, *LPSRLDEVICEINFO;
```

## ■ Cautions

If you issue a call using \*piNum returned from a previous call, this subsequent call may not succeed if the system topology changed between calls.

## ■ Example

```
#include <srllib.h>
...
char szVB[] = "dtiB7";
SRLDEVICEINFO *pVBInfo;
int iNumSDs;
long retVal;

iNumSDs = 0;
pSDInfo = 0;
do
{
    free(pSDInfo);
    pSDInfo = iNumSDs;
    ? (SRLDEVICEINFO *)malloc(iNumSDs * sizeof(*pSDInfo)) : 0;
    retVal = SRLGetSubDevicesOnVirtualBoard(phyBd, &iNumSDs, pSDInfo);
} while (ESR_INSUFBUF == retVal);
if (ESR_NOERR != retVal)
{ // do some error handling
    ...
}
```

## ■ Errors

The user of this function is responsible for allocating an array of SRLDEVICEINFOs (pInfo) of length \*piNum. If \*piNum is insufficient, **SRLGetSubDevicesOnVirtualBoard()** returns ESR\_INSUFBUF.

If the return code is ESR\_NOERR or ESR\_INSUFBUF, \*piNum on output is the actual number of sub devices.

*returns a list of sub devices*

*SRLGetSubDevicesOnVirtualBoard()*

---

■ **See Also**

SRLGetVirtualBoardsOnPhysicalBoard(), SRLGetJackForR4Device()



The SRLDEVICEINFO structure is defined as follows:

```
typedef struct _tagSrlDeviceInfo
{
    char szDevName[12]; /* Name of the device */
    int iDevType;      /* Device type - see ATDV_DEVICETYPE */
} SRLDEVICEINFO, *LPSRLDEVICEINFO;
```

## ■ Cautions

If you issue a call using \*piNum returned from a previous call, this subsequent call may not succeed if the system topology changed between calls.

## ■ Example

```
#include <srllib.h>
...
AUID phyBd;
SRLDEVICEINFO *pVBInfo;
int iNumVBs;
long retVal;

... // phyBd assigned
iNumVBs = 0;
pVBInfo = 0;
do
{
    free(pVBInfo);
    pVBInfo = iNumVBs;
    ? (SRLDEVICEINFO *)malloc(iNumVBs * sizeof(*pVBInfo)) : 0;
    retVal = SRLGetVirtualBoardsOnPhysicalBoard(phyBd, &iNumVBs, pVBInfo);
} while (ESR_INSUFBUF == retVal);
if (ESR_NOERR != retVal)
{ // do some error handling
    ...
}
```

## ■ Errors

The user of this function is responsible for allocating an array of SRLDEVICEINFOs (pInfo) of length \*piNum. If \*piNum is insufficient, **SRLGetVirtualBoardsOnPhysicalBoard()** returns ESR\_INSUFBUF.

If the return code is ESR\_NOERR or ESR\_INSUFBUF, \*piNum on output is the actual number of virtual boards.

***SRLGetVirtualBoardsOnPhysicalBoard()***      *returns a list of virtual boards*

---

■ **See Also**

**SRLGetAllPhysicalBoards()**, **SRLGetSubDevicesOnVirtualBoard()**,  
**SRLGetJackForR4Device()**

*returns a list of virtual boards*      *SRLGetVirtualBoardsOnPhysicalBoard()*

---

# Appendix A

---

## Related Publications

For more information on related hardware and software products see the following Dialogic publications:

- For information about the voice programming libraries, see *Voice Software Reference: Programmer's Guide for Linux*
- For definitions and descriptions of various features of Dialogic voice hardware and software, see *Voice Software Reference: Features Guide for Linux*
- For information on fax software development, see the *Fax Software Reference for Linux*
- For descriptions of the network programming libraries, see  
*Digital Network Interface Software Reference*  
*Digital Audio Conferencing Software Reference*  
*MSI/SC Software Reference*
- For information on GlobalCall software development, see:  
*GlobalCall API Software Reference*  
*GlobalCall ISDN Technology User's Guide*  
*GlobalCall E-1/T-1 Technology User's Guide*  
*GlobalCall Country-Dependent Parameters (CDP) Reference*  
*GlobalCall Country-Dependent Parameters Reference for PDK Protocols*  
*GlobalCall DPNSS ISDN Protocol Reference*  
*GlobalCall Analog Technology User's Guide*
- For information on ISDN software development, see the *ISDN Software Reference*
- For information on SCbus routing, see:  
*SCbus Routing Guide*  
*SCbus Routing Software Reference for Linux*
- For information on Dialogic boards, see also the respective *Quick Installation Card*.



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

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

**Dialogic Standard Runtime Library (SRL) for Linux:** Device-independent library that consists of Event Management functions and Standard Attribute functions.

**Event:** Any message sent from the device.

**Handler:** A user-defined function called by the SRL when a specified event occurs on a specified event.

**Event Management functions:** SRL 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.

**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."

**SRL Device Mapper functions:** SRL Device Mapper functions are contained in the SRL Device Mapper Library Interface (SDM API), a subset of the Dialogic Standard Runtime Library for Linux (SRL). They return information about the structure of the system, such as a list of all the virtual boards on a physical board. The SDM API works for any component that exposes R4 devices.

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

**Subdevices:** 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, e.g., silence-on or silence-off events on a channel.

# Index

---

## A

- Application Development models
  - asynchronous programming, 3
  - Callback model, 7
  - invalid model combinations, 13
  - model combinations, 12
  - polled model, 5
  - synchronous programming, 3
  - types, 3
- applications, linking, 13
- Asynchronous models
  - callback, 4, 7
  - polled, 4, 5
- Asynchronous programming, 3
- ATDV\_ERRMSGP(), 57, 58
- ATDV\_IOPORT(), 59, 60
- ATDV\_IRQNUM(), 61, 62
- ATDV\_LASTERR(), 63, 64
- ATDV\_NAMEP(), 65, 66
- ATDV\_SUBDEVS(), 67, 68

## C

- Callback model
  - asynchronous, 4
  - cbansr, 11
  - event handlers, 7
  - modes, 7
  - non-signal, 8
  - non-signal mode, 4
  - signal, 9
  - signal mode, 4
- cbansr, 11

- codes, error, 20
- compiling applications, 13

## D

- device name pointer, 65
- Dialogic Standard Runtime Library for Linux (SRL)
  - library contents, 1
  - major function of, 1
  - overview, 1
  - SRL\_DEVICE, 2
  - Standard Attribute functions, 68
- DV\_TPT data structure, 2

## E

- error codes, 20
- error handling, 19, 20
- error, on last library call, 63
- event control functions, 17, 18
- Event Data Retrieval Functions, 19
- event handlers
  - backup event handlers, 17
  - definition, 7
  - disabling, 22
  - enabling, 25
  - enabling and disabling, 17
  - guidelines, 7
  - hierarchy, 8, 18
    - device non-specific/event non-specific, 8, 18, 26
    - device non-specific/event specific, 8, 18, 26
    - device specific/event non-specific, 8, 18, 26

## Voice Software Reference - Standard Runtime Library for Linux

- device/event specific, 8, 18, 26
  - in signal mode, 26
  - reenetrancy, 27
  - reenetrancy protection, 43, 46
  - reenetry, 10
  - rules, 26
- Event Management functions
- error codes, 20
  - error handling, 19, 20
  - event control functions:, 17, 18
  - include files, 20
  - overview, 1, 17, 20
  - reference, 21
  - sr\_dishdlr(), 22, 24
  - sr\_enbhdlr(), 25, 29
  - sr\_getevtdatap(), 30, 31
  - sr\_getevtdev(), 32, 33
  - sr\_getevtlen(), 34, 35
  - sr\_getevttype(), 36, 37
  - sr\_getparm(), 41
  - sr\_hold(), 43, 45
  - sr\_release(), 46, 47
  - sr\_setparm(), 48, 49
- event notification
- modes, 18
  - non-signal mode, 18
  - signal mode, 18
- events
- data extraction, 30, 32, 34, 36
  - data retrieval, 6, 50
  - polling for and handling, 50
- Examples
- non-signal callback model, 9
  - polled model, 6
  - Signal callback model, 10, 11
- F**
- functions
- Event Data Retrieval, 19
  - Event Management
    - error codes, 20
    - error handling, 19, 20
    - event control functions, 17, 18
    - include files, 20
    - overview, 1, 17, 20
    - reference, 21
    - sr\_dishdlr(), 22, 24
    - sr\_enbhdlr(), 25, 29
    - sr\_getevtdatap(), 30, 31
    - sr\_getevtdev(), 32, 33
    - sr\_getevtlen(), 34, 35
    - sr\_getevttype(), 36, 37
    - sr\_getparm(), 41
    - sr\_hold(), 43, 45
    - sr\_release(), 46, 47
    - sr\_setparm(), 48, 49
  - SRL Device Mapper
    - include files, 70
    - overview, 70
    - SRLGetAllPhysicalBoards(), 71, 73
    - SRLGetJackForR4Device(), 74, 75
    - SRLGetSubDevicesOnVirtualBoard(), 76, 78
    - SRLGetVirtualBoardsOnPhysicalBoard(), 79, 81
  - SRL Device Mapper
    - conventions, 69
    - overview, 2, 69
    - reference, 70
  - SRL parameter, 19
  - Standard Attribute, 68
    - ATDV\_ERRMSGP(), 57, 58
    - ATDV\_IOPORT(), 59, 60
    - ATDV\_IRQNUM(), 61, 62
    - ATDV\_LASTERR(), 63, 64
    - ATDV\_NAMEP(), 65, 66
    - ATDV\_SUBDEVS(), 67, 68
    - include files, 55
    - naming conventions, 55
    - overview, 2, 55, 56
    - reference, 56, 68

**H**

handlers. *See* event handlers

**I**

Include Files, 20, 55, 70

interrupt number, 61

Invalid model combinations, 13

IRQ, 61

**L**

libdxxx.a, 13

libraries, linking, 13

libsrl.a, 13

linking libraries, 13

**M**

mode, signal and non-signal, 18

Model combinations

- polled/non-signal callback, 12
- polled/synchronous, 13
- polled/synchronous/callback, 13
- synchronous/callback, 12
- synchronous/polled, 12
- synchronous/polled/non-signal callback, 12
- valid model combinations, 12

**N**

Non-signal callback model, 8, 9

non-signal mode, 18

**P**

pansr, 6

physical boards, retrieving a list of all,  
71

Polled model

- asynchronous, 4
- data retrieval, 6
- description, 5
- example, 6
- pansr, 6

Polled/non-signal callback model

- combinations, 12

Polled/synchronous model

- combinations, 13

Polled/synchronous/callback model

- combinations, 13

**R**

R4 device, retrieving the jack number,  
74

reentrancy protection, 43, 46

reentry protection, 11

related publications, 83

**S**

SDM API

- conventions, 69
- overview, 2, 69

Signal Callback model

- event handlers, 7
- event notification, 9
- example, 10
- overview, 9
- reentry, 10
- reentry protection example, 11

signal mode, 18

SIGPOLL, 10

SIGPOLL signal, 18

sr\_dishdlr( ), 22, 24

sr\_enbhdlr( ), 25, 29

## Voice Software Reference - Standard Runtime Library for Linux

- sr\_getevtdatap(), 30, 31
  - sr\_getevtdev(), 32, 33
  - sr\_getevtlen(), 34, 35
  - sr\_getevttype(), 36, 37
  - sr\_getfdcnt(), 38
  - sr\_getfdinfo(), 40
  - sr\_getparm(), 41
  - sr\_hold(), 43, 45
  - sr\_release(), 46, 47
  - sr\_setparm(), 48, 49
  - sr\_waitevt(), 50
  - SRL Device Mapper functions
    - conventions, 69
    - include files, 70
    - overview, 2, 69, 70
    - reference, 70
    - SRLGetAllPhysicalBoards(), 71, 73
    - SRLGetJackForR4Device(), 74, 75
    - SRLGetSubDevicesOnVirtualBoard(), 76, 78
    - SRLGetVirtualBoardsOnPhysicalBoard(), 79, 81
  - SRL parameters
    - functions, 19
    - retrieving, 41
    - setting, 48
  - SRL\_DEVICE, 2
  - SRLGetAllPhysicalBoards(), 71, 73
  - SRLGetJackForR4Device(), 74, 75
  - SRLGetSubDevicesOnVirtualBoard(), 76, 78
  - SRLGetVirtualBoardsOnPhysicalBoard(), 79, 81
  - Standard Attribute functions
    - ATDV\_ERRMSGP(), 57, 58
    - ATDV\_IOPORT(), 59, 60
    - ATDV\_IRQNUM(), 61, 62
    - ATDV\_LASTERR(), 64
    - ATDV\_NAMEP(), 65, 66
    - ATDV\_SUBDEVS(), 67, 68
    - include files, 55
    - naming conventions, 55
    - overview, 2, 55, 56
    - reference, 56, 68
    - sub devices, retrieving a list, 76
    - subdevices, retrieving number of, 67
  - Synchronous model, 4
  - Synchronous programming, 3
  - Synchronous/callback model
    - combinations, 12
  - Synchronous/pollled model
    - combinations, 12
  - Synchronous/pollled/non-signal callback
    - model combinations, 12
- ## T
- Termination Parameter Table, 2
- ## V
- Valid model combinations
    - polled/non-signal callback, 12
    - polled/synchronous, 13
    - polled/synchronous/callback, 13
    - synchronous/callback, 12
    - synchronous/pollled, 12
    - synchronous/pollled/non-signal callback, 12
  - virtual boards, retrieving a list, 79

