Dialogic®

Dialogic® Standard Runtime Library API

Library Reference

May 2008

05-1882-006

Copyright © 1992-2008, Dialogic Corporation. All Rights Reserved. You may not reproduce this document in whole or in part without permission in writing from Dialogic Corporation at the address provided below.

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

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

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

Due to differing national regulations and approval requirements, certain Dialogic products may be suitable for use only in specific countries, and thus may not function properly in other countries. You are responsible for ensuring that your use of such products occurs only in the countries where such use is suitable. For information on specific products, contact Dialogic Corporation at the address indicated below or on the web at www.dialogic.com.

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

Dialogic, Dialogic Pro, Brooktrout, Cantata, SnowShore, Eicon, Eicon Networks, Eiconcard, Diva, SIPcontrol, Diva ISDN, TruFax, Realblocs, Realcomm 100, NetAccess, Instant ISDN, TRXStream, Exnet, Exnet Connect, EXS, ExchangePlus VSE, Switchkit, N20, Powering The Service-Ready Network, Vantage, Making Innovation Thrive, Connecting People to Information, Connecting to Growth and Shiva, among others as well as related logos, are either registered trademarks or trademarks of Dialogic. Dialogic's trademarks may be used publicly only with permission from Dialogic. Such permission may only be granted by Dialogic's legal department at 9800 Cavendish Blvd., 5th Floor, Montreal, Quebec, Canada H4M 2V9. Any authorized use of Dialogic's trademarks will be subject to full respect of the trademark guidelines published by Dialogic from time to time and any use of Dialogic's trademarks requires proper acknowledgement.

Windows is a registered trademark of Microsoft Corporation in the United States and/or other countries. The other names of actual companies and products mentioned herein are the trademarks of their respective owners.

Publication Date: May 2008

Document Number: 05-1882-006

Contents

Revision History

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

Document No.	Publication Date	Description of Revisions
05-1882-006	May 2008	Function Summary by Category chapter: Added new sr_createevtdatapcopy( ) and sr_getevtdatapex( ) functions in Event Data Retrieval Functions section. Added new sr_destroy( ) function in Miscellaneous Functions section. sr_createevtdatapcopy( ) function: New function. sr_destroy( ) function: New function. sr_getevtdatapex( ) function: New function. sr_waitevt( ) function: Under Cautions, added that sr_waitevt( ) and sr_waitevtEx( ) should not be used in the same application, and added that sr_waitevt( ) is recommended for single-threaded applications. sr_waitevtEx( ) function: In the Description, added that sr_waitevtEx( ) is recommended for multithreaded applications. Under Cautions, added that sr_waitevt( ) provides better performance for single-threaded applications.
05-1882-005	November 2007	Global changes: Made global changes to reflect Dialogic brand. Corrected the code examples to show that the function prototype for user-supplied event handler functions in Linux is *long usr_hdlr (void *parm)*. The examples were changed for the following functions: sr_dishdlr( ), sr_enbhdlr( ), sr_getevtdatap( ), sr_getevtdev( ), sr_getevtlen( ), sr_getevttype( ), sr_waitevt( ). (IPY00039889) sr_dishdlr( ) and sr_enbhdlr( ) functions: Added caution about enabling/disabling handlers from within handlers. (IPY00035369, IPY00035465) sr_waitevt( ) function: Deleted note about the Device Grouping functions providing an improved mechanism for managing events. Added caution that the application should not call blocking functions in the event handling loop. sr_waitevtEx( ) function: Revised the function description to indicate that using sr_waitevtEx( ) does not result in any performance improvements over using sr_waitevt( ). Deleted note about the Device Grouping functions providing an improved mechanism for managing events. Added recommendation that the same device should not be used across multiple threads. Creating a thread per span or a thread per board is recommended. (IPY00039620) Added caution that the application should not call blocking functions in the event handling loop.
05-1882-004	June 2005	sr_putevt( ) function: Revised description for **dev** in parameter table. (PTR 34309)

Document No.	Publication Date	Description of Revisions
05-1882-003	October 2004	Function Summary by Category chapter: Removed sr_libinit() and sr_GetDllVersion() functions from section Standard Runtime Library Parameter Functions as they are no longer supported. Cross-compatibility libraries are no longer distributed. (PTR 32966) Function Information chapter: Removed sr_libinit() and sr_GetDllVersion() functions as they are no longer supported. Cross-compatibility libraries are no longer distributed. (PTR #32966) sr_AddToThreadDeviceGroup( ) function: Corrected errors in function prototype; Name should show “long” (for Linux) and “int” (for Windows®) rather than “void” and Returns should show 0 or -1 rather than “none”. sr_CreateThreadDeviceGroup( ) function: Corrected errors in function prototype; Name should show “long” (for Linux) and “int” (for Windows®) rather than “int”; Devices should show “long” rather than “int”. sr_DeleteThreadDeviceGroup( ) function: Corrected errors in function prototype; Name should show “long” (for Linux) and “int” (for Windows®) rather than “void” and Returns should show 0 or -1 rather than “none”. sr_GetThreadDeviceGroup( ) function: Corrected errors in function prototype; Name should show “long” (for Linux) and “int” (for Windows®) rather than “void” and Returns should show 0 or -1 rather than “none”. sr_NotifyEvent( ) function: Corrected error in Returns section of function prototype. There are no returns for this function. sr_RemoveFromThreadDeviceGroup( ) function: Corrected errors in function prototype; Name should show “long” (for Linux) and “int” (for Windows®) rather than “void” and Returns should show 0 or -1 rather than “none”. sr_waitevt( ) function: Corrected error in function prototype; Mode should show “synchronous” only. Revised description for -1 error return in the Description section and Errors section. sr_waitevtEx( ) function: Corrected error in function prototype; Mode should show “synchronous” only. sr_WaitThreadDeviceGroup( ) function: Corrected errors in function prototype; Name should show “long” (for Linux) and “int” (for Windows®) rather than “int”; TimeOut should show “long” rather than “int”. SRLGetAllPhysicalBoards( ) function: Corrected error returns in Example code to ESR_NOERR and ESR_INSUFBUF. SRLGetSubDevicesOnVirtualBoard( ) function: Corrected error return in Example code to ESR_NOERR. SRLGetVirtualBoardsOnPhysicalBoard( ) function: Corrected error returns in Example code to ESR_NOERR and ESR_INSUFBUF. SRLGetPhysicalBoardName( ) function: Corrected error in function prototype; Returns should show specific values rather than 0 and -1. Added new information in Errors section.
05-1882-002	November 2003	Function Summary by Category chapter: Added new sr_getUserContext( ) function in Event Data Retrieval Functions section. sr_getUserContext( ) function: New function. SRLGetAllPhysicalBoards( ) function: Revised description and example code. SRLGetSubDevicesOnVirtualBoard( ) function: Revised example code. SRLGetVirtualBoardsOnPhysicalBoard( ) function: Revised example code. SRLDEVICEINFO structure: Removed reference to devmapr4.h; device types are defined in srllib.h. Added two device types: TYPE_R4_MOH_BOARD and TYPE_R4_PHYSICAL_BOARD.

Document No.	Publication Date	Description of Revisions
05-1882-001	September 2002	Initial version of document. Much of the information contained in this document was previously published in the *Voice Software Reference—Standard Runtime Library for Linux*, document number 05-1455-003, and the *Voice Software Reference: Standard Runtime Library for Windows*, document number 05-1458-002.

About This Publication

The following topics provide information about this publication:

Purpose

The Dialogic® Standard Runtime Library (SRL) contains functions that provide event handling and other functionality common to Dialogic® devices. This publication contains details of the Standard Runtime Library functions, events, and error codes supported on Linux and Windows® operating systems.

Applicability

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

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

Intended Audience

This publication is intended for software developers who will access the Standard Runtime Library software. This may include any of the following:

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

How to Use This Publication

Refer to this publication after you have installed the hardware and the system software which includes the Dialogic® Standard Runtime Library software. This publication assumes that you are familiar with the Linux or Windows® operating system and the C programming language.

The information in this guide is organized as follows:

- Chapter 1, "Function Summary by Category", groups the Standard Runtime Library APIs into categories.
- Chapter 2, "Function Information", provides details about each Standard Runtime Library API function, including parameters, cautions, and error codes.
- Chapter 3, "Events", describes the events returned by the Standard Runtime Library software.
- Chapter 4, "Data Structures", provides details about each data structure used by the Standard Runtime Library software, including fields and descriptions.
- Chapter 5, "Error Codes", lists the error codes included in the Standard Runtime Library software.
- The Glossary provides a definition of terms used in this guide.

Related Information

This publication is a companion to the *Dialogic® Standard Runtime Library API Programming Guide*, which discusses supported programming models and programming guidelines for building applications.

Refer to the following documents and websites for more information:

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

Function Summary by Category 1

The Dialogic® Standard Runtime Library (SRL) contains functions that provide event handling and other functionality common to Dialogic® devices. This chapter contains an overview of the Standard Runtime Library functions, which are grouped into the following categories:

1.1 Event Handling Functions

Event handling functions are used to enable or disable event handlers, to hold events while other processing takes place, or to specify the amount of time to wait for the next event. You can enable and disable event handlers for specific events on specific devices. You can also enable backup event handlers to serve as contingencies for events that you have not specifically enabled. See the *Dialogic® Standard Runtime Library API Programming Guide* for detailed guidelines about using event handlers.

sr_dishdlr( )
disable an event handler

sr_enbhdlr( )
enable an event handler

sr_NotifyEvent( ) [Windows® only]
send event notification to a window

sr_putevt( )
add an event to the Standard Runtime Library event queue

sr_waitevt( )
wait for next event

sr_waitevtEx( )
wait for events on certain devices

1.2 Event Data Retrieval Functions

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

sr_createevtdatapcopy( )
copy SRL event data block

sr_getevtdatap( )
return a pointer to the variable data associated with the current event

sr_getevtdatapex( )
return the address and optionally copy an event data block

sr_getevtdev( )
get the handle for the current event

sr_getevtlen( )
get the length of variable data associated with the current event

sr_getevttype( )
get the event type for the current event

sr_getfdcnt( ) [Linux only]
get the total number of Linux file descriptors

sr_getfdinfo( ) [Linux only]
populate the **fdarray** argument with Linux file descriptors

sr_getUserContext( )
return a user-supplied pointer originally passed to the function

1.3 Standard Runtime Library Parameter Functions

Parameter functions are used to check the status of and set the value of Standard Runtime Library parameters.

sr_getparm( )
get a Standard Runtime Library parameter

sr_setparm( )
set a Standard Runtime Library parameter

sr_getboardcnt( ) [Windows® only]
get the number of boards of a specific type

1.4 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: The Standard Runtime Library contains a special device called SRL_DEVICE, which has attributes and can generate events just as any other Dialogic® device. Parameters for SRL_DEVICE can be set within the application program.

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_", and the name after the underscore describes the attribute.

The standard attribute functions and the information they return are listed below.

ATDV_ERRMSGP( )
pointer to string describing error on last library call

ATDV_IOPORT( ) [Linux only]
base address of I/O port

ATDV_IRQNUM( )
interrupt being used

ATDV_LASTERR( )
error that occurred on last library call

ATDV_NAMEP( )
pointer to device name

ATDV_SUBDEVS( )
number of subdevices

1.5 Device Mapper Functions

The device mapper API is a subset of the Standard Runtime Library. It returns information about the structure of the system, such as a list of all the virtual boards on a physical board. The device mapper API works for any component that exposes R4 devices.

The device mapper functions and the information they return are listed below:

SRLGetAllPhysicalBoards( )
retrieve a list of all physical boards in a node

SRLGetJackForR4Device( )
retrieve the jack number of an R4 device

SRLGetPhysicalBoardName( )
returns the physical board name for a specified AUID

SRLGetSubDevicesOnVirtualBoard( )
retrieve a list of all subdevices on a virtual board

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

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

1.6 Device Grouping Functions

The device grouping functions allow a direct association between threads and devices. The device grouping APIs can be used to group devices together and wait for events from one of the devices. If your application requires more sophistication, you can use other device grouping APIs to manipulate a device group after it has been established. See the *Dialogic® Standard Runtime Library API Programming Guide* for more information on implementing the device grouping API variant of the Extended Asynchronous model.

sr_AddToThreadDeviceGroup( )
add specified devices to the group

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

sr_DeleteThreadDeviceGroup( )
remove all devices from the group

sr_GetThreadDeviceGroup( )
retrieve all devices from the group

sr_RemoveFromThreadDeviceGroup( )
remove specified devices from the group

sr_WaitThreadDeviceGroup( )
wait for events on devices in the specified group

1.7 Miscellaneous Functions

These are Standard Runtime Library functions that don't fall into any other category.

sr_destroy( )
free a copy of event data from the SRL

Function Information 2

This chapter contains a detailed description of each function in the Dialogic® Standard Runtime Library (SRL). The functions are presented in alphabetical order.

2.1 Function Syntax Conventions

The Standard Runtime Library functions use the following format:

```
sr_Function (Parameter1, Parameter2, …, ParameterN)
```

where:

sr_Function
is the name of the function

Parameter1, Parameter2, …, ParameterN
are input or output fields

ATDV_ERRMSGP( )

Name: char * ATDV_ERRMSGP(dev)

Inputs: int dev • valid device handle

Returns: pointer to string

Includes: srllib.h

Category: Standard Attribute functions

Mode: Synchronous

Description

The **ATDV_ERRMSGP( )** function returns a pointer to an ASCIIZ string containing the error that occurred on the device during the last function 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".

Parameter	Description
dev	specifies the valid device handle obtained when the device was opened using **xx_open( )**, where *xx* is the technology-specific prefix identifying the device to be opened; for example, **dx_open( )** for voice technology.

Cautions

None.

Errors

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

Example

```
#include <windows.h>  /* Windows apps only */
#include <srllib.h>
#include <dxxxlib.h>

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

  /* Open dxxx channel device */
   if(( dxxxdev = dx_open( "dxxxB1C1", 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 ));
   }
}
```

See Also

- The appropriate library-specific Programming Guide

ATDV_IOPORT( )

Name: long ATDV_IOPORT(dev)

Inputs: int dev • valid device handle

Returns: AT_FAILURE if failure
otherwise Base Port Address of device

Includes: srllib.h

Category: Standard Attribute functions (Linux only)

Mode: Synchronous

Description

Supported under Linux only. The **ATDV_IOPORT( )** function returns the base port address used by the device.

Parameter	Description
dev	specifies the valid device handle obtained when the device was opened using **xx_open( )**, where *xx* is the technology-specific prefix identifying the device to be opened.

Cautions

None.

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.

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 ));
}
```

See Also

None

ATDV_IRQNUM( )

Name: long ATDV_IRQNUM(dev)

Inputs: int dev • valid device handle

Returns: AT_FAILURE if failure
otherwise, IRQ of device

Includes: srllib.h

Category: Standard Attribute functions

Mode: Synchronous

Description

The **ATDV_IRQNUM( )** function returns the interrupt number (IRQ) used by the device.

Parameter	Description
dev	specifies the valid device handle obtained when the device was opened using **xx_open( )**, where *xx* is the technology-specific prefix identifying the device to be opened.

Cautions

None.

Errors

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

Example

```
#include <windows.h>  /* Windows apps only */
#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 ));
}
```

See Also

None

ATDV_LASTERR( )

Name: long ATDV_LASTERR(dev)

Inputs: int dev • valid device handle

Returns: EDV_BADDESC if an invalid device handle (Linux)
AT_FAILURE if an invalid device handle (Windows®)
otherwise a valid error number

Includes: srllib.h

Category: Standard Attribute functions

Mode: Synchronous

Description

Linux: The **ATDV_LASTERR( )** function returns a long that indicates the last error that occurred on this device. 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.

Windows®: The **ATDV_LASTERR( )** function returns a long value that indicates the last error that occurred on this device. The errors are defined in the technology-specific header (.h) file of the specified device.

The function parameters are described as follows:

Parameter	Description
dev	specifies the valid device handle obtained when the device was opened using **xx_open( )**, where *xx* is the technology-specific prefix identifying the device to be opened.

Cautions

None

Errors

Linux: This function returns EDV_BADDESC if an invalid device handle is specified in **dev**.

Windows®: This function returns AT_FAILURE if an invalid device handle is specified in **dev**.

Example

```
#include <windows.h>  /* Windows apps only */
#include <srllib.h>
#include <dxxxlib.h>
```

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

  /* Open dxxx channel device */
   if(( dxxxdev = dx_open( "dxxxB1C1", 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 ));
   }
}
```

See Also

- The appropriate library-specific Programming Guide

ATDV_NAMEP( )

Name: char * ATDV_NAMEP(dev)

Inputs: int dev • valid device handle

Returns: pointer to string

Includes: srllib.h

Category: Standard Attribute functions

Mode: Synchronous

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:

- dxxxB*b*C*c*
- dtiB*b*T*t*

where

- *b* is the number of the board in the system
- *c* is the number of the channel on the real or emulated Dialogic® D/4x board
- *t* is the number of the time slot of a digital network interface

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

Parameter	Description
dev	specifies the valid device handle obtained when the device was opened using **xx_open**( ), where *xx* is the technology-specific prefix identifying the device to be opened.

Cautions

None

Errors

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

Example

```
#include <windows.h>  /* Windows apps only */
#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 ));
}
```

See Also

- The appropriate library-specific Programming Guide

ATDV_SUBDEVS( )

Name: long ATDV_SUBDEVS(dev)

Inputs: int dev
- valid device handle

Returns: AT_FAILURE if failure
otherwise, number of subdevices

Includes: srllib.h

Category: Standard Attribute functions

Mode: Synchronous

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 digital network interface virtual board and channels on a virtual voice board.

Parameter	Description
dev	specifies the valid device handle obtained when the device was opened using **xx_open( )**, where xx is the technology-specific prefix identifying the device to be opened.

Cautions

None.

Errors

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

Example

```
#include <windows.h>  /* Windows apps only */
#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 ));
}
```

See Also

- The appropriate library-specific Programming Guide

sr_AddToThreadDeviceGroup( )

Name: Linux: long sr_AddToThreadDeviceGroup (*Devices, NumDevices)
Windows®: int sr_AddToThreadDeviceGroup (*Devices, NumDevices)

Inputs: long *Devices — pointer to a list of device handles
int NumDevices — number of devices in the list

Returns: 0 on success
-1 on failure

Includes: srllib.h

Category: Device Grouping functions

Mode: Synchronous

Description

The **sr_AddToThreadDeviceGroup( )** function adds the listed devices to the grouping established for the thread. If any devices listed by **sr_AddToThreadDeviceGroup( )** already exist in the thread's group, no action is taken for those devices.

Parameter	Description
Devices	pointer to a list of device handles
NumDevices	number of devices in the list

Cautions

None.

Errors

If this function returns -1 to indicate failure, obtain the reason for the error by calling the Standard Runtime Library standard attribute function **ATDV_LASTERR( )** or **ATDV_ERRMSGP( )** to retrieve either the error code or a pointer to the error description, respectively.

Example

```
EventPollThread ()
{
   long Devices [24];
   int DevNum;
   long EventHandle;

   for (DevNum = 0; DevNum < 24; DevNum++)
   {
      Devices [DevNum] = dx_open(...);
   }
```

```
   sr_CreateThreadDeviceGroup (Devices);
   while (1)
   {
      sr_WaitThreadDeviceGroup (-1);
      // do something with the event
      if (done == true)
      {
         break;
      }
   }
   sr_RemoveFromThreadDeviceGroup(Devices, 24);
}
```

See Also

- **sr_CreateThreadDeviceGroup( )**

sr_createevtdatapcopy( )

Name: void * sr_createevtdatapcopy(long evt, const void * evtdatap, int datalen)

Inputs:

long evt	• event type ID
const void * evtdatap	• data block pointer
int datalen	• data block size

Returns: valid memory pointer if successful
NULL if failure

Includes: srllib.h

Category: Event Data Retrieval functions

Mode: Synchronous

Description

The **sr_createevtdatapcopy( )** function is used to copy SRL event data blocks. It returns a pointer to the copied data. An example use case is where a separate event handling thread is running and the data for events is passed to it for processing. This data must be copied, because the SRL will invalidate it once **sr_waitevt( )** or **sr_waitevtEx( )** is called.

SRL event data blocks can be copied using either **sr_createevtdatapcopy( )** or **sr_getevtdatapex( )**.

The data allocated through the **sr_createevtdatapcopy( )** function must be freed by the application using **sr_destroy( )**.

Parameter	Description
evt	event type ID
evtdatap	data block pointer. If the **evtdatap** passed is NULL, then **sr_createevtdatapcopy( )** returns NULL instead of a valid memory pointer.
datalen	data block size

The **sr_createevtdatapcopy( )** function may be called from threads other than the one that called **sr_waitevt( )** to make copies of copies. In other words, the following scenario is allowed:

- **sr_waitevt( )** is called in thread_1
- **sr_getevtdatap( )** is called in thread_1 assigning the returned value to pevtdatap_1
- **sr_createevtdatapcopy( )** is called in thread_1 to create a copy of the event data, pevtdata_2
- pevtdata_2, the event type ID, and event length are passed to thread_2
- **sr_createevtdatapcopy( )** is called in thread_2 on pevtdata_2 to create pevtdatap_3
- pevtdatap_3 is passed to thread_3 for some user defined use case
- **sr_destroy( )** is called on pevtdatap_2 in thread_2
- **sr_destroy( )** is called on pevtdatap_3 in thread_3

Cautions

Data copies made with this function **must** be freed using **sr_destroy( )**. Do not use **free( )** to free data returned from this function.

Errors

None

Linux Example

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

typedef struct MYEVENT
{
    long evttype;
    long evtdev;
    long evtlen;
    void * evtdatap;
}MyEvent;

/* assume this adds an event to a list of events to be processed by a pool of
 * worker threads
 */
void addEvent(MyEvent* newEvent);

  /* assume this returns when a new event for processing by the worker threads and
   * that it blocks until there is an event to process
   */
MyEvent* getNextEvent();

int generic_handler()
{
    MyEvent *pNewEvent = (MyEvent *)malloc(sizeof(MyEvent));

    pNewEvent->evttype = sr_getevttype();
    pNewEvent->evtdev = sr_getevtdev();
    pNewEvent->evtlen = sr_getevtlen();
    void * evtdatap = sr_getevtdatap();

    if (evtdatap)
    {
        pNewEvent->evtdatap = sr_createevtdatapcopy(newEvent.evttype,
                                                    evtdatap,
                                                    newEvent.evtlen);
        if (pNewEvent->evtdatap == NULL)
        {
            /* handle error */
            free(pNewEvent);
            return 1;
        }
    }
    else
    {
        pNewEvent->evtdatap = NULL;
    }

    addEvent(pNewEvent);

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

```
  /* assume this function is running in a separate thread */
void eventPoolThread()
{
      /* process events while global application running flag is non-zero */
    while(gRunning)
    {
        MyEvent *pNewEvent = getNextEvent();
        printf("New event to process: type %ld, device %ld, length %ld, datap %ld\n",
               pNewEvent->evttype,
               pNewEvent->evtdev,
               pNewEvent->evtlen,
               pNewEvent->evtdatap);
        sr_destroy(pNewEvent->evtdatap);
        free(pNewEvent);
    }
}

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(( dxxxdev = dx_open( "dxxxB1C1", 0 )) == -1 )
    {
        printf( "failed to open device\n" );
        exit( 1 );
    }

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

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

■ Windows® Example

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

```
typedef struct MYEVENT
{
    long evttype;
    long evtdev;
    long evtlen;
    void * evtdatap;
}MyEvent;

/* assume this adds an event to a list of events to be processed by a pool of
 * worker threads
 */
void addEvent(MyEvent* newEvent);

  /* assume this returns when a new event for processing by the worker threads and
   * that it blocks until there is an event to process
   */
MyEvent* getNextEvent();
bool gRunning = true;

long int generic_handler(unsigned long evthandle)
{
    MyEvent *pNewEvent = (MyEvent *)malloc(sizeof(MyEvent));

    pNewEvent->evttype = sr_getevttype();
    pNewEvent->evtdev = sr_getevtdev();
    pNewEvent->evtlen = sr_getevtlen();
    void * evtdatap = sr_getevtdatap();

    if (evtdatap)
    {
        pNewEvent->evtdatap = sr_createevtdatapcopy(pNewEvent->evttype,
                                                    evtdatap,
                                                    pNewEvent->evtlen);
        if (pNewEvent->evtdatap == NULL)
        {
            /* handle error */
            free(pNewEvent);
            return 1;
        }
    }
    else
    {
        pNewEvent->evtdatap = NULL;
    }

    addEvent(pNewEvent);

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

  /* assume this function is running in a separate thread */
void eventPoolThread()
{
      /* process events while global application running flag is non-zero */
    while(gRunning)
    {
        MyEvent *pNewEvent = getNextEvent();
        printf("New event to process: type %ld, device %ld, length %ld, datap %ld\n",
               pNewEvent->evttype,
               pNewEvent->evtdev,
               pNewEvent->evtlen,
               pNewEvent->evtdatap);
        sr_destroy(pNewEvent->evtdatap);
        free(pNewEvent);
    }
}
```

```
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(( dxxxdev = dx_open( "dxxxB1C1", 0 )) == -1 )
    {
        printf( "failed to open device\n" );
        exit( 1 );
    }

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

    /* Generate events via async calls */
    if( dx_sethook( dxxxdev, DL_ONHOOK, EV_ASYNC ) == -1 )
    {
        printf( "dx_sethook failed: error = %s\n", ATDV_ERRMSGP( dxxxdev ));
        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 */
        return 0;
}
```

See Also

- **sr_destroy( )**
- **sr_getevtdatapex( )**

sr_CreateThreadDeviceGroup( )

Name: Linux: long sr_CreateThreadDeviceGroup (*Devices, NumDevices)
Windows®: int sr_CreateThreadDeviceGroup (*Devices, NumDevices)

Inputs: long *Devices — pointer to a list of device handles
int NumDevices — number of devices in the list

Returns: 0 on success
-1 on failure

Includes: srllib.h

Category: Device Grouping functions

Mode: Synchronous

Description

The **sr_CreateThreadDeviceGroup( )** function is used to specify a list of devices that a thread will poll for events. This function must be called prior to calling **sr_WaitThreadDeviceGroup( )**. If this function is called repeatedly without calling **sr_DeleteThreadDeviceGroup( )**, the error ESR_THREAD_DEVICE_GROUP_EXISTS will be generated.

Parameter	Description
Devices	points to a list of device handles
NumDevices	number of devices in the device list

Cautions

This function must be called prior to calling **sr_WaitThreadDeviceGroup( )**.

Errors

If this function returns -1 to indicate failure, obtain the reason for the error by calling the Standard Runtime Library standard attribute function **ATDV_LASTERR( )** or **ATDV_ERRMSGP( )** to retrieve either the error code or a pointer to the error description, respectively. One of the following errors may be returned:

ESR_NOMEM
Standard Runtime Library could not allocate internal memory to accommodate the devices specified

ESR_THREAD_DEVICE_GROUP_EXISTS
A thread device group has already been created for the thread

Example

```
EventPollThread ()
{
   long Devices [24];
   int DevNum;
   long EventHandle;

   for (DevNum = 0; DevNum < 24; DevNum++)
   {
      Devices [DevNum] = dx_open(...);
   }

   sr_CreateThreadDeviceGroup (Devices, 24);
   while (1)
   {
      sr_WaitThreadDeviceGroup (-1);
      // do something with the event
   }
}
```

See Also

- sr_DeleteThreadDeviceGroup( )
- sr_RemoveFromThreadDeviceGroup( )
- sr_WaitThreadDeviceGroup( )

sr_DeleteThreadDeviceGroup( )

Name: Linux: long sr_DeleteThreadDeviceGroup(void)
Windows®: int sr_DeleteThreadDeviceGroup(void)

Inputs: none

Returns: 0 on success
-1 on failure

Includes: srllib.h

Category: Device Grouping functions

Mode: Synchronous

Description

The **sr_DeleteThreadDeviceGroup( )** function removes all devices from the grouping established for the thread.

Cautions

- After calling this function, **sr_CreateThreadDeviceGroup( )** must be called again in the thread before **sr_WaitThreadDeviceGroup( )** can be used.
- Devices should be idle before being removed from a group, otherwise events may accumulate unnecessarily in the Standard Runtime Library's event queue. This can result in a significant memory leak.

Errors

If this function returns -1 to indicate failure, obtain the reason for the error by calling the Standard Runtime Library standard attribute function **ATDV_LASTERR( )** or **ATDV_ERRMSGP( )** to retrieve either the error code or a pointer to the error description, respectively.

Example

```
EventPollThread ()
{
   long Devices [24];
   int DevNum;
   long EventHandle;

   for (DevNum = 0; DevNum < 24; DevNum++)
   {
      Devices [DevNum] = dx_open(...);
   }

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

```
        if (done == true)
        {
           break;
        }
     }
     sr_DeleteThreadDeviceGroup();
}
```

See Also

- **sr_CreateThreadDeviceGroup( )**

sr_destroy( )

Name: long sr_destroy(void * evtdatap)

Inputs: void * evtdatap
- data block pointer

Returns: 0 on success
-1 on failure

Includes: srllib.h

Category: Miscellaneous functions

Mode: Synchronous

Description

The **sr_destroy( )** function is used to free SRL event data blocks generated by calls to **sr_createevtdatapcopy( )** and **sr_getevtdatapex( )**.

Parameter	Description
evtdatap	pointer to data block to be freed. The data block **must** have been generated by a call to **sr_createevtdatapcopy( )** or **sr_getevtdatapex( )**.

Cautions

The **sr_destroy( )** function can only be used on memory allocated by the SRL functions **sr_createevtdatapcopy( )** and **sr_getevtdatapex( )**. The function's behavior and resultant system behavior is undetermined if **evtdatap** was not allocated through a call to either of those functions.

Errors

None

Linux Example

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

typedef struct MYEVENT
{
    long evttype;
    long evtdev;
    long evtlen;
    void * evtdatap;
}MyEvent;

// assume this adds an event to a list of events to be processed by a pool of
// worker threads
void addEvent(MyEvent* newEvent);

// assume this returns when a new event for processing by the worker threads and
// that it blocks until there is an event to process
MyEvent* getNextEvent();
```

```
int generic_handler()
{
    MyEvent *pNewEvent = (MyEvent *)malloc(sizeof(MyEvent));

    pNewEvent->evttype = sr_getevttype();
    pNewEvent->evtdev = sr_getevtdev();
    pNewEvent->evtlen = sr_getevtlen();
    void * evtdatap = sr_getevtdatap();

    if (evtdatap)
    {
        pNewEvent->evtdatap = sr_createevtdatapcopy(newEvent.evttype,
                                                    evtdatap,
                                                    newEvent.evtlen);
        if (pNewEvent->evtdatap == NULL)
        {
            // handle error;
            free(pNewEvent);
            return 1;
        }
    }
    else
    {
        pNewEvent->evtdatap = NULL;
    }

    addEvent(pNewEvent);

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

// assume this function is running in a separate thread
void eventPoolThread()
{
    // process events while global application running flag is non-zero
    while(gRunning)
    {
        MyEvent *pNewEvent = getNextEvent();
        printf("New event to process: type %ld, device %ld, length %ld, datap %ld\n",
               pNewEvent->evttype,
               pNewEvent->evtdev,
               pNewEvent->evtlen,
               pNewEvent->evtdatap);
        sr_destroy(pNewEvent->evtdatap);
        free(pNewEvent);
    }
}

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(( dxxxdev = dx_open( "dxxxB1C1", 0 )) == -1 )
    {
        printf( "failed to open device\n" );
        exit( 1 );
    }
```

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

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

Windows® Example

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

typedef struct MYEVENT
{
    long evttype;
    long evtdev;
    long evtlen;
    void * evtdatap;
}MyEvent;

// assume this adds an event to a list of events to be processed by a pool of
// worker threads
void addEvent(MyEvent* newEvent);
bool gRunning = true;
// assume this returns when a new event for processing by the worker threads and
// that it blocks until there is an event to process
MyEvent* getNextEvent();

long int generic_handler(unsigned long evthandle)
{
    MyEvent *pNewEvent = (MyEvent *)malloc(sizeof(MyEvent));

    pNewEvent->evttype = sr_getevttype();
    pNewEvent->evtdev = sr_getevtdev();
    pNewEvent->evtlen = sr_getevtlen();
    void * evtdatap = sr_getevtdatap();

    if (evtdatap)
    {
        pNewEvent->evtdatap = sr_createevtdatapcopy(pNewEvent->evttype,
                                                    evtdatap,
                                                    pNewEvent->evtlen);
        if (pNewEvent->evtdatap == NULL)
        {
            // handle error;
            free(pNewEvent);
```

```
            return 1;
        }
    }
    else
    {
        pNewEvent->evtdatap = NULL;
    }

    addEvent(pNewEvent);

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

// assume this function is running in a separate thread
void eventPoolThread()
{
    // process events while global application running flag is non-zero
    while(gRunning)
    {
        MyEvent *pNewEvent = getNextEvent();
        printf("New event to process: type %ld, device %ld, length %ld, datap %ld\n",
               pNewEvent->evttype,
               pNewEvent->evtdev,
               pNewEvent->evtlen,
               pNewEvent->evtdatap);
        sr_destroy(pNewEvent->evtdatap);
        free(pNewEvent);
    }
}

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(( dxxxdev = dx_open( "dxxxB1C1", 0 )) == -1 )
    {
        printf( "failed to open device\n" );
        exit( 1 );
    }

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

    /* Generate events via async calls */
    if( dx_sethook( dxxxdev, DL_ONHOOK, EV_ASYNC ) == -1 )
    {
        printf( "dx_sethook failed: error = %s\n", ATDV_ERRMSGP( dxxxdev ));
        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 */
        return 0;
}
```

See Also

- **sr_createevtdatapcopy( )**
- **sr_getevtdatapex( )**

sr_dishdlr( )

Name: long sr_dishdlr(dev, evt_type, handler)

Inputs:

long dev	• device handle
long evt_type	• event type
Linux: long (*handler) (void *parm) Windows®: long (*handler) (unsigned long parm)	• event handling function

Returns: 0 if success
-1 if failure

Includes: srllib.h

Category: Event Handling functions

Mode: Synchronous

Description

The **sr_dishdlr( )** function disables the handler function, **handler( )**, that was previously enabled using **sr_enbhdlr( )** on a device/event type/handler triplet.

The function parameters are described as follows:

Parameter	Description
dev	specifies the valid device handle obtained when the device was opened using an **xx_open( )** function, where *xx* is the technology-specific prefix identifying the device to be opened. Specify EV_ANYDEV to be notified of an event on any device.
evt_type	specifies the event for which the application is waiting, for example: • specify the specific event. Refer to the library-specific Programming Guide for a list of possible event types. • specify EV_ANYEVT in **evt_type** and the device in the **dev** parameter to be notified of any event on a specific device. • specify EV_ANYEVT in **evt_type** and EV_ANYDEV in the **dev** parameter to be notified of any event on any device.
handler	points to an application-defined event handler that processes the event. See **sr_enbhdlr( )** for more information on the application-defined event handlers.

Cautions

- Enabling/disabling handlers from within handlers is not recommended. The SRL should call the handlers in response to a received event.
- Only one handler is disabled by this function; the handler must be disabled under the same conditions as it 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.

- Handlers cannot be used with the Extended Asynchronous Programming model.
- Linux: When a handler is disabled while an event is being dealt with, the handler is not called if it has not been called previously.

Errors

Linux: 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 device/type/handler triple has not already been registered.

Windows®: If this function returns -1 to indicate failure, obtain the reason for the error by calling the Standard Runtime Library standard attribute function **ATDV_LASTERR(SRL_DEVICE)** or **ATDV_ERRMSGP(SRL_DEVICE)** to retrieve either the error code or a pointer to the error description, respectively. One of the following errors may be returned:

ESR_SYS
Error from operating system; use **dx_fileerrno( )** to obtain error value.

Example

```
#include <windows.h>  /* Windows apps only  */
#include <srllib.h>
#include <dxxxlib.h>

/* LINUX: set up handler  */
long dx_handler(void *parm)
{
   printf( "dx_handler() called, event is 0x%x\n", sr_getevttype());
   return( 0 );
/* tell SRL to dispose of the event */
}

/* Windows: set up handler  */
long int dx_handler(unsigned long evhandle)
{
   printf( "dx_handler() called, event is 0x%x\n", sr_getevttype(evhandle));
   return( 0 );                /* tell SRL to dispose of the event */
}

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

  /* LINUX: 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_enbhdlr( 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 );
     }
}
```

See Also

- **sr_enbhdlr( )**
- The appropriate library-specific Programming Guide

sr_enbhdlr( )

Name: long sr_enbhdlr(dev, evt_type, handler)

Inputs:

long dev	• device handle
long evt_type	• event type
Linux: long (*handler) (void *parm) Windows®: long (*handler) (unsigned long parm)	• event handling function

Returns: 0 if success
-1 if failure

Includes: srllib.h

Category: Event Handling functions

Mode: Synchronous

Description

The **sr_enbhdlr( )** function enables the handler function, **handler( )**, for the device/event pair. A handler is a user-defined function called by the Standard Runtime Library to handle a specified event that occurs on a specified device. See the *Dialogic® Standard Runtime Library API Programming Guide* for details about using handlers, including the hierarchy in which handlers are called.

Handlers must return 1 to advise the Standard Runtime Library to keep the event, or 0 to advise the Standard Runtime Library to release the event. If a handler returns 0 and the event is released, **sr_waitevt( )** does not return for that event.

The function parameters are described as follows:

Parameter	Description
dev	specifies the valid device handle obtained when the device was opened using an **xx_open( )** function, where *xx* is the technology-specific prefix identifying the device to be opened. Specify EV_ANYDEV to be notified of an event on any device.
evt_type	specifies the event for which the application is waiting; for example: • specify the specific event. Refer to the appropriate library-specific Programming Guide for a list of possible event types. • specify EV_ANYEVT in **evt_type** and the device in the **dev** parameter to be notified of any event on a specific device. • Specify EV_ANYEVT in **evt_type** and EV_ANYDEV in the **dev** parameter to be notified of any event on any device.
handler	points to an application-defined event handler function that processes the event

The following guidelines apply to event handlers:

- You can enable more than one handler for any event. The Standard Runtime Library calls all specified handlers when a thread detects the event.
- You can enable general handlers that handle all events on a specified device.
- You can enable a handler for any event on any device.
- You cannot call synchronous functions in a handler.
- You can enable or disable handlers from any thread.

You can enable a handler from within another handler because the Standard Runtime Library's event handling is fully re-entrant. For the same reason, you can open and close devices inside handlers. However, you cannot call handlers from within handlers because you cannot call **sr_waitevt( )** from within a handler. Control must return to the main thread before **sr_waitevt( )** can be called again.

Cautions

- Enabling/disabling handlers from within handlers is not recommended. The SRL should call the handlers in response to a received event.
- Handlers are not supported in multithreaded / extended asynchronous mode.
- 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 is also 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.

Errors

Linux: If the function returns -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 device/type/handler triple has not already been registered.

Windows®: If the function returns -1 to indicate failure, obtain the reason for the error by calling the Standard Runtime Library standard attribute function **ATDV_LASTERR(SRL_DEVICE)** or **ATDV_ERRMSGP(SRL_DEVICE)** to retrieve either the error code or a pointer to the error description, respectively. One of the following errors may be returned:

ESR_SYS
: Error from operating system; use **dx_fileerrno( )** to obtain error value.

Example

```
#include <windows.h>  /* Windows apps only  */
#include <srllib.h>
#include <dxxxlib.h>

/* LINUX: set up handler  */
long dx_handler(void *parm)
{
   printf( "dx_handler() called, event is 0x%x\n", sr_getevttype());
   return( 0 );
/* tell SRL to dispose of the event */
}

/* Windows: set up handler  */
long int dx_handler(unsigned long evhandle)
{
   printf( "dx_handler() called, event is 0x%x\n", sr_getevttype(evhandle));
   return( 0 );                /* tell SRL to dispose of the event */
}

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

   /* LINUX: 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_enbhdlr( dxxxdev, EV_ANYEVT, dx_handler ) == -1 )
   {
      printf( "Error: could not enable handler\n)
      exit( 1 );
   }
}
```

See Also

- **sr_dishdlr( )**

sr_getboardcnt( )

Name: long sr_getboardcnt(class_namep, boardcntp)

Inputs: char *class_namep • pointer to class name

int *boardcntp • pointer where to return count of boards in this class

Returns: 0 if success
-1 if failure

Includes: srllib.h

Category: Standard Runtime Library Parameter functions (Windows® only)

Mode: Synchronous

■ Description

Supported on Windows® only. The **sr_getboardcnt( )** function retrieves the number of boards of a particular type. Use this function prior to starting an application in order to determine the amount of available resources. Once returned, the application can use technology-specific attribute functions to determine the number of devices (or subdevices) on the board, as well as other particular attributes of the board.

Note: The device mapper Standard Runtime Library functions provide an improved mechanism for retrieving board information. For more information on these functions, refer to Section 1.5, "Device Mapper Functions", on page 13.

Parameter	Description
class_namep	pointer to class name. Valid defines for class name are: • DEV_CLASS_AUDIO_IN – For boards equipped with an audio input jack. • DEV_CLASS_DCB – For boards supporting audio conferencing • DEV_CLASS_DTI – For DTI boards • DEV_CLASS_IPT – For boards supporting IP media operations • DEV_CLASS_MSI – For MSI (modular station interface) boards • DEV_CLASS_VOICE – For voice boards
boardcntp	pointer where to return count of boards of this class

This function returns the number of boards of the specified class. In the case of voice, the number of four-channel boards is returned. For boards with Audio Input (AI) jacks, the number of AI board devices in the system is returned.

The following standard names are applied, where **?** represents board or channel numbers (incremental, starting with 1).

DEV_CLASS_VOICE
dxxxB**?**C**?**

DEV_CLASS_DTI
dtiB**?**T**?**

DEV_CLASS_MSI
msiB?C?

DEV_CLASS_AUDIO_IN
aiB?

Note: There are no channel devices on AI devices, only board devices.

Cautions

None.

Errors

None

Example

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

long  chdev[MAXDEVS];
long  evt_handle;

main( ... )
{
   char channel_name[12], board_name[12];
   int  brd_handle;
   int brd, ch, devcnt = 0;
   int numvoxbrds = 0;

   if ( sr_getboardcnt(DEV_CLASS_VOICE, &numvoxbrds) == -1)
   {
       /* error retrieving voice boards */
   }

   for (brd = 1; brd <= numvoxbrds; brd++)
   {
       /* build the board name and open the board device to get number of channels */
      sprintf(board_name, "dxxxB%d",brd);
      if ( (brd_handle = dx_open(board_name, 0)) == -1)
      {
             /* Board open error */
      }
      for (ch = 1; ch <= ATDV_SUBDEVS(brd_handle); ch++)
      {
         sprintf(channel_name, "%sC%d", board_name, ch);
         if ( (chdev[devcnt++] = dx_open(channel_name, 0)) == -1)
         {
                /* Channel open error */
         }
      }    /* End of channel for loop */
      dx_close(brd_handle);
   }   /* End of board loop */
}
```

See Also

- **SRLGetAllPhysicalBoards( )**
- **SRLGetJackForR4Device( )**

- **SRLGetPhysicalBoardName( )**
- **SRLGetSubDevicesOnVirtualBoard( )**
- **SRLGetVirtualBoardsOnPhysicalBoard( )**

sr_getevtdatap( )

Name: Linux: void *sr_getevtdatap( )
Windows®: void *sr_getevtdatap(ehandle)

Inputs: Linux: none

Windows®: unsigned long ehandle
- event handle returned by **sr_waitevtEx( )**, or zero if **sr_waitevt( )** is used

Returns: address of variable data block
NULL if no variable data

Includes: srllib.h

Category: Event Data Retrieval functions

Mode: Synchronous

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.

Cautions

Linux: 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.

Windows®: For applications using **sr_waitevt( )**, pass a 0 parameter; for example: **sr_getevtdatap**(0).

Errors

None

Linux Example

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

long dx_handler(void *parm)
{
   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(( dxxxdev = dx_open( "dxxxB1C1", 0  )) == -1 )
   {
      printf( "failed to open device\n" );
      exit( 1 );
   }

   /* Enable handlers */
   if( sr_enbhdlr( dxxxdev, 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( dxxxdev, DL_ONHOOK, EV_ASYNC ) == -1 )
   {
      printf( "dx_sethook failed: error = %s\n", ATDV_ERRMSGP( dxxxdev ));
      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 */

}
```

Windows® Example

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

long  chdev[MAXDEVS];
unsigned long  evt_handle;

main( ... )
{
   char channel_name[12];
   int ch;
```

```
 for (ch = 0; ch < MAXDEVS; ch++)
 {
    /* Build the channel name for each channel */
    if ( (chdev[ch] = dx_open(channel_name, 0) ) == -1 )
    {
       printf("dx_open failed\n");
       exit(1);
     }
 }

/*
  * Now initialize each device setting up the event masks and then issue
  * the command asynchronously to start off the state machine.
  */
 for (ch = 0; ch < MAXDEVS; ch++)
 {
    /* Set up the event masks and other initialization */

    /* set the channel onhook asynchronously */
    if (dx_sethook( chdev[ch]. DX_ONHOOK, EV_ASYNC) == -1)
    {
          /* sethook failed, handle the error */
    }
 }

/* This is the main loop to control the Voice hardware */
 while (FOREVER)
 {
    /* wait for the event */
    sr_waitevtEx( chdev, MAXDEVS, -1, &evt_handle);
    process_event( evt_handle);
 }

 int process_event( ehandle)
 unsigned long ehandle;
 {
    int  voxhandle = sr_getevtdev(ehandle);
    DX_CST *datap;
    switch(sr_getevttype(ehandle))
    {
       case TDX_CST:
          *datap  = (DX_CST *) sr_getevtdatap(ehandle);
          if (datap->cst_event == DE_RINGS)
          {
             .
             .
          }
          break;

       case TDX_PLAY:
          .
          break;

    }
}
```

See Also

- **sr_getevtdatapex( )**
- **sr_getevtlen( )**
- The appropriate library-specific Programming Guide

sr_getevtdatapex( )

Name:	Linux: void * sr_getevtdatapex(SR_EVENTDATASCOPE evtdatascopetag) Windows®: void * sr_getevtdatapex(ehandle, SR_EVENTDATASCOPE evtdatascopetag)	
Inputs:	SR_EVENTDATASCOPE evtdatascopetag	• event data scope, denotes scope of the data block returned
	Windows® only: unsigned long ehandle	• event handle returned by **sr_waitevtEx( )**, or zero if **sr_waitevt( )** is used
Returns:	valid memory pointer if successful NULL if failure	
Includes:	srllib.h	
Category:	Event Data Retrieval functions	
Mode:	Synchronous	

Description

The **sr_getevtdatapex( )** function returns the address of the event data block associated with the current event. Use this data pointer to extract the event data. The function returns a value of NULL if no additional data is associated with the current event.

The **sr_getevtdatapex( )** function provides the flexibility of making a copy of the event data block, which can be kept in memory until the application frees it by calling **sr_destroy( )**.

SRL event data blocks can be copied using either **sr_getevtdatapex( )** or **sr_createevtdatapcopy( )**.

Parameter	Description
evtdatascopetag	denotes the scope of the data block returned The SR_EVENTDATASCOPE data type is an enumeration that defines the following values: • SR_EVENTDATASCOPE_TAG_SRL – For Windows®, signifies that the event data memory is to be freed upon the next call to **sr_waitevt( )**. For Linux, signifies that the event data memory is to be freed upon the next call to **sr_waitevt( )** or at the end of the event handle as documented in **sr_getevtdatap( )**. • SR_EVENTDATASCOPE_TAG_USER – Signifies that the event data must be freed by the user through a call to **sr_destroy( )**.

SR_EVENTDATASCOPE is defined as follows:

```
typedef enum tagSR_EVENTDATASCOPE_TAG
{
   SR_EVENTDATASCOPE_TAG_SRL,/* SRL will free up the event data memory upon
                              the next call to sr_waitevt() or sr_getevtdatap().*/
   SR_EVENTDATASCOPE_TAG_USER /* Application must free up the event data memory
                               through a call to sr_destroy().*/
}SR_EVENTDATASCOPE;
```

Calling **sr_getevtdatapex( )** with **evtdatascope** = SR_EVENTDATASCOPE_TAG_SRL is equivalent to **sr_getevtdatap( )**. No copy of the event data block is made.

Calling **sr_getevtdatapex( )** with **evtdatascope** = SR_EVENTDATASCOPE_TAG_USER is equivalent to **sr_createevtdatapcopy( )**. Both functions create a copy of the event data block that must be freed by calling **sr_destroy( )**.

Cautions

Data copies made by passing **evtdatascope** = SR_EVENTDATASCOPE_TAG_USER to this function **must** be freed using **sr_destroy( )**. Do not use **free( )** to free data returned from this function.

Errors

None

Linux Example

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

typedef struct MYEVENT
{
    long evttype;
    long evtdev;
    long evtlen;
    void * evtdatap;
}MyEvent;

  /* assume this adds an event to a list of events to be processed by a pool of
   * worker threads
   */
void addEvent(MyEvent* newEvent);

  /* assume this returns when a new event for processing by the worker threads and
   * that it blocks until there is an event to process
   */
MyEvent* getNextEvent();

int generic_handler()
{
    MyEvent *pNewEvent = (MyEvent *)malloc(sizeof(MyEvent));

    pNewEvent->evttype = sr_getevttype();
    pNewEvent->evtdev = sr_getevtdev();
    pNewEvent->evtdatap = sr_getevtdatapex(SR_EVENTDATASCOPE_TAG_USER);

    addEvent(pNewEvent);

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

```
  /* assume this function is running in a separate thread */
void eventPoolThread()
{
      // process events while global application running flag is non-zero
    while(gRunning)
    {
        MyEvent *pNewEvent = getNextEvent();
        printf("New event to process: type %ld, device %ld, length %ld, datap %ld\n",
               pNewEvent->evttype,
               pNewEvent->evtdev,
               pNewEvent->evtlen,
               pNewEvent->evtdatap);
        sr_destroy(pNewEvent->evtdatap);
        free(pNewEvent);
    }
}

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(( dxxxdev = dx_open( "dxxxB1C1", 0 )) == -1 )
    {
        printf( "failed to open device\n" );
        exit( 1 );
    }

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

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

Windows® Examples

Example with SR_EVENTDATASCOPE_TAG_USER (Creates a Copy of the Event Data Block)

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

typedef struct MYEVENT
{
    long evttype;
    long evtdev;
    long evtlen;
    void * evtdatap;
}MyEvent;

  /* assume this adds an event to a list of events to be processed by a pool of
   * worker threads
   */
void addEvent(MyEvent* newEvent);

  /* assume this returns when a new event for processing by the worker threads and
   * that it blocks until there is an event to process
   */
MyEvent* getNextEvent();
bool gRunning = true;

long int generic_handler(unsigned long evthandle)
{
    MyEvent *pNewEvent = (MyEvent *)malloc(sizeof(MyEvent));

    pNewEvent->evttype = sr_getevttype();
    pNewEvent->evtdev = sr_getevtdev();
    pNewEvent->evtdatap = sr_getevtdatapex(evthandle, SR_EVENTDATASCOPE_TAG_USER);

    addEvent(pNewEvent);

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

  /* assume this function is running in a separate thread */
void eventPoolThread()
{
      // process events while global application running flag is non-zero
    while(gRunning)
    {
        MyEvent *pNewEvent = getNextEvent();
        printf("New event to process: type %ld, device %ld, length %ld, datap %ld\n",
               pNewEvent->evttype,
               pNewEvent->evtdev,
               pNewEvent->evtlen,
               pNewEvent->evtdatap);
        sr_destroy(pNewEvent->evtdatap);
        free(pNewEvent);
    }
}

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(( dxxxdev = dx_open( "dxxxB1C1", 0 )) == -1 )
    {
        printf( "failed to open device\n" );
        exit( 1 );
    }

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

    /* Generate events via async calls */
    if( dx_sethook( dxxxdev, DL_ONHOOK, EV_ASYNC ) == -1 )
    {
        printf( "dx_sethook failed: error = %s\n", ATDV_ERRMSGP( dxxxdev ));
        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 */
        return 0;
}
```

Example with SR_EVENTDATASCOPE_TAG_SRL (Does Not Create a Copy of the Event Data Block)

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

typedef struct MYEVENT
{
    long evttype;
    long evtdev;
    long evtlen;
    void * evtdatap;
}MyEvent;

  /* assume this adds an event to a list of events to be processed by a pool of
   * worker threads
   */
void addEvent(MyEvent* newEvent);

  /* assume this returns when a new event for processing by the worker threads and
   * that it blocks until there is an event to process
   */
MyEvent* getNextEvent();
bool gRunning = true;
```

```
long int generic_handler(unsigned long evthandle)
{
    MyEvent *pNewEvent = (MyEvent *)malloc(sizeof(MyEvent));

    pNewEvent->evttype = sr_getevttype();
    pNewEvent->evtdev = sr_getevtdev();
    pNewEvent->evtdatap = sr_getevtdatapex(evthandle, SR_EVENTDATASCOPE_TAG_SRL);

    addEvent(pNewEvent);

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

  /* assume this function is running in a separate thread */
void eventPoolThread()
{
      // process events while global application running flag is non-zero
    while(gRunning)
    {
        MyEvent *pNewEvent = getNextEvent();
        printf("New event to process: type %ld, device %ld, length %ld, datap %ld\n",
               pNewEvent->evttype,
               pNewEvent->evtdev,
               pNewEvent->evtlen,
               pNewEvent->evtdatap);
        free(pNewEvent);
    }
}

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(( dxxxdev = dx_open( "dxxxB1C1", 0 )) == -1 )
    {
        printf( "failed to open device\n" );
        exit( 1 );
    }

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

    /* Generate events via async calls */
    if( dx_sethook( dxxxdev, DL_ONHOOK, EV_ASYNC ) == -1 )
    {
        printf( "dx_sethook failed: error = %s\n", ATDV_ERRMSGP( dxxxdev ));
        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 */
        return 0;
}
```

See Also

- **sr_createevtdatapcopy( )**
- **sr_destroy( )**
- **sr_getevtdatap( )**

sr_getevtdev( )

Name: Linux: long sr_getevtdev( )
Windows®: long sr_getevtdev(ehandle)

Inputs: Linux: none

Windows®: unsigned long ehandle
- event handle returned by **sr_waitevtEx( )**, or zero if **sr_waitevt( )** is used

Returns: device handle if successful
-1 if no current event

Includes: srllib.h

Category: Event Data Retrieval functions

Mode: Synchronous

Description

The **sr_getevtdev( )** function returns the device handle associated with the current event. If no current event exists, a value of -1 is returned. If a timeout occurs while waiting for an event, this function returns SRL_DEVICE.

Note: Information about the device handles that can be returned by this function can be found in the appropriate library-specific Programming Guide.

Cautions

Linux: 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.

Windows®: For applications using **sr_waitevt( )**, pass a 0 (zero) parameter: **sr_getevtdev**(0).

Errors

None.

Linux Example

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

long dx_handler(void *parm)
{
   printf( "Got event on device %s\n", ATDV_NAMEP( sr_getevtdev()));

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

   /* Enable handlers */
   if( sr_enbhdlr( dxxxdev, 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( dxxxdev, DL_ONHOOK, EV_ASYNC ) == -1 )
   {
      printf( "dx_sethook failed: error = %s\n", ATDV_ERRMSGP( dxxxdev ));
      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 */

}
```

■ Windows® Example

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

long  chdev[MAXDEVS];
long  evt_handle;

main( ... )
{
   char channel_name[12];
   int ch;
```

```
for (ch = 0; ch < MAXDEVS; ch++)
{
   /* Build the channel name for each channel */
   if ( (chdev[ch] = dx_open(channel_name, 0) ) == -1 )
   {
      printf("dx_open failed\n");
      exit(1);
   }
}

/*
 * Now initialize each device setting up the event masks and then issue
 * the command asynchronously to start off the state machine.
 */
for (ch = 0; ch < MAXDEVS; ch++)
{
   /* Set up the event masks and other initialization */

   /* set the channel onhook asynchronously */
   if (dx_sethook( chdev[ch]. DX_ONHOOK, EV_ASYNC) == -1)
   {
      /* sethook failed, handle the error */
   }
}

/* This is the main loop to control the Voice hardware */
while (FOREVER)
{
   /* wait for the event */
   sr_waitevtEx( chdev, MAXDEVS, -1, &evt_handle);
   process_event( evt_handle);
}

int process_event( ehandle)
unsigned long ehandle;
{
   int  voxhandle = sr_getevtdev(ehandle);

   switch(sr_getevttype(ehandle))
   {
   case TDX_CST:
        .
      break;

   case TDX_PLAY:
        .
      break;

   }
}
```

See Also

- sr_waitevt( )
- sr_waitevtEx( )
- The appropriate library-specific Programming Guide

sr_getevtlen( )

Name: Linux: long sr_getevtlen( )
Windows®: long sr_getevtlen(ehandle)

Inputs: Linux: none

Windows®: unsigned long ehandle
- event handle returned by **sr_waitevtEx( )**, or zero if **sr_waitevt( )** is used

Returns: length of data if successful
0 if no current event
-1 if error

Includes: srllib.h

Category: Event Data Retrieval functions

Mode: Synchronous

Description

The **sr_getevtlen( )** function returns the length of the variable data associated with the current event. If there is no data associated with the current event, a value of zero is returned.

Note: Information about the length of data returned by this function can be found in the appropriate library-specific Programming Guide.

Cautions

Linux: 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.

Windows®: This function is called differently when used with **sr_waitevt( )** or **sr_waitevtEx( )**. For applications using **sr_waitevt( )**, pass a 0 parameter: **sr_getevtlen(0)**. Do not use the event descriptor handle parameter, **ehandle**.

Errors

None

Linux Example

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

long dx_handler(void *parm)
{
   printf( "Got event with data length %d\n", sr_getevtlen());
   /* 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(( dxxxdev = dx_open( "dxxxB1C1", 0  )) == -1 )
   {
      printf( "failed to open device\n" );
      exit( 1 );
   }

   /* Enable handlers */
   if( sr_enbhdlr( dxxxdev, 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( dxxxdev, DL_ONHOOK, EV_ASYNC ) == -1 )
   {
      printf( "dx_sethook failed: error = %s\n", ATDV_ERRMSGP( dxxxdev ));
      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 */

}
```

Windows® Example

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

long  chdev[MAXDEVS];
unsigned long  evt_handle;

main( ... )
{
   char channel_name[12];
   int ch;
```

```
 for (ch = 0; ch < MAXDEVS; ch++)
 {
    /* Build the channel name for each channel */
    if ( (chdev[ch] = dx_open(channel_name, 0) ) == -1 )
    {
       printf("dx_open failed\n");
       exit(1);
    }
 }

/*
  * Now initialize each device setting up the event masks and then issue
  * the command asynchronously to start off the state machine.
  */
 for (ch = 0; ch < MAXDEVS; ch++)
 {
    /* Set up the event masks and other initialization */

    /* set the channel onhook asynchronously */
    if (dx_sethook( chdev[ch]. DX_ONHOOK, EV_ASYNC) == -1)
    {
       /* sethook failed, handle the error */
    }
 }

/* This is the main loop to control the Voice hardware */
 while (FOREVER)
 {
    /* wait for the event */
    sr_waitevtEx( chdev, MAXDEVS, -1, &evt_handle);
    process_event( evt_handle);
 }

 int process_event( ehandle)
 unsigned long ehandle;
 {
    long varlen
    int  voxhandle = sr_getevtdev(ehandle);

    switch(sr_getevttype(ehandle))
    {
       case TDX_CST:
          varlen = sr_getevtlen(ehandle)
          break;

       case TDX_PLAY:
             .
          break;
    }
 }
```

See Also

- The appropriate library-specific Programming Guide

sr_getevttype( )

Name: Linux: long sr_getevttype( )
Windows®: long sr_getevttype(ehandle)

Inputs: Linux: none

Windows®: unsigned long ehandle
- event handle returned by **sr_waitevtEx( )**, or zero if **sr_waitevt( )** is used

Returns: event type if successful
-1 if no current event

Includes: srllib.h

Category: Event Data Retrieval functions

Mode: Synchronous

Description

The **sr_getevttype( )** function returns the event type for the current event. If no current event exists, a value of -1 is returned.

Linux: This function returns SR_TMOUTEVT if a timeout occurs.

Windows:® If a timeout occurs while waiting for an event, this function returns SR_TIMEOUTEVT.

Note: Information about the device-specific event types that are returned by this function can be found in the appropriate library-specific Programming Guide.

Cautions

Linux: 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.

Windows®: This function is called differently when used with **sr_waitevt( )** or **sr_waitevtEx( )**. For applications using **sr_waitevt( )**, pass a 0 parameter: **sr_getevttype(0)**. Do not use the event descriptor handle parameter, **ehandle**.

Errors

None

Linux Example

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

```
long dx_handler(void *parm)
{
   printf( "Got event of type 0x%x\n", sr_getevttype());

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

   /* Enable handlers */
   if( sr_enbhdlr( dxxxdev, 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( dxxxdev, DL_ONHOOK, EV_ASYNC ) == -1 )
   {
      printf( "dx_sethook failed: error = %s\n", ATDV_ERRMSGP( dxxxdev ));
      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 */

}
```

Windows® Example

```
#include <windows.h>
#include <dsrllib.h>
#include <dxxxlib.h>

long  chdev[MAXDEVS];
unsigned long  evt_handle;

main( ... )
{
   char channel_name[12];
   int ch;
```

```
 for (ch = 0; ch < MAXDEVS; ch++) {
    /* Build the channel name for each channel */
    if ( (chdev[ch] = dx_open(channel_name, 0) ) == -1 ) {
       printf("dx_open failed\n");
       exit(1);
    }
 }

/*
  * Now initialize each device setting up the event masks and then issue
  * the command asynchronously to start off the state machine.
  */
 for (ch = 0; ch < MAXDEVS; ch++) {
    /* Set up the event masks and other initialization */

    /* set the channel onhook asynchronously */
    if (dx_sethook( chdev[ch]. DX_ONHOOK, EV_ASYNC) == -1) {
       /* sethook failed, handle the error */
    }
 }

/* This is the main loop to control the Voice hardware */
 while (FOREVER) {
    /* wait for the event */
    sr_waitevtEx( chdev, MAXDEVS, -1, &evt_handle);
    process_event( evt_handle);
 }

 int process_event( ehandle)
 unsigned long ehandle;
 {
    int  voxhandle = sr_getevtdev(ehandle);

    switch(sr_getevttype(ehandle)) {
       case TDX_CST:
           .
          break;

       case TDX_PLAY:
           .
          break;

    }
 }
```

See Also

- **sr_waitevt( )**
- **sr_waitevtEx( )**
- The appropriate library-specific Programming Guide

sr_getfdcnt( )

Name: int sr_getfdcnt( )
Inputs: none
Returns: the number of Linux file descriptors
Includes: srllib.h
Category: Event Data Retrieval functions (Linux only)
Mode: Synchronous

Description

Supported on Linux only. The **sr_getfdcnt( )** function returns the total number of Linux file descriptors that are used internally by the SRL eventing mechanism. This API can be used with **sr_getfdinfo( )** to allow combining multiple sources of I/Os in a single **select( )** or **poll( )** statement in the application.

Cautions

None.

Errors

None

Example

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

main()
{
   int numDesc;
   int *fdarray;

   numDesc = sr_getfdcnt();
   /* allocate storage for array of descriptors */
   fdarray = (int *)malloc(sizeof(int) * numDesc);
   sr_getfdinfo(fdarray);

   /* combine I/Os already used by application with
      the SRL event queue to handle everything in a
      single select() statement. */

   /* currently only one descriptor is returned by the
      SRL so add it to the existing descriptors used by
      this application. */
   FD_SET(fdarray[0], &app_fdset);
```

```
   /* wait for an I/O */
   select(number_of_desc, &app_fdset, NULL, NULL, NULL);

   /* is it an SRL event? */
   if (FD_ISSET(fdarray[0], &app_fdset) ==  true)
   {
      if (sr_waitevt(0) != -1)
      {
         /* we have an event, process it here */
      }
   }
   else
   {
      /* not an SRL event, must be some other I/O expected
         by this application */
   }
}
```

See Also

- **sr_getfdinfo( )**

sr_getfdinfo( )

Name: void sr_getfdinfo(fdarray[ ])

Inputs: int * fdarray[ ] • file descriptor array

Returns: none

Includes: srllib.h

Category: Event Data Retrieval functions (Linux only)

Mode: Synchronous

Description

Supported on Linux only. The **sr_getfdinfo( )** function populates the **fdarray** argument with Linux file descriptors that are used internally by the SRL eventing mechanism. Upon return, only the first *n* entries in **fdarray** are valid, where *n* is the value returned from **sr_getfdcnt( )**.

This function can be used with **sr_getfdcnt( )** to allow combining multiple sources of I/Os in a single **select( )** or **poll( )** statement in the application.

Parameter	Description
fdarray	specifies the user-allocated file descriptor array to be filled in by this function. Array should be initialized with -1.

Cautions

None.

Errors

None

Example

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

main()
{
   int numDesc;
   int *fdarray;

   numDesc = sr_getfdcnt();
   /* allocate storage for array of descriptors */
   fdarray = (int *)malloc(sizeof(int) * numDesc);
   sr_getfdinfo(fdarray);
```

```
   /* combine I/Os already used by application with
      the SRL event queue to handle everything in a
      single select() statement. */

   /* currently only one descriptor is returned by the
      SRL so add it to the existing descriptors used by
      this application. */
   FD_SET(fdarray[0], &app_fdset);

   /* wait for an I/O */
   select(number_of_desc, &app_fdset, NULL, NULL, NULL);

   /* is it an SRL event? */
   if (FD_ISSET(fdarray[0], &app_fdset) ==  true)
   {
      if (sr_waitevt(0) != -1)
      {
         /* we have an event, process it here */
      }
   }
   else
   {
      /* not an SRL event, must be some other I/O expected
         by this application */
   }
}
```

See Also

- **sr_getfdcnt( )**

sr_getparm( )

Name: long sr_getparm(dev, parmno, valuep)

Inputs:

long dev	• device
long parmno	• parameter number
void *valuep	• parameter value

Returns: 0 if successful
-1 if failure

Includes: srllib.h

Category: Standard Runtime Library Parameter functions

Mode: Synchronous

Description

The **sr_getparm**( ) function returns the value of a Standard Runtime Library parameter. The function parameters are described as follows:

Parameter	Description
dev	device handle. Generally, you should set this parameter to SRL_DEVICE, the predefined Standard Runtime Library device handle. However, if the parameter being set is SR_USERCONTEXT, then **dev** should be set to the handle returned by the technology-specific **xx_open**( ).
parmno	specifies the Standard Runtime Library parameter for which values are returned. Possible values are as follows: • SR_MODEID – **Linux:** get the value for the event notification mode. Returns an integer. • SR_MODELTYPE – **Windows® only:** get the value for the model type. Returns an integer. • SR_USERCONTEXT – get user-specific context. This lets you quickly get application-specific context given on a device handle.
valuep	pointer to an area of memory to receive the value for the specified parameter. This memory should be large enough to hold the parameter. Possible values are as follows: • SR_MODEID – **Linux:** SR_POLLMODE • SR_MODELTYPE – **Windows® only:** SR_STASYNC or SR_MTASYNC • SR_USERCONTEXT – user-specific context set through the **sr_setparm( )** function

Cautions

Normally, when getting Standard Runtime Library parameters, you must set the **dev** parameter to SRL_DEVICE. However, if you set the **parmno** parameter to SR_USERCONTEXT, you must set the **dev** parameter to the device on which context is being retrieved.

Errors

Linux: If the function returns -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 device/type/handler triple has not already been registered.

Windows®: If the function returns -1 to indicate failure, obtain the reason for the error by calling the standard attribute function **ATDV_LASTERR(SRL_DEVICE)** or **ATDV_ERRMSGP(SRL_DEVICE)** to retrieve either the error code or a pointer to the error description, respectively. One of the following errors may be returned:

ESR_SYS
Error from operating system; use **dx_fileerrno( )** to obtain error value.

Linux 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" );
}
```

Windows® Example

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

main()
{
   int mode;
   if( sr_getparm( SRL_DEVICE, SR_MODELTYPE, &mode ) == -1 )
   {
      printf( "Error: cannot get srl modeltype\n" );
      exit( -1 );
   }
   printf( "SRL is running in %s mode type\n", (mode == SR_MTASYNC) ? "MTASYNC" : "STASYNC" );
}
```

See Also

- **sr_setparm( )**

sr_GetThreadDeviceGroup( )

Name: Linux: long sr_GetThreadDeviceGroup (*Devices, *NumDevices)
Windows®: int sr_GetThreadDeviceGroup (*Devices, *NumDevices)

Inputs: long *Devices — pointer to a list of device handles
int *NumDevices — pointer to the number of devices in the list

Returns: 0 on success
-1 on failure

Includes: srllib.h

Category: Device Grouping functions

Mode: Synchronous

Description

The **sr_GetThreadDeviceGroup( )** function returns all devices from the grouping established for the thread. **NumDevices** must indicate the maximum number of devices that **Devices** can accommodate. Upon return, the value of **NumDevices** is changed to indicate the actual number of devices stored in **Devices**.

Parameter	Description
Devices	pointer to a list of device handles
NumDevices	pointer to the number of devices in the list

Cautions

- The value of **NumDevices** is changed upon return of this function.
- If **NumDevices** is specified to be a number larger than the size of the **Devices** array, a memory access violation may occur when the Standard Runtime Library builds the **Devices** list.

Errors

If this function returns -1 to indicate failure, obtain the reason for the error by calling the standard attribute function **ATDV_LASTERR( )** or **ATDV_ERRMSGP( )** to retrieve either the error code or a pointer to the error description, respectively.

Example

```
#define MAX_DEVICES 96
#define DEVICES_PER_TRUNK 24
.
.
.
```

```
EventPollThread ()
{
   long Devices [MAX_DEVICES];
   int DevNum, index;
   long EventHandle;

   for (DevNum = 0; DevNum < DEVICES_PER_TRUNK; DevNum++)
   {
      Devices [DevNum] = dx_open(...);
   }

   sr_CreateThreadDeviceGroup (Devices);
   while (1)
   {
      sr_WaitThreadDeviceGroup (-1);
      // do something with the event
      if (done == true)
      {
         break;
      }
   }

   DevNum = MAX_DEVICES;
   sr_GetThreadDeviceGroup (&Devices, &DevNum);
   printf ("%d devices in the device group:\n", DevNum);
   for (index = 0; index < DevNum; index++)
   {
      // NOTE: This loop will execute 24 times
      printf ("Device handle #%2d: %d\n", index, Devices[index]);
   }
   sr_DeleteThreadDeviceGroup();
}
```

See Also

- sr_CreateThreadDeviceGroup( )

sr_getUserContext( )

Name: Linux: void *sr_getUserContext( )
Windows®: void *sr_getUserContext(ehandle)

Inputs: Linux: none

Windows®: unsigned long ehandle
- event handle returned by **sr_waitevtEx( )**, or zero if **sr_waitevt( )** is used

Returns: address of user-supplied pointer
NULL if no pointer specified

Includes: srllib.h

Category: Event Data Retrieval functions

Mode: Synchronous

Description

The **sr_getUserContext( )** function returns a user-supplied pointer or NULL if no pointer was provided. The pointer returned is the same pointer provided to a function that supports the user context field in asynchronous mode. This pointer allows you to match an event that was received with the function call that originally produced the event.

The user context field is only available in the Dialogic® Conferencing (CNF) API library in Dialogic® System Release 6.0. For more information on the functions that support user context, see the *Dialogic® Conferencing API Library Reference*. For more information on user context, see the *Dialogic® Conferencing API Programming Guide*.

Cautions

Windows®: 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.

Windows®: For applications using **sr_waitevt( )**, pass a 0 parameter; for example: **sr_getUserContext**(0).

Errors

None

Example

For an example of this function, see the example code of a conferencing function that supports user context in the *Dialogic® Conferencing API Library Reference*.

See Also

None.

sr_NotifyEvent( )

Name: void sr_NotifyEvent(handle, message, flags)

Inputs:

HWND handle	• window handle
unsigned int message	• message number to send to window
unsigned int flags	• notification on/off flag

Returns: None

Includes: srllib.h

Category: Event Handling functions (Windows® only)

Mode: Synchronous

Description

Supported on Windows® only. The **sr_NotifyEvent( )** function informs the Standard Runtime Library to send event notification to a window. This provides a method of getting notification of events through the Windows event queue. Although the actual event does not come in through the Windows event queue, a notification message is sent which, when received, causes the application to call the **sr_waitevt( )** function with a 0 timeout to pull an event from the event queue.

Parameter	Description
handle	window handle to which the message is to be sent
message	number of the message (message ID)
flag	flags to turn event notification on or off: • SR_NOTIFY_ON – event notification on • SR_NOTIFY_OFF – event notification off

Cautions

- This function should be only used when doing an asynchronous application controlling all devices within one thread. The application must call **sr_waitevt(0)** to get the event off the event queue.
- The **sr_NotifyEvent( )** function should be called before any devices are opened.
- If **sr_NotifyEvent( )** is called a second time to send messages to a new window, the previous window is no longer available to receive messages.

Errors

None.

Example

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

```
#define  WM_SRNOTIFYEVENT    WM_USER + 100    /* user defined message for event notification */

long PASCAL  FrameWndProc(HWND, UINT, UINT, LONG);

WNDCLASS wndclass;
har szFrameClass[] = "MdiFrame";
HWND  hwndFrame;

main( ... )
{
   int chdev;

         .
         .
         .
  /* Register window class */
   wndclass.style       = CS_HREDRAW | CS_VREDRAW;
   wndclass.lpfnWndProc = FrameWndProc;
         .
         .
         .
   wndclass.lpszClassName  = szFrameClass;

   RegisterClass(&wndclass);

  /* Create Frame window */
   hwndFrame = CreateWindow(szFrameClass,
                    "Sample Dialogic Voice Application",
                    WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN,
                    CW_USEDEFAULT,
                    CW_USEDEFAULT,
                    CW_USEDEFAULT,
                    CW_USEDEFAULT,
                    NULL,
                    NULL,
                    hInstance,
                    NULL);

  /* Turn on event notification as Window message */
   sr_NotifyEvent(hwndFrame,WM_SRNOTIFYEVENT, SR_NOTIFY_ON);
         .
         .
         .
   if ((chdev = dx_open("dxxB1C1",0)) == -1)
   {
      MyPrintf("Failed to open dxxB1C1\n");
      exit(0);
   }

   if (dx_sethook(chdev,DX_ONHOOK,EV_ASYNC) == -1)
   {
      MyPrintf("Failed to go offhook: %s\n",ATDV_ERRMSGP(chdev));
   }
         .
         .
         .
}

/*********************************************************
 *  NAME: FrameWndProc()
 *  DESCRIPTION: Frame window procedure in an MDI application.
 *********************************************************/
long PASCAL FrameWndProc(HWND hwnd, UINT message, UINT wParam, LONG lParam)
```

```
{
   switch(message)
   {
      case WM_CREATE:
               .
               .
               .
      case WM_COMMAND:
               .
               .
               .
      case WM_SRNOTIFYEVENT:
         if (sr_waitevt(0) == -1)
         {
            MyPrintf("sr_waitevt: %s",ATDV_ERRMSGP(SRL_DEVICE));
            break;
         }

         switch( sr_getevttype())
         {
            case TDX_SETHOOK:
               mPrintf("Sethook complete\n");
               break;
            case TDX_PLAY:
            case TDX_RECORD:
                     .
                     .
                     .
         }
         break;

      case WM_DESTROY:
         PostQuitMessage(0);
         return 0;
   }
     .
     .
     .
}
```

See Also

- **sr_waitevt( )**
- **sr_getevtdev( )**
- **sr_getevttype( )**
- **sr_getevtlen( )**
- **sr_getevtdatap( )**
- The appropriate library-specific Programming Guide

sr_putevt( )

Name: long sr_putevt(dev, evttype, evtlen, evtdatap, errcode)

Inputs:

long dev	• device
unsigned long evttype	• event type ID
long evtlen	• data block size
void *evtdatap	• data block pointer
long errcode	• error code

Returns: 0 if successful
-1 if error

Includes: srllib.h
dxxxlib.h

Category: Event Handling functions

Mode: Synchronous

■ Description

The **sr_putevt( )** function allows the application to add an event to the Standard Runtime Library event queue within the same process. If event data is to be passed through the **evtdatap** parameter, the **evtlen** parameter should be set to its corresponding length. The Standard Runtime Library makes a copy of the user's data that is pointed to by the **evtdatap** parameter. Therefore, the caller can dispose of the event data immediately after calling the **sr_putevt( )** function.

Note: In the Linux version of *srllib.h*, this function has the signature **_sr_putevt( );** however, the functionality is identical under both operating systems.

Parameter	Description
dev	valid device handle of the device obtained by calling one of the **xx_open( )** functions. Use the **sr_getevtdev( )** function to retrieve this value when an event occurs.
evttype	user-supplied event type. This can be any unique event number. Use the **sr_getevttype( )** function to retrieve this value when an event occurs.
evtlen	length of variable-length data associated with the event. Use the **sr_getevtlen( )** function to retrieve this value when the event occurs. If no data is associated with an event, set **evtlen** to zero, and set **evtdatap** to NULL.

Parameter	Description
evtdatap	pointer to the variable-length data of the size in the **evtlen** parameter. The Standard Runtime Library makes a copy of this data and attaches it to the event. The **sr_getevtdatap( )** function returns a pointer to the memory containing the variable-length information. If no data is associated with an event, set this parameter to NULL, and set **evtlen** to zero.
errcode	specifies an error code that can be set when the event is posted. Calling the **ATDV_LASTERR( )** function on the device handle on which the event is posted returns this error. Set to zero if no error is to be reported on this device. If non-zero, you can use the **ATDV_LASTERR( )** function on this device to retrieve the error.

Use either the **sr_waitevt( )** function or the **sr_waitevtEx( )** function to retrieve the events generated by the **sr_putevt( )** function. The **sr_putevt( )** function can also unblock synchronous functions that are blocked waiting for certain events. For example, if a Dialogic® Voice library **dx_getevt( )** or **dx_wtring( )** function is waiting for CST events to be posted to the queue, you can simulate incoming calls by posting TDX_CST events to the queue.

Cautions

- This function fails if you specify an invalid device handle.
- This function returns SR_NOMEM if memory cannot be allocated to store the event on the Standard Runtime Library event queue.

Errors

None

Example

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

int        dev;     /* device handle */
DX_CST     cst;     /* TDX_CST event data block */

/* Open board 1 channel 1 device */
if ((dev = dx_open("dxxxB1C1", 0)) == -1)
{
   printf("Cannot open channel dxxxB1C1”);
   exit(1);
}

/* Simulate an incoming call */
cst.cst_event = DE_RINGS;
cst.cst_data  = 0;

/* Put the event on the event queue */
if (sr_putevt(dev, TDX_CST, sizeof(DX_CST), &cst, 0) == -1)
{
   printf("_sr_putevt failed - %s", ATDV_ERRMSGP(SRL_DEVICE)));
   .
   .
   .
}
```

See Also

- sr_waitevt( )

sr_RemoveFromThreadDeviceGroup( )

Name: Linux: long sr_RemoveFromThreadDeviceGroup (*Devices, NumDevices)
Windows®: int sr_RemoveFromThreadDeviceGroup (*Devices, NumDevices)

Inputs: long *Devices — pointer to list of device handles
int NumDevices — number of devices in the list

Returns: 0 on success
-1 on failure

Includes: srllib.h

Category: Device Grouping functions

Mode: Synchronous

Description

The **sr_RemoveFromThreadDeviceGroup( )** function removes the listed devices from the grouping established for the thread. If any of the devices do not exist in the thread's grouping, no action is taken for those devices.

Parameter	Description
Devices	pointer to a list of device handles
NumDevices	number of devices in the list

Cautions

Devices should be idle before being removed from a group; otherwise events may accumulate unnecessarily in the Standard Runtime Library's event queue. This can result in a significant memory leak. If the application cannot guarantee that a device is idle before removing a device from a group, the application must call **sr_waitevt( )** to capture events for the device once it is removed from the group.

Errors

If this function returns -1 to indicate failure, obtain the reason for the error by calling the standard attribute function **ATDV_LASTERR( )** or **ATDV_ERRMSGP( )** to retrieve either the error code or a pointer to the error description, respectively.

Example

```
EventPollThread ()
{
   long Devices [24];
   int DevNum;
   long EventHandle;
```

```
   for (DevNum = 0; DevNum < 24; DevNum++)
   {
      Devices [DevNum] = dx_open(...);
   }

   sr_CreateThreadDeviceGroup (Devices);
   while (1)
   {
      sr_WaitThreadDeviceGroup (-1);
      // do something with the event
      if (done == true)
      {
         break;
      }
   }
   sr_RemoveFromThreadDeviceGroup(Devices, 24);
}
```

See Also

- sr_CreateThreadDeviceGroup( )

sr_setparm( )

Name: long sr_setparm(dev, parmno, parmval)

Inputs:

long dev	• device handle
long parmno	• parameter number
void *parmval	• pointer to parameter value

Returns: 0 if successful
-1 if failure

Includes: srllib.h

Category: Standard Runtime Library Parameter functions

Mode: Synchronous

Description

The **sr_setparm**( ) function allows the application to set the value of a Standard Runtime Library parameter. Usually, this function's parameters govern the mode of operation for eventing and synchronization. The function parameters are described as follows:

Parameter	Description
dev	device handle. Generally, you should set this parameter to SRL_DEVICE, which is the predefined Standard Runtime Library device handle. However, if the parameter being set is SR_USERCONTEXT, then **dev** should be set to the handle returned by the technology-specific **xx_open**( ).
parmno	specifies the value for the Standard Runtime Library parameter to be changed. Possible values are as follows: • SR_MODEID – **Linux:** set event notification mode parameter to SR_POLLMODE (default) • SR_MODELTYPE – **Windows® only:** set the model type to turn off creation of internal thread used to service event handler action • SR_USERCONTEXT – set user-specific context. This lets you quickly set application-specific context on a given device handle. • SR_WIN32INFO – **Windows®:** set the Win32 integration mode (See below.)

Parameter	Description
parmval	A pointer to an area of memory that contains the value for the specified parameter. • SR_MODEID – **Linux:** The value is expected to point to an integer that contains SR_POLLMODE. • SR_MODELTYPE – **Windows® only:** The value is expected to point to an integer that contains SR_STASYNC or SR_MTASYNC. • SR_USERCONTEXT – The value is expected to point to arbitrary user-supplied data. • SR_WIN32INFO – **Windows®:** The value is expected to point to either an SRLWIN32INFO structure that indicates the Win32 synchronization method or to NULL to disable Win32 notification.

Cautions

Normally, when setting Standard Runtime Library parameters, you must set the **dev** parameter to SRL_DEVICE. However, if you set the **parmno** parameter to SR_USERCONTEXT, you must set the **dev** parameter to the device on which context is being retrieved. That is, **dev** should be set to the handle returned by the technology-specific **xx_open( )**.

Errors

Linux: If the function returns -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.

Windows®: If the function returns -1 to indicate failure, obtain the reason for the error by calling the standard attribute function **ATDV_LASTERR(SRL_DEVICE)** or **ATDV_ERRMSGP(SRL_DEVICE)** to retrieve either the error code or a pointer to the error description, respectively. One of the following errors may be returned:

ESR_SYS
Error from operating system; use **dx_fileerrno( )** to obtain error value.

Linux 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 );
   }
}
```

Windows® Example A

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

main()
{
   int mode = SR_STASYNC;

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

Windows® Example B

The following example calls the **sr_setparm( )** function with its **parmno** parameter set to SR_USERCONTEXT:

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

//
// 4 devices maximum
//
#define MAXDEVICECOUNT  4

//
// Per device structure, one for each device
//
APPDEVICESTRUCT AppDeviceStruct[MAXDEVICECOUNT];
BOOL OpenAllDevices()
{
   ULONG DeviceIndex;
   CHAR DeviceName[16];
   INT  hDevice;
   for (DeviceIndex = 0; DeviceIndex < MAXDEVICECOUNT; DeviceIndex++)
   {
      //
      // Build the device name and open it
      //
      sprintf(DeviceName,"dxxxB1C%d", DeviceIndex+1);
      hDevice = dx_open(DeviceName, 0);
      if (hDevice == -1)
      {
         return(FALSE);
      }
      //
      // Now store away device handle and save device index on device
      // handles user context
      // This way given the device handle I get back to the device structure
      //  and vice versa
      AppDeviceStruct[DeviceIndex].hDevice = hDevice;
      if (sr_setparm(hDevice, SR_USERCONTEXT, (void *)&DeviceIndex) == -1)
      {
         //
         // perform clean up and return
         //
```

```
            return(FALSE);
         }
      } // End of for loop
      return(TRUE);
}

//
// Main loop processing showing how to retrieve SR_USERCONTEXT
//
APPDEVICESTRUCT * WaitEvent()
{
   ULONG DeviceIndex;

   //
   // Wait for any event
   //
   sr_waitevt(-1);

   //
   // got event so get device structure and return. Sr_getevtdev()
   // returns the device handle. DeviceIndex retrieves the
   // applications Index for this device. Of course the pointer
   // to the AppDeviceStruct could have been stored directly as well
   //
   sr_getparm(sr_getevtdev(), SR_USERCONTEXT, (void *)&DeviceIndex);

   return(&AppDeviceStruct[DeviceIndex]);
}
```

Windows® Example C

The following example uses Win32 notification through an I/O Completion Port:

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

#define DIALOGIC_KEY  1

SRLWIN32INFO  AppWin32Info;
HANDLE hCompletionPort;

BOOL CreateAndRegisterEventNotification()
{

   hCompletionPort = CreateIoCompletionPort( (HANDLE)NULL,  // no handle
                                             (HANDLE)NULL,  // it's new
                                              0,            // no key
                                              0);           // scaling

  // set up the information for SRL
   AppWin32Info.dwTotalSize = sizeof(SRLWIN32INFO);
   AppWin32Info.ObjectHandle = hCompletionPort;
   AppWin32Info.UserKey = DIALOGIC_KEY;
   AppWin32Info.dwHandleType = SR_IOCOMPLETIONPORT;
   AppWin32Info.lpOverlapped = (LPOVERLAPPED)NULL;
   sr_setparm(SRL_DEVICE, SR_WIN32INFO, (void *)&AppWin32Info);

  //
  // now add all the Win32 devices to the Completion Port assigning each
  // one a unique key
  //

   return(TRUE);
}
```

```
BOOL WaitForCompletion()
{
   DWORD UserKey;
   DWORD NumberOfBytesTransferred;
   LPOVERLAPPED lpOverlapped;
   BOOL bStatus;

  // block waiting for the event
   bStatus = GetQueueCompletionStatus (  hCompletionPort,
                                         &NumberOfBytesTransferred,
                                         &UserKey,
                                         &lpOverlapped,
                                         INFINITE );
   if (bStatus == FALSE)
   {
      return(bStatus);
   }

   switch (UserKey)
   {
      case DIALOGIC_KEY:
        // get the event of the SRL event queue
         sr_waitevt(0);
        //
        // use the sr_getevtxxx()functions now to get event information
        //
         break;
      default:
        // notification on some other win32 device, process accordingly
         break;
   }
   return(TRUE)
}
```

See Also

- **sr_getparm( )**

sr_waitevt( )

Name: long sr_waitevt(timeout)

Inputs: long timeout • timeout in milliseconds (msec)

Returns: >=0 indicates success; 0 indicates immediate occurrence, and a positive value indicates elapsed time in milliseconds from function execution to occurrence of the event
<0 indicates failure due to timeout or other reason

Includes: srllib.h

Category: Event Handling functions

Mode: Synchronous

Description

The **sr_waitevt( )** function waits for any event to occur, for a specified period of time, on any device.

The function completes successfully when an event occurs within the timeout duration. In this case, the function returns zero or a positive value to indicate success (a positive value indicates the elapsed time in milliseconds from execution of the function to occurrence of the event).

If the function fails to complete successfully, it may be due to a timeout or an error. In these cases, the function returns a negative value.

- If the function returns -1, this indicates a timeout.
- On Windows®, the function can return -2 to indicate an operating system error; call **dx_fileerrno( )** to obtain the error value.

Parameter	Description
timeout	Specifies a duration (timeout) in milliseconds for the function to wait for an event. A value of -1 specifies an infinite duration. A value of 0 specifies that a Standard Runtime Library event must occur immediately upon execution, regardless of whether or not an event is present in the Standard Runtime Library event queue.

Cautions

- The **sr_waitevt( )** and **sr_waitevtEx( )** functions should not be used in the same application.
- Use of **sr_waitevt( )** is recommended for single-threaded applications; it provides better performance for single-threaded applications as opposed to passing all of the devices to **sr_waitevtEx( )**. Use of **sr_waitevtEx( )** is recommended for multithreaded applications.
- 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.

- The application should not call blocking functions in the event handling loop.
- The **sr_waitevt( )** function cannot be called from within a handler.
- **Linux:** Note that the current implementation of **sr_waitevt( )** uses the Linux **time( )** function, which only has a granularity of 1 second.
- **Windows®:** If you use the **sr_waitevt( )** function to retrieve events, pass 0 as a parameter when using the following functions:
  - **sr_getevtdatap( )**
  - **sr_getevtdev( )**
  - **sr_getevtlen( )**
  - **sr_getevttype( )**

Errors

Linux: If the function returns -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.

Windows®: This function returns a negative value to indicate a failure.

-1
Indicates a timeout.

-2
Indicates an error from the operating system; call **dx_fileerrno( )** to obtain the error value.

Linux Example

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

long dx_handler(void *parm)
{
   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 );
   }

/* 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 );
}
```

Windows® Example

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

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

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

main()
{
   int dxxxdev;
   int mode = SR_STASYNC;
```

```
   /* Set SRL to turn off creation of internal thread */
    if( sr_setparm( SRL_DEVICE, SR_MODELTYPE, &mode ) == -1 )
    {
       printf( "Error: cannot set srl mode\n" );
       exit( 1 );
    }

   /* Open 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 );
    }

   /* 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 );
}
```

See Also

- **sr_CreateThreadDeviceGroup( )**
- **sr_GetThreadDeviceGroup( )**
- **sr_waitevtEx( )**
- **sr_WaitThreadDeviceGroup( )**

sr_waitevtEx( )

Name: long sr_waitevtEx(handlep, handlecnt, timeout, event_handlep)

Inputs:

long *handlep	• pointer to array of handles to wait for event on
int handlecnt	• count of valid handles pointed to by **handlep**
long timeout	• timeout in milliseconds (msec)
long *event_handlep	• pointer where to return event handle

Returns: >=0 success; where 0 indicates immediate occurrence, and a positive value indicates elapsed time in milliseconds from function execution to occurrence of the event
<0 indicates failure due to timeout or other reason

Includes: srllib.h

Category: Event Handling functions

Mode: Synchronous

Description

The **sr_waitevtEx( )** function is used to asynchronously wait for events on certain devices determined by passing an array of the device handles to wait for events on. It offers developers a different approach for event retrieval across multiple threads. This function is an extension of the standard **sr_waitevt( )** function and provides the convenience to the calling thread to wait for events on certain devices.

Note: Using **sr_waitevtEx( )** does **not** result in any performance improvements over using **sr_waitevt( )**. **sr_waitevt( )** will likely yield higher performance, as it bypasses the extra work performed by **sr_waitevtEx( )** in matching the events to devices.

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

Functions initiated asynchronously from one thread can be completed and have the event returned by **sr_waitevtEx( )** running in another thread. An application might also have multiple threads blocked in **sr_waitevtEx( )** waiting for events on the same device, however it is indeterminate which thread picks up the event, so each thread should be running the same state machine.

Parameter	Description
handlep	points to an array of opened handles
handlecnt	indicates how many devices are contained in the array

Parameter	Description
timeout	specifies a duration (timeout) in milliseconds for the function to wait for an event. A value of -1 specifies an infinite duration. A value of 0 specifies that a Standard Runtime Library event must occur immediately upon execution, regardless of whether or not an event is present in the Standard Runtime Library event queue.
event_handlep	When an event is reported, an event handle is passed backed to the application through the **event_handlep** argument. The value returned here should be passed to the **sr_getevtdev( )**, **sr_getevttype( )**, **sr_getevtlen( )**, and **sr_getevtdatap( )** functions to retrieve the event information.

The **sr_waitevtEx( )** function detects events from devices opened through any of the device-specific **xx_open( )** function calls for that device. The function completes successfully when an event occurs within the timeout duration. In this case, the function returns zero or a positive value to indicate success (a positive value indicates the elapsed time in milliseconds from execution of the function to occurrence of the event).

If the function fails to complete successfully, it may be due to a timeout or an error. In these cases, the function returns a negative value.

Cautions

- The **sr_waitevt( )** and **sr_waitevtEx( )** functions should not be used in the same application.
- Use of **sr_waitevt( )** is recommended for single-threaded applications; it provides better performance for single-threaded applications as opposed to passing all of the devices to **sr_waitevtEx( )**.
- **sr_waitevtEx( )** can **NOT** be used in conjunction with handlers.
- The same device should not be used across multiple threads.
- The application should not call blocking functions in the event handling loop.

Errors

This function returns a negative value to indicate a failure. Follow these instructions to discover the cause:

Linux: If the function returns -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.

Windows®: The function returns the following values on Windows®:

-1 (SR_TMOUT)
Indicates either a timeout or an API error. To discover which, call **ATDV_LASTERR**(SRL_DEVICE), which will return an appropriate error code. If **ATDV_LASTERR**(SRL_DEVICE) returns ESR_NOERR, then the failure is due to a

timeout, and you can confirm this with a call to **sr_getevttype**(SRL_DEVICE), which should return the current event type of SR_TMOUTEVT.

-2

Error from operating system; use **dx_fileerrno**( ) to obtain error value.

-3

Bad parameter; occurs if **handlep** or **event_handlep** is NULL or if **handlecnt** is 0.

Example

```
#include <windows.h>   /*  Windows apps only  */
#include <srllib.h>
#include <dxxxlib.h>

long  chdev[MAXDEVS];
unsigned long  evt_handle;

main( ... )
{
   char channel_name[12];
   int ch;

   for (ch = 0; ch < MAXDEVS; ch++)
   {
      /* Build the channel name for each channel */
      if ( (chdev[ch] = dx_open(channel_name, 0) ) == -1 )
      {
         printf("dx_open failed\n");
         exit(1);
      }
   }

  /*
   * Now initialize each device setting up the event masks and then issue
   * the command asynchronously to start off the state machine.
   */
   for (ch = 0; ch < MAXDEVS; ch++)
   {
     /* Set up the event masks and other initialization */

     /* set the channel onhook asynchronously */
      if (dx_sethook( chdev[ch]. DX_ONHOOK, EV_ASYNC) == -1)
      {
        /* sethook failed, handle the error */
      }
   }

  /* This is the main loop to control the Voice hardware */
   while (FOREVER)
   {
     /* wait for the event */
      sr_waitevtEx( chdev, MAXDEVS, -1, &evt_handle);
      process_event( evt_handle);
   }

   int process_event( ehandle)
   unsigned long ehandle;
   {
      int  voxhandle = sr_getevtdev(ehandle);
```

```
        switch(sr_getevttype(ehandle))
        {
           case TDX_CST:
                .
              break;

           case TDX_PLAY:
                .
              break;
        }
     }
}
```

See Also

- **sr_CreateThreadDeviceGroup( )**
- **sr_getevtdatap( )**
- **sr_getevtdev( )**
- **sr_getevtlen( )**
- **sr_getevttype( )**
- **sr_GetThreadDeviceGroup( )**
- **sr_waitevt( )**
- **sr_WaitThreadDeviceGroup( )**
- The appropriate library-specific Programming Guide

sr_WaitThreadDeviceGroup( )

Name: Linux: long sr_WaitThreadDeviceGroup (TimeOut)
Windows®: int sr_WaitThreadDeviceGroup (TimeOut)

Inputs: long TimeOut • time to wait for an event (msec)

Returns: 0 on success
-1 on timeout

Includes: srllib.h

Category: Device Grouping functions

Mode: Synchronous

Description

The **sr_WaitThreadDeviceGroup( )** function retrieves events for the devices previously specified in the thread by **sr_CreateThreadDeviceGroup( )**.

Parameter	Description
TimeOut	number of milliseconds to wait for an event. Values include: • -1 – wait infinitely • 0 – return immediately • >0 – wait for the specified number of milliseconds

Cautions

sr_CreateThreadDeviceGroup( ) must have been called in the same thread prior to calling this function.

Errors

If this function returns -1 to indicate failure, obtain the reason for the error by calling the standard attribute function **ATDV_LASTERR( )** or **ATDV_ERRMSGP( )** to retrieve either the error code or a pointer to the error description, respectively. One of the following errors may be returned:

ESR_THREAD_DEVICE_GROUP_NO_GROUP_DEFINED
sr_CreateThreadDeviceGroup( ) was not called prior to calling this function.

Example

```
EventPollThread ()
{
   long Devices [24];
   int DevNum;
   long EventHandle;
   for (DevNum = 0; DevNum < 24; DevNum++)
   {
      Devices [DevNum] = dx_open(...);
   }
```

```
   sr_CreateThreadDeviceGroup (Devices, 24);
   while (1)
   {
      sr_WaitThreadDeviceGroup (-1);
      // do something with the event
      if (done == true){
         break;
      }
   }
   sr_RemoveFromThreadDeviceGroup(Devices, 24);
}
```

See Also

- **sr_CreateThreadDeviceGroup( )**

SRLGetAllPhysicalBoards( )

Name: long SRLGetAllPhysicalBoards(**IN OUT** *piNum
OUT *pPhysicalBoards)

Inputs: int *piNum • pointer to the number of physical boards

AUID *pPhysicalBoards • pointer to physical board

Returns: ESR_NOERR to indicate success
ESR_INSUFBUF to indicate ***piNum** specified an insufficient board array

Includes: srllib.h

Category: Device Mapper functions

Mode: Synchronous

Description

The **SRLGetAllPhysicalBoards( )** function returns a list of all physical boards in the system.

This function returns the AUIDs for all boards in the system, regardless of whether they have been downloaded. This list also includes boards that are disabled or uninitialized. However, it does not return boards that are in the INIT state.

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.

Parameter	Description
***piNum**	specifies the user allocated length of an array of **pPhysicalBoards** ***piNum** may be set to NULL; in that case, **pPhysicalBoards** must be 0.
pPhysicalBoards	an opaque identifier for an important object in the system; in this case, a physical board

The 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 subdevices.
- A virtual board *is* an R4 device.
- A subdevice *is* an R4 device.
- One or more jacks *are associated with* one or more R4 devices.

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.

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.

Example

```
#include <srllib.h>

...

   AUID *pAU;
   int iNumPhyBds;
   long retVal;

   iNumPhyBds = 0;
   retVal = SRLGetAllPhysicalBoards(&iNumPhyBds, 0);
   if (ESR_INSUFBUF == retVal)
   {
      if ( iNumPhyBds != 0 )
         // success getting number of boards
         pAU = (AUID *)malloc(iNumPhyBds*sizeof(AUID));

         if (pAU)
         {
            retVal = SRLGetAllPhysicalBoards(&iNumPhyBds,pAU);
            if (ESR_NOERR == retval)
            {
               // success getting requested information
            }
         }
      }
```

See Also

- **SRLGetVirtualBoardsOnPhysicalBoard( )**

SRLGetJackForR4Device( )

Name: long SRLGetJackForR4Device(**IN** *szR4Device
OUT *piJackNum)

Inputs: char *szR4Device • R4 device
int *piJackNum • pointer to jack number

Returns: ESR_NOERR to indicate success
ESR_BADDEV to indicate that this R4 device does not have an associated jack

Includes: srllib.h

Category: Device Mapper functions

Mode: Synchronous

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:

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

The 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 subdevices.
- A virtual board *is* an R4 device.
- A subdevice *is* an R4 device.
- One or more jacks *are associated with* one or more R4 devices.

Cautions

None

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.

Example

```
#include <srllib.h>

...

   char szVB[] = "dtiB7";
   long retVal;
   int iJack;

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

See Also

- **SRLGetVirtualBoardsOnPhysicalBoard( )**
- **SRLGetSubDevicesOnVirtualBoard( )**

SRLGetPhysicalBoardName( )

Name: long SRLGetPhysicalBoardName(physicalBoard, *piLen, *szName)

Inputs:

AUID physicalBoard	• AUID of the physical board
int *piLen	• maximum length of the szName string
char *szName	• physical device name

Returns: ESR_NOERR to indicate success
ESR_BADDEV to indicate physicalBoard specified an invalid AUID
ESR_INSUFBUF to indicate *piLen specified an insufficient buffer length

Includes: srllib.h

Category: Device Mapper functions

Mode: Synchronous

Description

The **SRLGetPhysicalBoardName( )** function takes an AUID in **physicalBoard** and returns the board name associated with that AUID in string **szName**. The AUID passed in can be obtained from the **SRLGetAllPhysicalBoards( )** function.

Before you call **SRLGetPhysicalBoardName( )**, you must set the **piLen** parameter to the maximum size of string **szName**. The **piLen** parameter is set to the actual length of the string upon successful function return. The physical board name has the format "brdB**n**", where **n** stands for any number starting with 1.

Parameter	Description
physicalBoard	an opaque identifier (AUID) for an important object in the system; in this case, a physical board
***piLen**	length of the **szName** string
***szName**	pointer to the string which contains the physical board name

Cautions

None

Errors

If *piLen is insufficient, **SRLGetPhysicalBoardName( )** returns ESR_INSUFBUF. If an invalid device is specified, the function returns ESR_BADDEV.

Example

```
#include <srllib.h>

...
```

```
AUID physicalAUID[5];
int count = 5;
int len;
char physicalName[10];
...

/* Get all physical AUIDs */
SRLGetAllPhysicalBoards(&count, physicalAUID);
for (iBrd = 0; iBrd < count; iBrd++)
{
    len = 10;
    SRLGetPhysicalBoardName(physicalAUID[iBrd], &len, physicalName);
    printf("Name of board #%d is %s\n", iBrd, physicalName)
}
```

See Also

- **SRLGetAllPhysicalBoards( )**

SRLGetSubDevicesOnVirtualBoard( )

Name: long SRLGetSubDevicesOnVirtualBoard(**IN** *szVirtualBoard
IN OUT *piNum
OUT *pInfo)

Inputs:	char *szVirtualBoard	• parent virtual board
	int *piNum	• pointer to the number of subdevices
	SRLDEVICEINFO *pInfo	• array of subdevices

Returns: ESR_NOERR to indicate success
ESR_INSUFBUF to indicate ***piNum** specified an insufficient subdevice array

Includes: srllib.h

Category: Device Mapper functions

Mode: Synchronous

Description

The **SRLGetSubDevicesOnVirtualBoard( )** function returns a list of subdevices on a virtual board from the virtual board's R4 device string.

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 subdevices.

The function parameters are described as follows:

Parameter	Description
***szVirtualBoard**	specifies the name of the parent virtual board to be used for this function call
***piNum**	specifies the user allocated length of an array of SRLDEVICEINFOs (**pInfo**) ***piNum** may be set to NULL; in that case, **pInfo** must be 0.
***pInfo**	pointer to structure that contains information about an R4 device; see SRLDEVICEINFO for details.

The 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 subdevices.
- A virtual board *is* an R4 device.

- A subdevice *is* an R4 device.
- One or more jacks *are associated with* one or more R4 devices.

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.

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.

Example

This example shows that you execute **SRLGetSubDevicesOnVirtualBoard( )** no more than two times: once to get the number of subdevices and (optionally) one more time after allocating the buffer to get the SRLDEVICEINFO structures.

```
#include <srllib.h>

...

   char szVB[] = "dtiB7";
   SRLDEVICEINFO *pVBInfo;
   int iNumSDs;
   long retVal;

   iNumSDs = 0;
   retVal = SRLGetSubDevicesOnVirtualBoard(szVB, &iNumSDs, 0);
   if (ESR_INSUFBUF == retVal )
   {
      // iNumSDs cannot be 0 at this point
      pVBInfo = (SRLDEVICEINFO *)malloc(iNumSDs * sizeof(*pVBInfo));

      if (0 !=  pVbInfo)
      {
         retVal = SRLGetSubDevicesOnVirtualBoard(szVB, &iNumSDs, pVBInfo);
         if (ESR_NOERR == retVal)
         {
            // Success: pVBInfo contains requested information
            ...
         }

      }

   }
```

See Also

- **SRLGetVirtualBoardsOnPhysicalBoard( )**
- **SRLGetJackForR4Device( )**

SRLGetVirtualBoardsOnPhysicalBoard( )

Name: long SRLGetVirtualBoardsOnPhysicalBoard(**IN** AUID physicalBoard
IN OUT *piNum
OUT *pInfo)

Inputs:	AUID physicalBoard	• parent physical board
	int *piNum	• pointer to the number of virtual boards
	SRLDEVICEINFO *pInfo	• array of virtual boards

Returns: ESR_NOERR to indicate success
ESR_INSUFBUF to indicate ***piNum** specified an insufficient virtual board array

Includes: srllib.h

Category: Device Mapper functions

Mode: Synchronous

Description

The **SRLGetVirtualBoardsOnPhysicalBoard( )** function returns a list of virtual boards on a physical board. The user of this function is responsible for allocating an array of SRLDEVICEINFOs (**pInfo**) of length ***piNum**. If ***piNum** is insufficient, this function returns ESR_INSUFBUF. If the return code is ESR_NOERR or ESR_INSUFBUF, ***piNum** on output is the actual number of virtual boards.

Parameter	Description
physicalBoard	an opaque identifier for an important object in the system; in this case, the parent physical board
***piNum**	specifies the user allocated length of an array of SRLDEVICEINFOs (**pInfo**) ***piNum** may be set to NULL; in that case, **pInfo** must be 0.
pInfo	pointer to structure that contains information about an R4 device; see SRLDEVICEINFO for details.

The 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 subdevices.
- A virtual board *is* an R4 device.
- A subdevice *is* an R4 device.
- One or more jacks *are associated with* one or more R4 devices.

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.

Errors

You are responsible for allocating an array of SRLDEVICEINFOs (**pInfo**) of length ***piNum**. If ***piNum** is insufficient, **SRLGetVirtualBoardsOnPhysicalBoard( )** returns ESR_INSUFBUF.

Example

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

   AUID phyBd;
   SRLDEVICEINFO *pVBInfo;
   int iNumVBs;
   long retVal;

   // phyBd assigned
   iNumVBs = 0
   pVBInfo = 0;
   retVal = SRLGetVirtualBoardsOnPhysicalBoard(phyBd,&iNumVBs,0);
   if  ( ESR_INSUFBUF == retVal )
   {
      // success getting number of boards
      pVBInfo = (SRLDEVICEINFO *)malloc(iNumPhyBds*sizeof(SRLDEVICEINFO));
      if (pVBInfo)
      {
         retVal = SRLGetVirtualBoardsOnPhysicalBoard(phyBd,&iNumVBs,pVBInfo);
         if (ESR_NOERR == retVal)
         {
            // success getting requested information
         }
      }
   }
```

See Also

- **SRLGetAllPhysicalBoards( )**
- **SRLGetSubDevicesOnVirtualBoard( )**
- **SRLGetJackForR4Device( )**

Events 3

This chapter provides information on events that may be returned by the Dialogic® Standard Runtime Library (SRL) software.

There is one library-specific event for the Standard Runtime Library:

SR_TMOUTEVT
Timeout event - occurs on the Standard Runtime Library device

Data Structures 4

This chapter lists the data structures used by Dialogic® Standard Runtime Library (SRL) functions. These structures are used to control the operation of functions and to return information. In this chapter, the data structure definition is followed by a detailed description of the fields in the data structure. The fields are listed in the sequence in which they are defined in the data structure.

SRLDEVICEINFO

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

Description

This structure is used for device information. Upon successful return of the **SRLGetSubDevicesOnVirtualBoard( )** function, the structure contains a list of subdevices. Upon successful return of the **SRLGetVirtualBoardsOnPhysicalBoard( )** function, the structure contains a list of virtual boards.

Field Descriptions

szDevName
name of the device

iDevType
device type. Valid values are:

- TYPE_R4_VOX_BOARD – R4 voice virtual board
- TYPE_R4_VOX_CHANNEL – R4 voice virtual channel
- TYPE_R4_DTI_BOARD – R4 DTI virtual board
- TYPE_R4_DTI_TIMESLOT – R4 DTI time slot
- TYPE_R4_MSI_BOARD – R4 MSI virtual board
- TYPE_R4_MSI_STATION – R4 MSI station
- TYPE_R4_FAX_BOARD – R4 fax virtual board
- TYPE_R4_FAX_CHANNEL – R4 fax virtual channel
- TYPE_R4_VR_BOARD – R4 voice recognition virtual board
- TYPE_R4_VR_CHANNEL – R4 voice recognition channel
- TYPE_R4_BRI_BOARD – R4 BRI virtual board
- TYPE_R4_BRI_TIMESLOT – R4 BRI time slot
- TYPE_R4_SCX_BOARD – R4 virtual SCX channel
- TYPE_R4_DCB_BOARD – R4 DCB virtual board
- TYPE_R4_DPD_BOARD – R4 DPD virtual board
- TYPE_R4_DPD_CHANNEL – R4 DPD channel
- TYPE_R4_TTS_BOARD – R4 TTS virtual board
- TYPE_R4_TTS_CHANNEL – R4 TTS channel
- TYPE_R4_DMX_BOARD – R4 DMX virtual board
- TYPE_R4_CSP_BOARD – R4 continuous speech processing virtual board
- TYPE_R4_CSP_CHANNEL – R4 continuous speech processing channel
- TYPE_R4_BRI_DATA_BOARD – R4 BRI data virtual board
- TYPE_R4_BRI_DATA_CHANNEL – R4 BRI data channel
- TYPE_R4_IPT_BOARD – R4 IPT virtual board
- TYPE_R4_IPT_TIMESLOT – R4 IPT time slot
- TYPE_R4_IPM_BOARD – R4 IPM virtual board
- TYPE_R4_IPM_CHANNEL – R4 IPM channel
- TYPE_R4_MOH_BOARD – R4 audio input virtual board (for music on hold)
- TYPE_R4_PHYSICAL_BOARD – R4 physical board

Error Codes 5

This chapter describes the error/cause codes supported by the Dialogic® Standard Runtime Library (SRL). All Standard Runtime Library functions return a value that indicates the success or failure of the function call. Success is indicated by a return value of zero or a non-negative number. Failure is indicated by a value of -1.

If a function fails, call the standard attribute functions **ATDV_LASTERR( )** to return the error code and **ATDV_ERRMSGP( )** to return a pointer to a descriptive error message.

The Standard Runtime Library contains the following error codes, listed in alphabetical order.

ESR_BADDEV
Invalid or missing device

ESR_BADPARM
Invalid parameter or parameter value

ESR_DATASZ
Invalid size for default event data memory

ESR_INSUFBUF
No or insufficient buffers available

ESR_MODE
Illegal mode for this operation

ESR_NOCOM
Standard Runtime Library cannot communicate with another subsystem

ESR_NOERR
No Standard Runtime Library errors

ESR_NOHDLR
No such handler

ESR_NOMEM
No or insufficient memory available

ESR_NULL_DATAP
Pointer argument is null

ESR_PARMID
Unknown parameter ID

ESR_QSIZE
Illegal event queue size

ESR_SCAN
Standard Runtime Library scanning function returned an error

ESR_SYS [Windows® only]
Error from operating system; use **dx_fileerrno( )** to obtain error value.

ESR_SYSTEM [Linux only]
System error, consult **errno** in *errno.h*

ESR_THREAD_DEVICE_GROUP_EXISTS
A thread device group has already been created for the thread

ESR_THREAD_DEVICE_GROUP_NO_GROUP_DEFINED
sr_CreateThreadDeviceGroup( ) was not called prior to calling this function

ESR_TMOUT
Returned by **ATDV_LASTERR**(SRL_DEVICE) when a Standard Runtime Library function timed out

Glossary

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

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

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

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

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

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

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

event: Any message sent from the device.

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

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

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

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

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

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

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

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

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

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

Index

Symbols

A

D

E

I

L

P

R

S

U

V