Voice Software Reference: Standard Runtime Library

for Windows

05-1458-002

COPYRIGHT NOTICE

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

Publication Date: September, 2000

Part Number: 05-1458-002

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

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

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

OPERATING SYSTEM SUPPORT

The term *Windows* refers to both the Windows® NT and Windows® 2000 operating systems. For a complete list of supported Windows operating systems, refer to the *Release Guide* that came with your Dialogic System Release for Windows, or to the Dialogic support site at *http://support.dialogic.com/releases*.

Table of Contents

1. SRL Overview

The following information is provided in this chapter:

- A description of this SRL Guide (*Section 1.1*).
- SRL Product Terminology (*Section 1.2*).
- A description of the major function of SRL (*Section 1.3*).
- A description of SRL event management functions (*Section 1.4*).
- A description of SRL standard attribute functions (*Section 1.5*).
- A description of the SRL termination parameter table (*Section 1.6*).

1.1. About This SRL Guide

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

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

- Event management functions
- Standard attribute functions

Chapter 1. SRL Overview provides a summary of the Dialogic Standard Runtime Library.

Chapter 2. Basic SRL Programming Concepts for Dialogic Products describes basic SRL programming.

Chapter 3. Basic SRL Programming Models provides guidelines for selecting the following programming models:

- Synchronous
- Asynchronous
- Extended Asynchronous

Chapter 4. Advanced SRL Programming Concepts for Dialogic *Products* describes advanced SRL programming.

Chapter 5. Advanced SRL Programming Models provides guidelines for selecting the following programming models:

- Synchronous with SRL Callback
- Asynchronous with SRL Callback
- Asynchronous with Windows Callback
- Asynchronous with Win32 Synchronization

Chapter 6. SRL Event Management Functions defines Event management functions and provides a functional description, cautions, and examples for each function.

Chapter 7. SRL Standard Attribute *Functions* defines standard attribute functions and provides a functional description, cautions, and examples for each function.

Chapter 8. DV_TPT Termination Parameter Table Structure defines the SRL data structure DV_TPT.

This guide assumes that you have a working knowledge of C programming in a Windows environment, and a basic understanding of multithreading.

1.2. SRL Product Terminology

The following terms are used throughout this guide:

D/xxx refers to Dialogic's series of voice processing boards. D/240SC, D/320SC, D/240SC-T1, and D/300SC-PCI are specific models of this board.

DTI/xxx refers in general to all of Dialogic's digital telephony interface boards. Specific models include the DTI/240SC and DTI/300SC boards.

SRL refers to the Dialogic Standard Runtime Library. This chapter provides an overview of the Dialogic Standard Runtime Library (SRL), provided as part of the Dialogic voice software for Windows.

1.3. SRL Description

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

The Dialogic SRL is a library that contains C functions and a data structure to support application development. These are:

- Event management C functions
- Standard attribute C functions
- Application context management
- A termination parameter table data structure

The SRL provides the following files:

- *srllib.h*
- *libsrlmt.lib*
- *libsrlmt.dll*
- *srllib.c*
- *srllib.cpp*

The multi-threaded library supports all basic and advanced SRL programming models. See *Chapter 3. Basic SRL Programming Models* and *Chapter 5. Advanced SRL Programming Models* to determine which model is best suited for your needs.

The SRL allows established Dialogic users to easily port their UNIX, OS/2, and MS DOS applications to the Windows environment. For new application developers, the SRL maximizes performance in the native Windows environment by providing:

- Tight integration with the Windows programming model
- New options for program development

1.4. The Cross-Compatibility Library

The C/C++ language libraries, *srllib.c* and *srllib.cpp*, are part of the Cross-Compatibility Library. With them, an application can perform runtime linking instead of compile-time linking. When using the cross-compatibility method, the application must call the function **sr_libinit( )** prior to calling any other Dialogic function. The **sr_libinit( )** function will load the SRL DLL and resolve the entry points to the DLL when the application is executed.

A C/C++ language interface library is also provided with each technology (voice, fax, and network interfaces).

1.5. SRL Event Management Functions

Event management functions provide an interface to manage events on devices, handling program flow associated with these different modes of application architecture. Application programmers can use event management functions to do the following:

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

Chapter 3. Basic SRL Programming Models and *Chapter 5. Advanced SRL Programming Models* contain detailed instructions for using the models in application development. Refer to *6. SRL Event Management Functions* for a description and complete reference for each of the event management functions.

1.6. SRL Standard Attribute Functions

Standard attribute functions return general information about a device, such as device name, board type, and the error that occurred on the last library call.

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

Chapter 7. SRL Standard Attribute Functions describes the standard attribute functions in detail.

1.7. Application Context Management

Application context management allows the application to set up and get user-specific context on a device-by-device basis. Two examples of user context are:

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

As detailed in *Section 6.4. SRL Event Management Function Reference Overview*, use the **sr_setparm( )** function to set user-specific context, and the **sr_getparm( )** function to get user-specific context.

1.8. SRL Termination Parameter Table

The SRL also includes the termination parameter table (TPT). The TPT is contained in the *srllib.h* file and consists of DV_TPT data structures that are used to specify termination conditions for Dialogic devices. *Chapter 8. DV_TPT Termination Parameter Table Structure* describes the DV_TPT structure in detail.

2. Basic SRL Programming Concepts for Dialogic Products

The following information is provided in this chapter:

- Definitions of SRL devices (*Section 2.1*).
- A description of SRL programming options (*Section 2.2*).
- A description of SRL synchronous programming (*Section 2.3*).
- A description of SRL asynchronous programming (*Section 2.4*).
- A description of SRL extended asynchronous programming (*Section 2.5*).

2.1. SRL Devices

In this SRL Guide, devices are addressable entities that can communicate through the SRL, such as:

- Voice channels
- Digital time slots
- FAX devices
- MSI station sets
- Board devices

2.2. SRL Programming Options

The SRL provides a rich set of programming options for developers in the Windows environment. The SRL works in conjunction with Windows to support:

- Standard synchronous and asynchronous programming.
- All combinations of Dialogic event handlers.
- Mixed programming environments that Dialogic products also support in other operating systems.

Some options available to the programmer include:

Support for pure asynchronous programming

Support for pure synchronous programming

Support for combined asynchronous and synchronous programming

Automated support for Dialogic event handling through a separate worker thread.

Extended asynchronous operation that allows multiple state machines to easily run on multiple device subgroups.

2.3. SRL Synchronous Programming

In synchronous programming, each function blocks thread execution until the function completes. The operating system can put individual device threads to sleep while allowing threads that control other Dialogic devices to continue their actions unabated. When a Dialogic function completes, the operating system wakes up the function's thread so that processing continues. For example, if the application is playing a file as a result of a **dx_play( )** function call, the calling thread does not continue execution until the play has completed and the **dx_play( )** function has terminated. For more information, see *Chapter 3. Basic SRL Programming Models.*

2.4. SRL Asynchronous Programming

In asynchronous programming, the calling thread performs further operations while the Dialogic function completes. At completion, the application receives event notification. Asynchronous programming is recommended for applications that require coordination of multiple tasks and have large numbers of devices. Asynchronous programming uses system resources more efficiently because it controls multiple devices in a single thread. On the other hand, synchronous programming requires a separate thread for each Dialogic device.

Due to concurrent processing requirements, a thread cannot block execution while waiting for Dialogic functions, such as **dx_play( )** or **dx_record( )**, to finish; this would interfere with the processing requirements of other devices being managed by the thread. In this case, the SRL lets you create an event-driven state machine for each device. Instead of each Dialogic function blocking until completion, it

returns immediately and allows thread processing to continue. Subsequently, when an event is returned through the SRL, signifying the completion of the Dialogic operation, state machine processing can continue.

You can also place user-defined events into the Dialogic event queue to get single-point state processing control of non-Dialogic application states.

IIn summary, asynchronous programming:

- Achieves a high level of resource management by combining multiple Dialogic devices in a single thread.
- Provides better control of applications that have high channel density.
- Provides several extended mechanisms that help you port Dialogic applications from other operating systems.
- Works with other SRL mechanisms that allow new developers to tightly integrate the SRL with standard Windows 32-bit programming mechanisms, such as the Win32 API and MFC.
- Reduces system overhead by minimizing thread context switching.
- Simplifies the coordination of events from many Dialogic devices.

2.5. SRL Extended Asynchronous Programming

Extended asynchronous programming is a variation of standard asynchronous programming. In extended asynchronous programming, you can create multiple threads, each of which controls multiple devices. In such an application, each thread has its own specific state machine for the devices that it controls. For example, you can have one grouping of devices that provides fax services and another grouping that provides interactive voice response (IVR) services, while both share the same process space and database resources.

For more information, see *Section 3.4. Extended Asynchronous Model.*

3. Basic SRL Programming Models

This chapter describes the following basic SRL programming models:

- Synchronous
- Asynchronous
- Extended Asynchronous

3.1. Selecting Basic SRL Programming Models

Basic programming models are recommended because you can apply them to almost all applications. Select a basic programming model according to the criteria shown in *Table 1*. The sections following this table describe these basic programming models.

Table 1. Guidelines for Selecting a Basic Programming Model

Application Requirements	Recommended Basic Programming Model	Threading and Event Handling Considerations
Few devices	Synchronous (*Section 3.2*)	Create a separate thread to execute processing for each Dialogic device.
Many devices Multiple tasks	Asynchronous (*Section 3.3*)	Call **sr_waitevt( )** to wait for events. Create a single thread to execute processing for all Dialogic devices.
Many devices Multiple tasks Needs to wait for events on more than one grouping of devices.	Extended Asynchronous (*Section 3.4*)	Call **sr_waitevtEx( )** for each grouping of devices, to wait for events on that grouping.

3.2. Synchronous Model

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

Choose the Synchronous model when you are programming an application that has:

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

Synchronous Model Advantages

- Because the Synchronous model is the easiest to program and maintain, it allows quicker deployment.

Synchronous Model Disadvantages

- Because the main thread creates a separate thread for each Dialogic device, this model requires a high level of system resources. This can limit maximum device density; thus, the Synchronous model provides limited scalability for growing systems.
- Because a synchronous Dialogic operation blocks thread execution, the thread cannot perform any other processing.
- Unsolicited events are not processed until the thread calls a Dialogic function such as **dx_getevt( )** or **dt_getevt( )**.

Synchronous Model Programming Notes

- You should use the Synchronous model only for simple and straight flow control, with only one action per device occuring at any time.

- Because each function in the Synchronous model blocks execution in its thread, the main thread in your application must create a separate thread for each Dialogic device.

Synchronous Model Example

The following example code shows a typical use of the Synchronous Programming model.

```
/*
 * This synchronous mode sample application was designed to work with
 * D/41ESC, VFX/40ESC, LSI/81SC, LSI/161SC and D/160SC-LS boards only.
 * It was compiled using MS-VC++.
 */

/* C includes */
#include <stdio.h>
#include <process.h>
#include <conio.h>
#include <ctype.h>
#include <windows.h>

/* Dialogic includes */
#include <srllib.h>
#include <dxxxlib.h>

/* Defines */
#define MAX_CHAN     4 /* maximun number of voice channels in system */

/* Globals */
int Thrd_num[MAX_CHAN];
int Kbhit_flag = 0;

/* Prototypes */
int main();
DWORD WINAPI sample_begin(LPVOID);


/*****************************************************************
*        NAME : int main()
* DESCRIPTION : prepare screen for ouput, create threads and
*             : poll for keyboard input
*       INPUT : none
*      OUTPUT : none
*     RETURNS : 0 on success; 1 if a failure was encountered
*    CAUTIONS : none
*****************************************************************/
int main()
{
   int cnt;
   HANDLE thread_handles[MAX_CHAN];
   DWORD threadID;

   /* show application's title */
   printf("Synchronous Mode Sample Application - hit any key to exit...\n");

   /* create one thread for each voice channel in system */
```

```
   for (cnt = 0; cnt < MAX_CHAN; cnt++) {
      Thrd_num[cnt] = cnt;
      if ((thread_handles[cnt] = (HANDLE)_beginthreadex(NULL,
                                 0,
                                 sample_begin,
                                 (LPVOID)&Thrd_num[cnt],
                                 0,
                                 &threadID)) == (HANDLE)-1) {
         /* Perform system error processing */
         exit(1);
      }
   }

   /* wait for Keyboard input to shutdown program */
   getch();

   Kbhit_flag++;  /* let threads know it's time to abort */

   /* sleep here until all threads have completed their tasks */
   if (WaitForMultipleObjects(MAX_CHAN, thread_handles, TRUE, INFINITE)
         == WAIT_FAILED) {
      printf("ERROR: Failed WaitForMultipleObjects(): error = %ld\n",
             GetLastError());
   }

   return(0);
}


/*************************************************************
*        NAME : DWORD WINAPI sample_begin(LPVOID argp)
* DESCRIPTION : do all channel specific processing
*       INPUT : LPVOID argp - pointer to the thread's index number
*      OUTPUT : none
*     RETURNS : 0 on success; 1 if a failure was encountered
*    CAUTIONS : none
*************************************************************/
DWORD WINAPI sample_begin(LPVOID argp)
{
   char channame[20];
   int chdesc;
   int thrd_num = *((int *)argp);

   /* build name of voice channel */
   sprintf(channame, "dxxxB%dC%d", (thrd_num / 4) + 1,
                                   (thrd_num % 4) + 1);

   /* open voice channel */
   if ((chdesc = dx_open(channame, 0)) == -1) {
      printf("%s - FAILED: dx_open(): system error = %d\n",
             channame, dx_fileerrno());
      return(1);
   }
   printf("%s - Voice channel opened\n", ATDV_NAMEP(chdesc));

   /* loop until Keyboard input is received */
   while (!Kbhit_flag) {
      /* set the voice channel off-hook */
      if (dx_sethook(chdesc, DX_OFFHOOK, EV_SYNC) == -1) {
         printf("%s - FAILED: dx_sethook(DX_OFFHOOK): %s (error #%d)\n",
                ATDV_NAMEP(chdesc), ATDV_ERRMSGP(chdesc), ATDV_LASTERR(chdesc));
         return(1);
```

```
      }
      printf("%s - Voice channel off-hook\n",ATDV_NAMEP(chdesc));

      /* dial number (without call progress) */
      printf("%s - Voice channel dialing...\n",ATDV_NAMEP(chdesc));
      if (dx_dial(chdesc, "12025551212", NULL, EV_SYNC) == -1) {
         printf("%s - FAILED: dx_dial(): %s (error #%d)\n",
                ATDV_NAMEP(chdesc), ATDV_ERRMSGP(chdesc), ATDV_LASTERR(chdesc));
         return(1);
      }
      printf("%s - Voice channel Done dialing\n",ATDV_NAMEP(chdesc));

      /* set the voice channel back on-hook */
      if (dx_sethook(chdesc, DX_ONHOOK, EV_SYNC) == -1) {
         printf("%s - FAILED: dx_sethook(DX_ONHOOK): %s (error #%d)\n",
                ATDV_NAMEP(chdesc), ATDV_ERRMSGP(chdesc), ATDV_LASTERR(chdesc));
         return(1);
      }
      printf("%s - Voice channel on-hook\n",ATDV_NAMEP(chdesc));
   }

   /* close the voice channel */
   if (dx_close(chdesc) == -1) {
      printf("%s - FAILED: dx_close(): %s (error #%d)\n",
             ATDV_NAMEP(chdesc), ATDV_ERRMSGP(chdesc), ATDV_LASTERR(chdesc));
      return(1);
   }
   printf("%s - Voice channel closed\n",channame);

         return(0);
}
```

3.3. Asynchronous Model

In the Asynchronous model, after the application issues an asynchronous function, it uses the **sr_waitevt( )** function to wait for events on Dialogic devices.

Choose the Asynchronous model for any application that:

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

Asynchronous Model Advantages

- Because the Asynchronous model uses one thread for all Dialogic devices, it requires a lower level of system resources than does the Synchronous model.
- The Asynchronous model lets you use a single thread to run the entire Dialogic portion of the application.

Asynchronous Model Disadvantages

- The Asynchronous model requires the development of a state machine, which is typically more complex to develop than is a synchronous application.

Asynchronous Model Programming Notes

- If an event is available, you can use the following functions to access information about the event:

sr_getevtdev( )	Get Dialogic device handle for the current event.
sr_getevttype( )	Get event type for the current event.
sr_getevtdatap( )	Get a pointer to additional data for the current event.
sr_getevtlen( )	Get the number of bytes of additional data that are pointed to by **sr_getevtdatap( )**.

- Use the **sr_getevtdatap( )** function to extract the event-specific data. Use the other functions to return values about the current event. The values returned are valid until **sr_waitevt( )** is called again.
- After the event is processed, the application determines what asynchronous function should be issued next; function issuance depends on what event has occurred, and on the last state of the device when the event occurred.

Asynchronous Model Example

An example of the Asynchronous model is shown below:

```
/*
 * This asynchronous mode sample application was designed to work with
 * D/41ESC, VFX/40ESC, LSI/81SC, LSI/161SC and D/160SC-LS boards only.
 * It was compiled using MS-VC++.
 * It cycles through channels going offhook, dialing a digit string,
 * going onhook. This is repeated until the user hits a keyboard key.
 * The thread to monitor the keyboard is peripheral to the main application
 * and the asynchronous programming mode and may be replaced with any other
 * mechanism the user may desire.
 */

/* C includes */
```

```
#include <stdio.h>
#include <process.h>
#include <string.h>
#include <conio.h>
#include <windows.h>
#include <winbase.h>

/* Dialogic includes */
#include <srllib.h>
#include <dxxxlib.h>
#include <sctools.h>  /* needed for nr_scroute() declaration */

/* Defines */
#define MAXCHAN     4 /* maximum number of voice channels in system */
#define USEREVT_KEYBOARD 1 /* User defined keyboard event */

/* This may be expanded to contain other information such as state */
typedef struct dx_info {
         int chdev;
} DX_INFO;

/* Globals */
DX_INFO dxinfo[MAXCHAN+1];
int Kbhit_flag = 0;

/* Prototypes */
int main();
DWORD WINAPI keyboard_monitor(LPVOID);
int process(int,int);

/*****************************************************************
*        NAME : int main()
* DESCRIPTION : prepare screen for output, create keyboard monitor
*                          :     thread and process asynchronous events received
*       INPUT : none
*      OUTPUT : none
*     RETURNS : 0 on success; 1 if a failure was encountered
*    CAUTIONS : none
*****************************************************************/
int main()
{
   int  numchan;
   char channame[20];
   HANDLE threadHdl;
   DWORD  ThrdID;
   int  evtdev, evttype;

   /* show application's title */
   printf("Asynchronous Mode Sample Application - hit any key to exit...\n");

   /* Create a thread for monitoring keyboard input */
   threadHdl = (HANDLE)_beginthreadex(NULL,
                                      0,
                                      keyboard_monitor,
                                      NULL,
                                      0,
                                      &ThrdID);
   if (threadHdl == (HANDLE) -1 ) {
          printf("Error creating keyboard monitor thread -- exiting\n");
          return(1);
   }
```

```
/* Initial processing for MAXCHANS */
for (numchan=1;numchan<=MAXCHAN;numchan++) {
      /* build name of voice channel */
      sprintf(channame, "dxxxB%dC%d", ((numchan-1) / 4) + 1,
                                      ((numchan -1)% 4) + 1);

      /* open voice channel */
      if ((dxinfo[numchan].chdev = dx_open(channame, 0)) == -1) {
      /* Perform system error processing */
            return(1);
      }

      /* Store numchan as USERCONTEXT for this device */
      sr_setparm(dxinfo[numchan].chdev,SR_USERCONTEXT,&numchan);

      printf( "Voice channel opened (%s)\n", ATDV_NAMEP(dxinfo[numchan].chdev));

      /* route voice channel to it's analog front end */
      if (nr_scroute(dxinfo[numchan].chdev,
                  SC_VOX,
                  dxinfo[numchan].chdev,
                  SC_LSI,
                  SC_FULLDUP) == -1) {
                  printf("FAILED: nr_scroute(%s): %s (error #%d)\n",
                              ATDV_NAMEP(dxinfo[numchan].chdev),
                              ATDV_ERRMSGP(dxinfo[numchan].chdev),
                              ATDV_LASTERR(dxinfo[numchan].chdev));
                  return(1);
      }
      printf( "Voice channel connected to analog front end\n");

      /* Start the application by putting the channel in onhook state */
      if (dx_sethook(dxinfo[numchan].chdev,DX_ONHOOK,EV_ASYNC) == -1) {
                  printf("FAILED: dx_sethook(%s): %s (error #%d)\n",
                              ATDV_NAMEP(dxinfo[numchan].chdev),
                              ATDV_ERRMSGP(dxinfo[numchan].chdev),
                              ATDV_LASTERR(dxinfo[numchan].chdev));
                  return(1);
      }

}

/* While no keyboard input, keep cycling through functions */
while (1) {
      /* Wait for events */
      sr_waitevt(-1);
      evtdev = sr_getevtdev();
      evttype = sr_getevttype();
      if ((evtdev ==  SRL_DEVICE) && (evttype == USEREVT_KEYBOARD))
                  break;
      if (process(evtdev, evttype) != 0)
                  break;


}

/* Close all voice devices before exiting */
for (numchan=1;numchan<=MAXCHAN;numchan++) {
       dx_close(dxinfo[numchan].chdev);
}

/* Wait here until thread exits */
```

```
   if (WaitForMultipleObjects(1, &threadHdl, TRUE, INFINITE) == WAIT_FAILED) {
         printf("ERROR: Failed WaitForMultipleObjects(): error = %ld\n",
                              GetLastError());
   }
   return(0);
}
/*****************************************************************************
 *        NAME: DWORD WINAPI keyboard_monitor( LPVOID argp )
 * DESCRIPTION: Wait for keyboard input
 *       INPUT: LPVOID argp
 *      OUTPUT: None
 *     RETURNS: none
 *    CAUTIONS: None
 *****************************************************************************/
DWORD WINAPI keyboard_monitor( LPVOID argp )
{
         getch();
         sr_putevt(SRL_DEVICE,USEREVT_KEYBOARD,0,NULL,0);
         return(0);
}

/*****************************************************************************
 *        NAME: int process( eventdev, event )
 * DESCRIPTION: Do the next function depending on the Event Received
 *       INPUT: int     eventdev;   - Device on which event was received
 *              int     event;      - Event being processed
 *      OUTPUT: None
 *     RETURNS: New Channel State
 *    CAUTIONS: None
 *****************************************************************************/
int process( eventdev, event )
   int eventdev;
   int event;
{
   DX_CST        *cstp;
   int            channum;

   /*
    * Retrieve USERCONTEXT for device
    */
    sr_getparm(eventdev, SR_USERCONTEXT, &channum);

   /*
    * Switch according to the event received.
    */
   switch ( event ) {

   case TDX_SETHOOK:
     cstp = (DX_CST *)sr_getevtdatap();
     switch( cstp->cst_event) {
       case DX_ONHOOK:
         /* Go offhook next */
         printf("Received onhook event\n");
         if (dx_sethook(dxinfo[ channum ].chdev, DX_OFFHOOK, EV_ASYNC) == -1) {
            printf("FAILED: dx_sethook(%s, DX_OFFHOOK): %s (error #%d)\n",
                        ATDV_NAMEP(dxinfo[channum].chdev),
                        ATDV_ERRMSGP(dxinfo[channum].chdev),
                        ATDV_LASTERR(dxinfo[channum].chdev));
                 return(1);
            }
            break;
```

```
      case DX_OFFHOOK:
        /* dial next */
        printf("Received offhook event\n");
        if (dx_dial(dxinfo[channum].chdev, "12025551212", NULL, EV_ASYNC) == -1) {
           printf("FAILED: dx_dial(%s): %s (error #%d)\n",
                       ATDV_NAMEP(dxinfo[channum].chdev),
                       ATDV_ERRMSGP(dxinfo[channum].chdev),
                       ATDV_LASTERR(dxinfo[channum].chdev));
                return(1);
        }
        break;
       }
       break;

  case TDX_DIAL:
        /* Next go onhook */
        printf("Received TDX_DIAL event\n");
        if (dx_sethook(dxinfo[ channum ].chdev, DX_ONHOOK, EV_ASYNC) == -1) {
                  printf("FAILED: dx_sethook(%s, DX_ONHOOK): %s (error #%d)\n",
                            ATDV_NAMEP(dxinfo[channum].chdev),
                            ATDV_ERRMSGP(dxinfo[channum].chdev),
                            ATDV_LASTERR(dxinfo[channum].chdev));
                  return(1);
         }

  }

  return(0);
}
```

3.4. Extended Asynchronous Model

This model is basically the same as the Asynchronous model, except that the application can control groupings of devices with separate threads. To do this, use **sr_waitevtEx( )**. See section

Choose the Extended Asynchronous model for any application that:

- Requires a state machine.
- Needs to wait for events on more than one grouping of devices.

Extended Asynchronous Model Advantages

- Because the Extended Asynchronous model uses only a few threads for all Dialogic devices, it requires a lower level of system resources than does the Synchronous model.
- The Extended Asynchronous model lets you use a few threads to run the entire Dialogic portion of the application.

Extended Asynchronous Model Disadvantages

- The Extended Asynchronous model requires the development of a state machine, which is typically more complex to develop than a synchronous application.

Extended Asynchronous Model Programming Notes

- The Extended Asynchronous model uses multiple threads and calls **sr_waitevtEx( )**. In contrast, the Asynchronous model uses only one thread, and calls **sr_waitevt( )**.
- If an event is available, you can use the following functions to access information about the event:

sr_getevtdev( )	Get Dialogic device handle for the current event.
sr_getevttype( )	Get event type for the current event.
sr_getevtdatap( )	Get a pointer to additional data for the current event.
sr_getevtlen( )	Get the number of bytes of additional data that are pointed to by **sr_getevtdatap( )**.

- Use the **sr_getevtdatap( )** function to extract the event-specific data; use the other functions to return values about the current event. The values returned are valid until **sr_waitevtEx( )** is called again.
- After the event is processed, the application determines what asynchronous function should be issued next depending on what event has occurred and the last state of the device when the event occurred.
- Do not use any Dialogic device in more than one grouping. Otherwise, it is impossible to determine which thread receives the event.
- Do not use **sr_waitevtEx( )** function in combination with either the **sr_waitevt( )** function or event handlers.

Extended Asynchronous Model Example

An example of the Extended Asynchronous model is shown below.

```
*************************************************************
*        NAME : int main()
* DESCRIPTION : create thread and poll for keyboard input
*       INPUT : none
*      OUTPUT : none
*     RETURNS : 0 on success; 1 if a failure was encountered
*    CAUTIONS : none
*************************************************************/
int main()
{
   HANDLE thread_handle[2];
   DWORD threadID;

   /* show application's title */
   printf("Extended Asynchronous Mode Sample Application - hit any key to exit...\n");

   /* create one thread to run one state machine */
   if ((thread_handle[0] = (HANDLE)_beginthreadex(NULL,
                                                  0,
                                                  StateMachine1,
                                                  (LPVOID)0,
                                                  0,
                                                  &threadID)) == (HANDLE)-1) {
         /* Perform system error processing */
         exit(1);
   }

   /* create a second thread to run the other state machine */
   if ((thread_handle[1] = (HANDLE)_beginthreadex(NULL,
                                                  0,
                                                  StateMachine2,
                                                  (LPVOID)2,
                                                  0,
                                                  &threadID)) == (HANDLE)-1) {
         /* Perform system error processing */
         exit(1);
   }

   /* wait for Keyboard input to shutdown program */
   getch();

   Kbhit_flag++;  /* let thread know it's time to abort */

   /* sleep here until thread has terminated */
   if (WaitForMultipleObjects(2, thread_handle, TRUE, INFINITE)
         == WAIT_FAILED) {
      printf("ERROR: Failed WaitForMultipleObjects(): error = %ld\n",
             GetLastError());
   }

   return(0);
}


/*************************************************************
*        NAME : DWORD WINAPI StateMachine1(LPVOID argp)
* DESCRIPTION : This tread runs the offhook-dial-onhook state machine
```

```
*       INPUT : LPVOID argp - NULL pointer (not used)
*      OUTPUT : none
*     RETURNS : 0 on success; 1 if a failure was encountered
*    CAUTIONS : none
*************************************************************/
DWORD WINAPI StateMachine1(LPVOID argp)
{
   char channame[20];
   int chdesc;
   int cnt;
   int hDevice[MAX_CHAN];
   int hEvent;
   long EventCode;
   int basechn = (int)argp;


   for (cnt = basechn; cnt < basechn + (MAX_CHAN/2); cnt++) {
      /* build name of voice channel */
      sprintf(channame, "dxxxB%dC%d", (cnt / 4) + 1,
                                      (cnt % 4) + 1);

      /* open voice channel */
      if ((chdesc = dx_open(channame, 0)) == -1) {
         printf("%s - FAILED: dx_open(): system error = %d\n",
                channame, dx_fileerrno());
         return(1);
      }
      hDevice[cnt] = chdesc;
      printf("%s - Voice channel opened\n", ATDV_NAMEP(chdesc));

      /* kick off the state machine by going offhook asynchronously */
      if (dx_sethook(chdesc, DX_OFFHOOK, EV_ASYNC) == -1) {
         printf("%s - FAILED: dx_sethook(DX_OFFHOOK): %s (error #%d)\n",
                ATDV_NAMEP(chdesc), ATDV_ERRMSGP(chdesc), ATDV_LASTERR(chdesc));
         return(1);
      }
      printf("%s - Voice channel off-hook initialized\n",ATDV_NAMEP(chdesc));
   }

   /* loop until Keyboard input is received */
   while (!Kbhit_flag) {
      /*
       * wait for event on the specific list of handles
       */
      sr_waitevtEx(&hDevice[basechn], MAX_CHAN/2, -1, &hEvent);

      /*
       * gather data about the event
       */
      chdesc = sr_getevtdev(hEvent);
      EventCode = sr_getevttype(hEvent);

      switch(EventCode) {
      case TDX_SETHOOK:
         if (ATDX_HOOKST(chdesc) == DX_OFFHOOK) {
            printf("%s - Voice channel off-hook\n",ATDV_NAMEP(chdesc));

            /* we went off hook so start dialing */
            if (dx_dial(chdesc, "12025551212", NULL, EV_ASYNC) == -1) {
               printf("%s - FAILED: dx_dial(): %s (error #%d)\n",
                  ATDV_NAMEP(chdesc), ATDV_ERRMSGP(chdesc), ATDV_LASTERR(chdesc));
               return(1);
```

```
            }
            printf("%s - Voice channel dialing initialized\n",ATDV_NAMEP(chdesc));
         } else {
            /* we went on hook so go off hook again */
            printf("%s - Voice channel on-hook\n",ATDV_NAMEP(chdesc));

            /* set the voice channel off-hook */
            if (dx_sethook(chdesc, DX_OFFHOOK, EV_ASYNC) == -1) {
               printf("%s - FAILED: dx_sethook(DX_OFFHOOK): %s (error #%d)\n",
                  ATDV_NAMEP(chdesc), ATDV_ERRMSGP(chdesc), ATDV_LASTERR(chdesc));
               return(1);
            }
            printf("%s - Voice channel off-hook initialized\n",ATDV_NAMEP(chdesc));
         }
         break;

      case TDX_DIAL:
         printf("%s - Voice channel Done dialing\n",ATDV_NAMEP(chdesc));

         /* done dialing so set the voice channel on-hook */
         if (dx_sethook(chdesc, DX_ONHOOK, EV_ASYNC) == -1) {
            printf("%s - FAILED: dx_sethook(DX_ONHOOK): %s (error #%d)\n",
               ATDV_NAMEP(chdesc), ATDV_ERRMSGP(chdesc), ATDV_LASTERR(chdesc));
            return(1);
         }
         printf("%s - Voice channel on-hook initialized\n",ATDV_NAMEP(chdesc));
         break;

      default:
         printf("Received unexpected event 0x%X on device %d\n", EventCode, chdesc);
         break;

      }

   }

   for (cnt = basechn; cnt < basechn + (MAX_CHAN/2); cnt++) {
      /* close the voice channel */
      chdesc = hDevice[cnt];
      printf("%s - Voice channel closing\n",ATDV_NAMEP(chdesc));
      if (dx_close(chdesc) == -1) {
         printf("%s - FAILED: dx_close(): %s (error #%d)\n",
                ATDV_NAMEP(chdesc), ATDV_ERRMSGP(chdesc), ATDV_LASTERR(chdesc));
         return(1);
      }
   }

   return(0);
}

/*************************************************************
*        NAME : DWORD WINAPI StateMachine2(LPVOID argp)
* DESCRIPTION : This tread runs the offhook-playtone-onhook state machine
*       INPUT : LPVOID argp - NULL pointer (not used)
*      OUTPUT : none
*     RETURNS : 0 on success; 1 if a failure was encountered
*    CAUTIONS : none
*************************************************************/
DWORD WINAPI StateMachine2(LPVOID argp)
{
   char channame[20];
   int chdesc;
```

```
int cnt;
int hDevice[MAX_CHAN];
int hEvent;
long EventCode;
int basechn = (int)argp;
TN_GEN ToneGeneration;

for (cnt = basechn; cnt < basechn + (MAX_CHAN/2); cnt++) {
   /* build name of voice channel */
   sprintf(channame, "dxxxB%dC%d", (cnt / 4) + 1,
                                   (cnt % 4) + 1);

   /* open voice channel */
   if ((chdesc = dx_open(channame, 0)) == -1) {
      printf("%s - FAILED: dx_open(): system error = %d\n",
              channame, dx_fileerrno());
      return(1);
   }
   hDevice[cnt] = chdesc;
   printf("%s - Voice channel opened\n", ATDV_NAMEP(chdesc));

   /* kick off the state machine by going offhook asynchronously */
   if (dx_sethook(chdesc, DX_OFFHOOK, EV_ASYNC) == -1) {
      printf("%s - FAILED: dx_sethook(DX_OFFHOOK): %s (error #%d)\n",
              ATDV_NAMEP(chdesc), ATDV_ERRMSGP(chdesc), ATDV_LASTERR(chdesc));
      return(1);
   }
   printf("%s - Voice channel off-hook initialized\n",ATDV_NAMEP(chdesc));
}

/* loop until Keyboard input is received */
while (!Kbhit_flag) {
   /*
    * wait for event on the specific list of handles
    */
   sr_waitevtEx(&hDevice[basechn], MAX_CHAN/2, -1, &hEvent);

   /*
    * gather data about the event
    */
   chdesc = sr_getevtdev(hEvent);
   EventCode = sr_getevttype(hEvent);

   switch(EventCode) {
   case TDX_SETHOOK:
      if (ATDX_HOOKST(chdesc) == DX_OFFHOOK) {
         printf("%s - Voice channel off-hook\n",ATDV_NAMEP(chdesc));

         /* we went off hook so build and play the tone */
         dx_bldtngen(&ToneGeneration, 340, 450, -10, -10, 300);
         if (dx_playtone(chdesc, &ToneGeneration, (DV_TPT *)NULL, EV_ASYNC) == -1) {
            printf("%s - FAILED: dx_playtone(): %s (error #%d)\n",
               ATDV_NAMEP(chdesc), ATDV_ERRMSGP(chdesc), ATDV_LASTERR(chdesc));
            return(1);
         }
         printf("%s - Voice channel play tone initialized\n",ATDV_NAMEP(chdesc));
      } else {
         /* we went on hook so go off hook again */
         printf("%s - Voice channel on-hook\n",ATDV_NAMEP(chdesc));

         /* set the voice channel off-hook */
         if (dx_sethook(chdesc, DX_OFFHOOK, EV_ASYNC) == -1) {
```

```
                printf("%s - FAILED: dx_sethook(DX_OFFHOOK): %s (error #%d)\n",
                   ATDV_NAMEP(chdesc), ATDV_ERRMSGP(chdesc), ATDV_LASTERR(chdesc));
                return(1);
             }
             printf("%s - Voice channel off-hook initialized\n",ATDV_NAMEP(chdesc));
          }
          break;

       case TDX_PLAYTONE:
          printf("%s - Voice channel Done playine tone\n",ATDV_NAMEP(chdesc));

          /* done playing a tone so set the voice channel on-hook */
          if (dx_sethook(chdesc, DX_ONHOOK, EV_ASYNC) == -1) {
             printf("%s - FAILED: dx_sethook(DX_ONHOOK): %s (error #%d)\n",
                ATDV_NAMEP(chdesc), ATDV_ERRMSGP(chdesc), ATDV_LASTERR(chdesc));
             return(1);
          }
          printf("%s - Voice channel on-hook initialized\n",ATDV_NAMEP(chdesc));
          break;

       default:
          printf("Received unexpected event 0x%X on device %d\n", EventCode, chdesc);
          break;

       }

    }

    for (cnt = basechn; cnt < basechn + (MAX_CHAN/2); cnt++) {
       /* close the voice channel */
       chdesc = hDevice[cnt];
       printf("%s - Voice channel closing\n",ATDV_NAMEP(chdesc));
       if (dx_close(chdesc) == -1) {
          printf("%s - FAILED: dx_close(): %s (error #%d)\n",
                  ATDV_NAMEP(chdesc), ATDV_ERRMSGP(chdesc), ATDV_LASTERR(chdesc));
          return(1);
       }
    }

    return(0);
}
```

4. Advanced SRL Programming Concepts for Dialogic Products

The following information is provided in this chapter:

- Guidelines for programming SRL event handlers (*Section 4.1*)
- Uses of Asynchronous with Windows callback programming (*Section 4.2*)
- Uses of Asynchronous with Win32 synchronization programming (*Section 4.3*)

4.1. SRL Event Handler Programming

An event handler is a user-defined function called by the SRL to handle a specified event that occurs on a specified device.

4.1.1. Setting Up Event Handlers

To manage many asynchronous events more simply, you can set up event handlers to act as application-level interrupt service routines. You can set up event handlers to be invoked for:

- A single event on any device.
- Any event on a specified device.
- Combinations of events on combinations of devices. Where overlap occurs, the SRL calls all applicable event handlers.

For details about how to use event handlers in asynchronous programming, see the following sections:

- *4.1.4. SRL Callback Thread Behavior*
- *5.3. Asynchronous with SRL Callback Model*

4.1.2. Event Handler Guidelines

The following guidelines apply to event handlers:

- You can enable more than one handler for any event. The SRL calls all specified handlers when a thread detects the event.
- You can enable or disable handlers from any thread.
- 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.

By default, on the first call to **sr_enbhdlr( )**, the SRL creates an internal thread, the SRL handler thread, that services event handlers. The SRL handler thread exists as long as one handler is still enabled. (See also the **sr_dishdlr( )** function.)

Handlers are called from the context of the SRL handler thread. Therefore, if the main thread is blocked in a non-Dialogic function and an asynchronous event, such as a hang-up, occurs on a device being controlled by the main thread, the handler is called immediately within the context of the SRL handler thread that has been created to service handlers.

Control this SRL handler thread creation through the SRL parameter SR_MODELTYPE, specified in the Asynchronous with SRL Callback model.

As explained in *Section 4.1.4. SRL Callback Thread Behavior*, the Asynchronous with SRL Callback model lets your application execute event handlers through the following threads:

- An SRL handler thread, which the SRL creates
- The application-handler thread, which you create

To execute event handlers through the SRL handler thread, enable the creation of the SRL handler thread by setting SR_MODELTYPE to SR_MTASYNC (default mode); to execute event handlers through the application-handler thread, disable creation of the SRL handler thread by setting SR_MODELTYPE to SR_STASYNC.

Another use of the SR_MODELTYPE parameter, and an alternate way for an application to service handlers in a multithreaded application, is to create one thread for each Dialogic device, then create a separate thread that calls **sr_waitevt( ).** The handlers are called within the context of this thread. If your

application uses this scenario, you should also set the SR_MODELTYPE parameter to SR_STASYNC to prevent creation of the SRL handler thread.

4.1.3. Hierarchy of Event Handlers

The SRL calls handlers according to a hierarchy based on device/event parings as follows:

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

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

long usr_hdlr(unsigned long evhandle);

4.1.4. SRL Callback Thread Behavior

The Asynchronous with SRL Callback model lets your application execute event handers through the following threads:

- An SRL handler thread, which SRL creates
- The application-handler thread, which you create

Using an SRL Handler Thread for SRL Callback

You can use an SRL handler thread to execute an event handler. Enable an event handler by calling the **sr_enbhdlr( )** function from within any application thread. You can set up separate event handlers for separate events for separate devices.

The first call to the **sr_enbhdlr( )** function automatically creates the SRL handler thread that services the event handler. You do not need to call the **sr_waitevt( )** function from anywhere within the application; the **sr_enbhdlr( )** thread already calls the **sr_waitevt( )** function to get events. Each call to the **sr_enbhdlr( )** function allows the Dialogic events to be serviced when Windows schedules the SRL handler thread for execution.

Call each event handler from the context of the SRL handler thread. The event handler must not call **sr_waitevt( )** or any synchronous Dialogic function.

The state machine is driven by the event handlers. If the event handler returns a 1, the event is kept. The next general handler in the hierarchy is notified.

Note: You can use the SRL handler thread in some UNIX applications that require porting.

Using an Application-Handler Thread

To create your own application thread, with which you can distribute your workload and gain more control over program structure, you can use the application-handler thread to make calls to the **sr_waitevt( )** function and execute event handlers. To avoid the creation of the SRL handler thread, you must set SR_MODELTYPE to SR_STASYNC. The thread must not call any synchronous functions.

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

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

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

4.2. Asynchronous with Windows Callback Programming

Asynchronous with Windows Callback programming allows an asynchronous application to receive SRL event notification through a standard Windows eventing technique. In Asynchronous with Windows Callback programming, the application informs the SRL to post a user-specified message to a user-specified window when an event occurs on a Dialogic device. When the application receives the user-specified message, it calls standard Dialogic event-retrieval functions to process the event.

Asynchronous with Windows Callback programming:

- Allows tighter integration with Windows GUI programming techniques.
- Uses system resources more efficiently than does Synchronous programming.
- Provides a single point of processing for all messages and events (Dialogic and Windows).

All of the asynchronous programming models let you program complex applications easily, and help you achieve a high level of resource management in your application by combining multiple voice channels in a single thread. This streamlined code reduces the system overhead required for inter-thread communication, and simplifies the coordination of events from many devices.

4.3. Asynchronous with Win32 Synchronization Programming

Asynchronous with Win32 Synchronization programming allows an asynchronous application to receive SRL event notification through standard Win32 synchronization mechanisms. The two mechanisms supported are Reset Events and I/O Completion Ports. In Asynchronous with Win32 Synchronization programming, the application informs the SRL to signal a user-specified wait point when an event occurs on a Dialogic device. When the application receives notification, it calls standard Dialogic event-retrieval functions to process the event. Asynchronous with Win32 Synchronization programming:

- Allows tighter integration with other devices that use Win32 event synchronization. These include, but are not limited to, Dialogic DM3 devices and the Windows Sockets library.

- Uses system resources more efficiently.
- Provides a single point of processing for all events (Dialogic and non Dialogic).

Asynchronous programming lets you program complex applications easily, and help you achieve a high level of resource management in your application by combining multiple voice channels in a single thread. This streamlined code reduces the system overhead required for inter-thread communication and simplifies the coordination of events from many devices.

5. Advanced SRL Programming Models

This chapter describes the following advanced SRL programming models:

- Synchronous with SRL Callback
- Asynchronous with SRL Callback
- Asynchronous with Windows Callback
- Asynchronous with Win32 Synchronization

5.1. Selecting Advanced SRL Programming Models

You can use advanced programming models in applications for which the basic programming models might not apply. Select an advanced programming model according to the criteria shown in *Table 2*. The sections following this table describe these advanced programming models.

Table 2. Guidelines for Selecting an Advanced Programming Model

Application Requirements	Recommended Advanced Programming Model	Threading and Event Handling Considerations
Few devices Needs to service unsolicited Dialogic events	Synchronous with SRL Callback (*Section 5.2*)	Create a separate thread to execute processing for each Dialogic device, typically a voice channel. Call **sr_enbhdlr( )** to enable callback functions that service unsoicited Dialogic events.

Application Requirements	**Recommended Advanced Programming Model**	**Threading and Event Handling Considerations**
Many devices Multiple tasks Needs user-defined event handlers	Asynchronous with SRL Callback (*Section 5.2*)	For details about how this model dispatches events to your event handlers, see *Section 4.1.4. SRL Callback Thread Behavior*.
Many devices Multiple tasks Needs the tightest possible integration with the Windows messaging scheme	Asynchronous with Windows Callback (*Section 5.4*)	Call **sr_NotifyEvent( )** to enable event notification to a user-specified window. When the window receives event notification, call **sr_waitevt(**0**)** to retrieve the Dialogic event.
Many devices Multiple tasks Needs the tightest possible integration with other Win32 devices	Asynchronous with Win32 Synchronization (*Section 5.5*)	At application intialization: • Use a Win32 function to create the Reset event or I/O Ccompletion Port. • Set up the **SRLWIN32INFO** structure. • Call the **sr_setparm( )** function. Use a Win32 function to wait for event notification. At notification, call the **sr_waitevt( )** function to retrieve the event from the Dialogic event queue.

5.2. Synchronous with SRL Callback Model

The Synchronous with SRL Callback Model combines some of the advantages of both synchronous and asynchronous programming.

Choose this model for an application that needs to execute processing for synchronous Dialogic devices, and that needs to be notified of some asynchronous Dialogic events.

Call the **sr_enbhdlr( )** function to enable a callback function that services some asynchronous Dialogic events. As an alternative, you can create your own thread which calls the **sr_waitevt( )** function to receive asynchronous Dialogic events.

Synchronous with SRL Callback Model Advantages

- The Synchronous with SRL Callback model executes processing for synchronous Dialogic devices, and receives notification of some unsolicited asynchronous Dialogic events.
- On the first call to **sr_enbhdlr( )**, the SRL automatically creates an SRL handler thread that services the event handlers.

Synchronous with SRL Callback Model Disadvantages

- Because the main thread creates a separate thread for each synchronous Dialogic device, this model requires a high level of system resources; thus, the synchronous model provides limited scalability for growing systems.
- You might need to set up a way for the event handler to communicate events to another thread. For example, the Dialogic event handler might need to stop a mutitasking function that is active in another thread.

Synchronous with SRL Callback Model Programming Notes

- Enable the event handler by calling **sr_enbhdlr( )** from within any application thread. You can set up separate event handlers for various devices and event types. The first call to **sr_enbhdlr( )** creates the SRL handler thread. You do not need to call **sr_waitevt( )** from anywhere within the application because the SRL handler thread already calls **sr_waitevt( )** to get events.

- The event handler must not call **sr_waitevt( )** or any synchronous Dialogic function.
    - For example, you can use this model to wait for inbound calls synchronously, then service those calls through telephony functions, such as play and record. You could use the event handlers to receive notifiation of unsolicited hangup events.

Synchronous with SRL Callback Model Example

An example of the Synchronous with SRL Callback model is shown below:

```
/*****************************************************************************
 *
 * This Windows Callback model sample application was designed to work with
 * D/41ESC, VFX/40ESC, LSI/81SC, LSI/161SC and D/160SC-LS boards only.
 * It was compiled using MS-VC++.
 * It cycles through 4 channels going offhook, dialing a digit string,
 * going onhook. This is repeated until the user stops the processing from the
 * Test menu in the main Window
 * The test can be started by choosing the Go option of the Test menu in the
 * program window
 *
*****************************************************************************/
#define STRICT
#include <windows.h>
#include <windowsx.h>
#include <afxres.h>
#include <process.h>
#include <dxxxlib.h>
#include <srllib.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include "resource.h"
#include <sctools.h>
// Defines
#define MAXCHAN         4 // maximum number of voice channels in system
#define WM_SRNOTIFYEVENT        WM_USER + 100
#define ROWHEIGHT       20
// Modified version of the normal HANDLE_MSG macro in windows.h
#define HANDLE_DLGMSG(hwnd, message, fn)     \
                        case (message):                          \
                        return(SetDlgMsgResult(hwnd, uMsg,     \
                        HANDLE_##message((hwnd), (wParam),  \
                           (lParam), (fn))))
// This may be expanded to contain other information such as state typedef struct
dx_info {
                        int chdev;
                        int iter;
} DX_INFO;
// Globals
```

```
DX_INFO dxinfo[MAXCHAN+1];
int Kbhit_flag = 0;
char tmpbuf[128];
HANDLE hInst;
char gRowVal[MAXCHAN+1][80];
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
void General_OnCommand(HWND , int , HWND , UINT );  // Window's WM_COMMAND handler int
dlgc_OnCommand(HWND );                              // WM_SRNOTIFYEVENT handler
int DialogicSysInit(HWND );
void DialogicClose(HWND);
int get_ts(int );
void disp_status(HWND, int , char *);
/*****************************************************************************
 *                                  NAME : WinMain()
 * DESCRIPTION : Windows application entry point
 *          CAUTIONS : none.
*****************************************************************************/
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hinstPrev, LPSTR lpszCmdLine, int
nCmdShow) {
                        HWND hWnd;
                        WNDCLASS wc;
            MSG msg;
                        hInst = hInstance;
                        if (!hinstPrev)
            {
                        // Fill in window class structure with parameters that
                        // describe the main window.
                        wc.style    = CS_HREDRAW | CS_VREDRAW;// Class style(s).
                        wc.lpfnWndProc          = (WNDPROC)WndProc;       // Window
Procedure
                        wc.cbClsExtra           = 0;                      // No per-
class extra data.
                        wc.cbWndExtra           = 0;                      // No per-
window extra data.
                        wc.hInstance            = hInstance;              // Owner of
this class
                        wc.hIcon    = LoadIcon (hInstance, MAKEINTRESOURCE(IDI_ICON1));
// Icon name from .RC
                        wc.hCursor  = LoadCursor(NULL, IDC_ARROW);// Cursor
            wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);// Default color
            wc.lpszMenuName  = MAKEINTRESOURCE(IDR_MENU1);
            wc.lpszClassName = "WinCallBack";    // Name to register
                        // Register the window class and return success/failure code.
                        if (!RegisterClass(&wc))
                                    return (FALSE);     // Exits if unable to register
             }
                         hWnd = CreateWindowEx(0L,"WinCallBack", "Windows Callback
Demo", WS_OVERLAPPEDWINDOW,
                                                                                CW_USEDEFAULT, 0,
CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
            // If window could not be created, return "failure"
            if (!hWnd)
                        return (FALSE);
                        sr_NotifyEvent(hWnd, WM_SRNOTIFYEVENT, SR_NOTIFY_ON);
                        ShowWindow(hWnd, SW_SHOW);  // Show the window
            UpdateWindow(hWnd);         // Sends WM_PAINT message

// Get and dispatch messages until a WM_QUIT message is received. while
(GetMessage(&msg,     NULL, 0,0))
{
TranslateMessage(&msg);// Translates virtual key code DispatchMessage(&msg); //
Dispatches message to window
```

```
            }
                        return (0);
}
/*****************************************************************************
 *                                  NAME : WndProc()
 * DESCRIPTION : Windows Procedure
 *    CAUTIONS : none.
*****************************************************************************/
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
                        HDC  hDC;
                        PAINTSTRUCT ps;
                        RECT rect;
                        int numchan;
                        switch (uMsg) {
            // Handle the WM_COMMAND messages HANDLE_MSG(hWnd, WM_COMMAND,
General_OnCommand);
                        case WM_SRNOTIFYEVENT:
                                    if (dlgc_OnCommand( hWnd))  {// Dialogic event
DialogicClose(hWnd);
DestroyWindow(hWnd); // if dlgc_OnCommand() returns FALSE }
                        break;
            case WM_CREATE:
                        break;
            case WM_PAINT:
                        // get the actual window rectangle
                        GetClientRect(hWnd, &rect);
                        hDC = BeginPaint(hWnd, &ps);

                        // display name of application
                        rect.top = ROWHEIGHT;
                                    sprintf(tmpbuf,"Windows Callback Demo");
                        DrawText(hDC,tmpbuf , -1, &rect, DT_SINGLELINE|DT_CENTER);

                        // display status of channel
                       for (numchan=1;numchan<=MAXCHAN;numchan++) {
                                    rect.top = (numchan+2) * ROWHEIGHT;
                                    DrawText(hDC, gRowVal[numchan], -1, &rect,
DT_SINGLELINE);
                        }

                        EndPaint(hWnd, &ps);
                        break;
case WM_CLOSE:
                        DestroyWindow(hWnd);
                        break;
            case WM_DESTROY:
                        PostQuitMessage(0);             // Allow GetMessage() to return
FALSE
                        break;
                        default:
                                    return (DefWindowProc(hWnd, uMsg, wParam, lParam));
                                    } // switch (uMsg)
}
/*****************************************************************************
 * NAME : General_OnCommand()
 * DESCRIPTION : Message Handler for WM_COMMAND
 * CAUTIONS : none.
 *****************************************************************************/
void General_OnCommand(HWND hWnd, int id, HWND hwndCtl, UINT codeNotify)
{
              switch (id) {
```

```
                        case ID_TEST_EXIT:
DestroyWindow(hWnd); // post WM_DESTROY message for WndProc to exit app break;
             case ID_TEST_GO:                    // create threads here and gray the
"start" menu and
                                                 // launch the call control threads
                        if (DialogicSysInit(hWnd)) {
// initialize Dialogic devices, if error initializing then show error message and exit
MessageBox(hWnd, "Error initializing",
"ERROR", MB_OK | MB_ICONSTOP| MB_APPLMODAL); break;
              }
// // ungrey the "stop" menu
EnableMenuItem(GetMenu(hWnd), ID_TEST_STOP, MF_ENABLED );
EnableMenuItem(GetMenu(hWnd), ID_TEST_GO, MF_DISABLED | MF_GRAYED);
break;
case ID_TEST_STOP:                                  // "terminate" the call control
threads and ungray menu items // terminate the call control
threads
             DialogicClose(hWnd);
// disable the Action/Stop menu item
EnableMenuItem(GetMenu(hWnd), ID_TEST_STOP, MF_DISABLED | MF_GRAYED);
EnableMenuItem(GetMenu(hWnd),
ID_TEST_GO, MF_ENABLED );
              break;
              default:
                        return;
              } // switch (id)
}
/*****************************************************************************
 *                                  NAME : dlgc_OnCommand()
 * DESCRIPTION : Message Handler for WM_SRNOTIFYEVENT
 *    CAUTIONS : none.
*****************************************************************************/
int dlgc_OnCommand(HWND hWnd)
{
             int rc = 0;
int chdev;
int event;
DX_CST *cstp;
static iter=0;
int channum;

if (sr_waitevt(0) == -1) {
sprintf(tmpbuf, "sr_waitevt() ERROR %s", ATDV_ERRMSGP( SRL_DEVICE ));
MessageBox(hWnd,tmpbuf, "ERROR" , MB_OK|MB_APPLMODAL);
              return (1);
}
chdev = sr_getevtdev();
event = sr_getevttype();
                        /*
* Switch according to the event received. */
                        switch ( event ) {
                        case TDX_SETHOOK:
cstp = (DX_CST *)sr_getevtdatap(); switch( cstp->cst_event) { case DX_ONHOOK:
                                  /* Go offhook next */
                                                 if (dx_sethook(chdev, DX_OFFHOOK,
EV_ASYNC) == -1) {
sprintf(tmpbuf,"FAILED: dx_sethook(%s, DX_OFFHOOK): %s (error #%d)", ATDV_NAMEP(chdev),
ATDV_ERRMSGP(chdev),
                                                                    ATDV_LASTERR(chdev));
                                                                    MessageBox(hWnd, tmpbuf,
"ERROR",MB_OK|MB_APPLMODAL); return(1);
                                  }
```

```
                                    break;
                          case DX_OFFHOOK:
                                    /* dial next */
if (dx_dial(chdev, "12025551212", NULL, EV_ASYNC) == -1) { sprintf(tmpbuf,"FAILED:
dx_dial(%s): %s (error #%d) ",         ATDV_NAMEP(chdev),
                         ATDV_ERRMSGP(chdev),
                         ATDV_LASTERR(chdev));
MessageBox(hWnd, tmpbuf, "ERROR",MB_OK|MB_APPLMODAL); return(1);
                                    }
                                    break;
                          }
            break;
            case TDX_DIAL:
                         /* Next go onhook */
            if (dx_sethook(chdev, DX_ONHOOK, EV_ASYNC) == -1) {
sprintf(tmpbuf,"FAILED: dx_sethook(%s, DX_ONHOOK): %s (error #%d) ", ATDV_NAMEP(chdev),
ATDV_ERRMSGP(chdev),

                         ATDV_LASTERR(chdev));
                                              MessageBox(hWnd, tmpbuf,
"ERROR",MB_OK|MB_APPLMODAL);
                                    return(1);
                          }
             }
             // retrieve channel number using the USERCONTEXT feature of the SRL
if (sr_getparm(chdev, SR_USERCONTEXT, (void *)&channum) == -1) { sprintf(tmpbuf,"FAILED:
sr_getparm(%s): %s (error #%d) ",
ATDV_NAMEP(dxinfo[channum].chdev),
                                    ATDV_ERRMSGP(dxinfo[channum].chdev),
                                    ATDV_LASTERR(dxinfo[channum].chdev));
                                    MessageBox(hWnd, tmpbuf,
"ERROR",MB_OK|MB_APPLMODAL);
                         return(1);
            }
sprintf(tmpbuf, "Iteration %d Completed", ++dxinfo[channum].iter); disp_status(hWnd,
channum, tmpbuf );
            return (0);
}
/*****************************************************************************
 *                                                                        NAME :
DialogicSysInit()
 * DESCRIPTION : Initialization of Dialogic devices
 *                                    CAUTIONS : none.
*****************************************************************************
/ int DialogicSysInit(HWND hWnd)
{
            int numchan;
            char channame[20];
            /* Initial processing for MAXCHANS */
            for (numchan=1;numchan<=MAXCHAN;numchan++) {
                         /* build name of voice channel */
                         sprintf(channame, "dxxxB%dC%d", ((numchan-1) / 4) + 1,

                                    ((numchan -1)% 4) + 1);
                         /* open voice channel */
            if ((dxinfo[numchan].chdev = dx_open(channame, 0)) == -1)
{sprintf(tmpbuf,"FAILED: dx_open(%s): system error = %d ", channame,

dx_fileerrno());
                          MessageBox(hWnd, tmpbuf, "ERROR",MB_OK|MB_APPLMODAL);
            return(1);
}
```

```
            /* set user specific information in the device, in this
            ** case the channel number

*/
            if (sr_setparm(dxinfo[numchan].chdev, SR_USERCONTEXT, (void *)&numchan) == -
1) { sprintf(tmpbuf,"FAILED: sr_setparm(%s): %s (error    #%d) ",
ATDV_NAMEP(dxinfo[numchan].chdev), ATDV_ERRMSGP(dxinfo[numchan].chdev),
ATDV_LASTERR(dxinfo[numchan].chdev));

                                    MessageBox(hWnd, tmpbuf,
"ERROR",MB_OK|MB_APPLMODAL);
                        return(1);
                        }
/* Start the application by putting the channel in onhook state */
if (dx_sethook(dxinfo[numchan].chdev,DX_ONHOOK,EV_ASYNC) == -1) {
sprintf(tmpbuf,"FAILED: dx_sethook(%s): %s (error #%d) ",
ATDV_NAMEP(dxinfo[numchan].chdev), ATDV_ERRMSGP(dxinfo[numchan].chdev),
ATDV_LASTERR(dxinfo[numchan].chdev));

                                    MessageBox(hWnd, tmpbuf,
"ERROR",MB_OK|MB_APPLMODAL); return(1);
                                    }
                        }
              return(0);
}
/*****************************************************************************
 * NAME : DialogicClose()
 * DESCRIPTION : Tier down of Dialogic devices
 * CAUTIONS : none.
*****************************************************************************/
void DialogicClose(HWND hWnd)
{
             int numchan;

             /* Close all voice devices before exiting */
             for (numchan=1;numchan<=MAXCHAN;numchan++) {
// attempt to stop the channel dx_stopch(dxinfo[numchan].chdev,EV_SYNC);
dx_close(dxinfo[numchan].chdev);
             }
}

/*****************************************************************************
* NAME: disp_status(hwnd, chnum, stringp)
* INPUTS: chno - channel number (1 - 12)
* stringp - pointer to string to display
* DESCRIPTION: display the current activity on the channel in window 2
* the string pointed to by stringp) using chno as a Y offset
*****************************************************************************/
void disp_status(HWND hWnd, int channum, char *stringp)
{
            RECT rect;

// get the entire window rectangle and modify it
GetClientRect(hWnd, &rect);
rect.top = (channum+2) * ROWHEIGHT;
             rect.bottom = (channum+3) * ROWHEIGHT;

// buffer the message
             sprintf(gRowVal[channum], "Channel %d: %s", channum, stringp);

InvalidateRect(hWnd, &rect, TRUE);
UpdateWindow(hWnd);
```

```
}
```

5.3. Asynchronous with SRL Callback Model

The Asynchronous with SRL Callback model lets your application execute event handers through either:

- An SRL handler thread
- An application-handler thread

For details, see *Section 4.1.4. SRL Callback Thread Behavior* and *Section*

Choose the Asynchronous with SRL Callback model for an application that:

- Requires a state machine.
- Can benefit from the use of event handlers.

Asynchronous with SRL Callback Model Advantages

- Because the Asynchronous with SRL Callback model uses one thread for all Dialogic devices, it requires a lower level of system resources than does the Synchronous model; thus, the Asynchronous with Win32 Synchronization model allows for greater scalability in growing systems.
- Even if the application's non-Dialogic threads block on non-Dialogic functions, the event handlers can still handle Dialogic events. This model ensures that Dialogic events can be serviced when an event occurs and when Windows schedules the thread for execution.

Asynchronous with SRL Callback Model Disadvantages

- You might need to set up a way for the event handler to communicate events to another thread.

Asynchronous with SRL Callback Model Programming Notes

See *Section 4.1.4. SRL Callback Thread Behavior*.

Asynchronous with SRL Callback Model Example

An example of the Aynchronous with SRL Callback model, using the main thread, is shown below:

```
/********* Aynchronous with SRL Callback Model - Using the Main Thread **********/


/*
 * Compiled using Visual C++ 5.0
 */

/* C includes */
#include <windows.h>
#include <stdio.h>
#include <StdLib.H>
#include <process.h>
#include <conio.h>
#include <ctype.h>
#include <String.H>

/* Dialogic includes */
#include <srllib.h>
#include <dxxxlib.h>

/* Defines */
#define MAXCHAN     4       /* maximun number of voice channels in system */
#define FOREVER     1
#define DIALSTRING  "01234"

/* Globals */
int vxh[MAXCHAN];
int errflag = FALSE;

/* Prototypes */
int main();
long fallback_hdlr(unsigned long parm);
int voxinit(void);
int sysexit(int exitcode);
int process_events(void);



/*****************************************************************
*        NAME : int main(void)
* DESCRIPTION : The Entry Point into the application.
*       INPUT : none
*      OUTPUT : none
*     RETURNS : 0 on success; 1 if a failure was encountered
*    CAUTIONS : none
*****************************************************************/

int main(void)
{

   /* Show application's title */
   printf("Asynchronous with Main-Thread Callback Model\n");

   /* Start Dialogic Devices */
   if (voxinit() == -1) {
```

```
        sysexit(-1);
    }

    /* Process events , monitor keyboard input and other activities */
    if (process_events() == -1) {
        sysexit(-1);
    }

    sysexit(0);

    return(0);
}


/*****************************************************************
*        NAME : int voxinit(void)
* DESCRIPTION : Initializes Dialogic Devices.
*       INPUT : none
*      OUTPUT : none
*     RETURNS : 0 on success; -1 if a failure was encountered
*    CAUTIONS : none
*****************************************************************/
int voxinit(void)
{

    int index;
    int mode;
    char devname[32];

    /** Set to SR_STASYNC so another thread is not created by the SRL to
     ** monitor events to pass to the handler.  We will use this thread
           ** to monitor events; creating another thread internally is not
           ** necessary.
           **/

    mode = SR_STASYNC;

    if (sr_setparm(SRL_DEVICE, SR_MODELTYPE, &mode ) == -1 ) {
       printf("ERROR:Unable to set SRL modeltype to SR_STASYNC \n" );
       return(-1);
    }


    /*
     * Set-up the fall-back handler
     */
    if (sr_enbhdlr((long)EV_ANYDEV, (unsigned long)EV_ANYEVT, fallback_hdlr) == -1 ) {
       printf("ERROR:Unable to set-up the fall back handler \n");
       return(-1);
    }

    /* Open the voice chans now */
    for (index = 0; index < MAXCHAN; index++) {
        sprintf(devname, "dxxxB1C%d", (index+1));
        if ((vxh[index] = dx_open(devname, 0)) == -1) {
         /* Perform system error processing */
            return(-1);
        }
    }

    /* Issue the dial without call progres on all the voice chans */
```

```
    for (index = 0; index < MAXCHAN; index++) {
        if (dx_dial(vxh[index], DIALSTRING, NULL, EV_ASYNC) == -1) {
            printf("ERROR: dx_dial(%s) failed, 0x%X(%s)\n",
                    ATDV_NAMEP(vxh[index]), ATDV_LASTERR(vxh[index]),
                    ATDV_ERRMSGP(vxh[index]));
            return(-1);
        }
    }
}


/*****************************************************************
*        NAME : int process_events(void)
* DESCRIPTION : The processing loop.
*       INPUT : none
*      OUTPUT : none
*     RETURNS : 0 on success; -1 if a failure was encountered
*    CAUTIONS : none
*****************************************************************/
int process_events(void)
{

    while (FOREVER) {
       if (kbhit() != (int)0) {
          return(-1);
       }

       /* Wait for Dialogic events 1 second */
       sr_waitevt(1000);

       if (errflag == TRUE) {
           return(-1);
       }

       /* Do other processing here */
       printf("Press Any Key to Exit \n");
    }
    return(0);

}


/*****************************************************************
*        NAME : long fallback_hdlr(unsigned long parm)
* DESCRIPTION : The fallback handler for all Dialogic events
*       INPUT : parm
*      OUTPUT : return value
*     RETURNS :
*    CAUTIONS : none
*****************************************************************/
long fallback_hdlr(unsigned long parm)
{
    int index, devh, evttype;

    /* Find out the event received */
    devh = sr_getevtdev();
    evttype = sr_getevttype();

    for (index=0; index<MAXCHAN; index++) {
        if (devh == vxh[index]) {
            break;
        }
```

```
    }

    switch(evttype) {
       case TDX_DIAL :
          printf("%s : Dialing again.\n", ATDV_NAMEP(vxh[index]));
          if (dx_dial(vxh[index], DIALSTRING, NULL, EV_ASYNC) == -1) {
              printf("ERROR: dx_dial(%s) failed, 0x%X(%s)\n",
                     ATDV_NAMEP(vxh[index]), ATDV_LASTERR(vxh[index]),
                     ATDV_ERRMSGP(vxh[index]));
              errflag = TRUE;
          }
          break;

       case TDX_ERROR :
          printf("%s : TDX_ERROR!\n", ATDV_NAMEP(vxh[index]));
          errflag = TRUE;
          break;

       default :
          printf("%s : unknown event(0x%X)\n", ATDV_NAMEP(vxh[index]), evttype);
          errflag = TRUE;
          break;
    }


    /* Remove the events from the SRL queue */
    return(1);

}



/*************************************************************
*        NAME : void sysexit(exitcode)
* DESCRIPTION : Closes the devices and exits the application.
*       INPUT : none
*      OUTPUT : none
*     RETURNS : exitcode
*    CAUTIONS : Exit of the application!
*************************************************************/
int sysexit(int exitcode)
{
    int index;

    /* Close the voice chans now */
    for (index = 0; index < MAXCHAN; index++) {
        if (dx_close(vxh[index]) == -1) {
            printf("ERROR: dx_close(%s) failed, 0x%X(%s)\n",
                   ATDV_NAMEP(vxh[index]), ATDV_LASTERR(vxh[index]),
                   ATDV_ERRMSGP(vxh[index]));
        }
    }

    exit(exitcode);
    return(exitcode);
}
```

5.4. Asynchronous with Windows Callback Model

This model uses the **sr_NotifyEvt( )** function to send event notification (a user-specified message) to a user-specified window. The main application loop follows the standard Windows message handling scheme. When a Dialogic completion event occurs:

1. A user-specified message is sent to a user-specified window.
2. The **sr_waitevt(**0**)** function is called with a zero time-out.
3. Event data retrieval functions **sr_gettevtdev( )** and **sr_gettevttype( )** retrieve information about the event.
4. The next action is initiated asynchronously.

Choose the Asynchronous with Windows Callback model for an application that needs the tightest possible integration with the Windows messaging scheme.

Asynchronous with Windows Callback Model Advantages

- This model (Asynchronous with Windows Callback) requires a low level of system resources, because it does not require a separate thread for each Dialogic device. You can run the entire Dialogic portion of the application on a single thread.
- Provides tight integration within the Windows messaging scheme.
- Services all events for all devices within the Windows messaging loop.

Asynchronous with Windows Callback Model Disadvantages

Since this message loop is handling all the processing, you must be more aware of the efficiency of your Windows procedure and not call many blocking functions that may take a long time to complete.

Asynchronous with Windows Callback Model Programming Notes

- The Windows message callback function must be re-entrant.
- Enable a Windows message callback function as follows:

1. Enable Windows message callback from any thread in the application by calling **sr_NotifyEvent( )**
2. When the application receives event notification through Windows message callback, the application must immediately call **sr_waitevt()**, with a zero timeout, to retrieve the event from the Dialogic event queue.
3. To retrieve information about the event, call:
   - **sr_gettevtdev( )** to get the event's device handle.
   - **sr_gettevttype( )** to get the event type.

- If any thread has a callback window that can block on a non-Dialogic function, the thread might not be able to service Dialogic events quickly enough. In this case, you can create a separate thread that creates a hidden window. The thread must own the message queue. The thread calls **sr_NotifyEvent()**, passing the handle to the hidden window.

Asynchronous with Windows Callback Model Example

An example of the Asynchronous with Windows Callback model is shown below:

```
/*****************************************************************************
 *
 * This Windows Callback model sample application was designed to work with
 * D/41ESC, VFX/40ESC, LSI/81SC, LSI/161SC and D/160SC-LS boards only.
 * It was compiled using MS-VC++.
 * It cycles through 4 channels going offhook, dialing a digit string,
 * going onhook. This is repeated until the user stops the processing from the
 * Test menu in the main Window
 * The test can be started by choosing the Go option of the Test menu in the
 * program window
 *
*****************************************************************************/
#define STRICT
#include <windows.h>
#include <windowsx.h>
#include <afxres.h>
#include <process.h>
#include <dxxxlib.h>
#include <srllib.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include "resource.h"
#include <sctools.h>

// Defines
#define MAXCHAN         4         // maximum number of voice channels in system
#define WM_SRNOTIFYEVENT     WM_USER + 100
#define ROWHEIGHT                20

// Modified version of the normal HANDLE_MSG macro in windowsx.h
```

```
#define HANDLE_DLGMSG(hwnd, message, fn)                \
        case (message):                                 \
             return(SetDlgMsgResult(hwnd, uMsg,           \
                         HANDLE_##message((hwnd), (wParam), \
                         (lParam), (fn))))
// This may be expanded to contain other information such as state typedef
struct dx_info {
        int chdev;
        int iter;
} DX_INFO;

// Globals
DX_INFO dxinfo[MAXCHAN+1];
int Kbhit_flag = 0;
char tmpbuf[128];
HANDLE hInst;
char gRowVal[MAXCHAN+1][80];


LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
void General_OnCommand(HWND , int , HWND , UINT );      // Window's WM_COMMAND handler
int dlgc_OnCommand(HWND );                              // WM_SRNOTIFYEVENT handler
int DialogicSysInit(HWND );
void DialogicClose(HWND);
int get_ts(int );
void disp_status(HWND, int , char *);

/*****************************************************************************
 *        NAME : WinMain()
 * DESCRIPTION : Windows application entry point
 *    CAUTIONS : none.
*****************************************************************************/
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hinstPrev, LPSTR lpszCmdLine, int
nCmdShow)
{
        HWND hWnd;
        WNDCLASS wc;
        MSG msg;

        hInst = hInstance;

        if (!hinstPrev)
        {
           // Fill in window class structure with parameters that
           // describe the main window.
           wc.style         = CS_HREDRAW | CS_VREDRAW;// Class style(s).
           wc.lpfnWndProc   = (WNDPROC)WndProc;       // Window Procedure
           wc.cbClsExtra    = 0;                      // No per-class extra data.
           wc.cbWndExtra    = 0;                      // No per-window extra data.
           wc.hInstance     = hInstance;              // Owner of this class
           wc.hIcon         = LoadIcon (hInstance, MAKEINTRESOURCE(IDI_ICON1)); // Icon
name from .RC
           wc.hCursor       = LoadCursor(NULL, IDC_ARROW);// Cursor
           wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);// Default color
           wc.lpszMenuName  = MAKEINTRESOURCE(IDR_MENU1);
           wc.lpszClassName = "WinCallBack";                       // Name to register
           // Register the window class and return success/failure code.
           if (!RegisterClass(&wc))
             return (FALSE);     // Exits if unable to register
        }

        hWnd = CreateWindowEx(0L,"WinCallBack", "Windows Callback Demo",
```

```
WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, 0, CW_USEDEFAULT,
                  0, NULL, NULL, hInstance, NULL);


        // If window could not be created, return "failure"
        if (!hWnd)
           return (FALSE);

        sr_NotifyEvent(hWnd, WM_SRNOTIFYEVENT, SR_NOTIFY_ON);

        ShowWindow(hWnd, SW_SHOW);  // Show the window
        UpdateWindow(hWnd);         // Sends WM_PAINT message


        // Get and dispatch messages until a WM_QUIT message is received.
        while (GetMessage(&msg, NULL, 0,0))
        {
            TranslateMessage(&msg);// Translates virtual key code
            DispatchMessage(&msg); // Dispatches message to window
        }

        return (0);
}


/*****************************************************************************
 *        NAME : WndProc()
 * DESCRIPTION : Windows Procedure
 *    CAUTIONS : none.
*****************************************************************************/
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
        HDC  hDC;
        PAINTSTRUCT ps;
        RECT rect;
        int numchan;


        switch (uMsg) {
           // Handle the WM_COMMAND messages
           HANDLE_MSG(hWnd, WM_COMMAND, General_OnCommand);

           case WM_SRNOTIFYEVENT:
                if (dlgc_OnCommand( hWnd ))  { // Dialogic event
                   DialogicClose(hWnd);
                   DestroyWindow(hWnd);       // if dlgc_OnCommand() returns 1
                }
                break;

           case WM_CREATE:
                break;

           case WM_PAINT:
                // get the actual window rectangle
                GetClientRect(hWnd, &rect);
                hDC = BeginPaint(hWnd, &ps);

                // display name of application
                rect.top = ROWHEIGHT;
                sprintf(tmpbuf,"Windows Callback Demo");
                DrawText(hDC,tmpbuf , -1, &rect, DT_SINGLELINE|DT_CENTER);
```

```
                // display status of channel
                for (numchan=1;numchan<=MAXCHAN;numchan++) {
                        rect.top = (numchan+2) * ROWHEIGHT;
                        DrawText(hDC, gRowVal[numchan], -1, &rect, DT_SINGLELINE);
                }

                EndPaint(hWnd, &ps);
                break;

           case WM_CLOSE:
                DestroyWindow(hWnd);
                break;

           case WM_DESTROY:
                PostQuitMessage(0);             // Allow GetMessage() to return FALSE
                break;

           default:
                return (DefWindowProc(hWnd, uMsg, wParam, lParam));
        } // switch (uMsg)
}

/*******************************************************************************
 *        NAME : General_OnCommand()
 * DESCRIPTION : Message Handler for WM_COMMAND
 *    CAUTIONS : none.
 *******************************************************************************/
void General_OnCommand(HWND hWnd, int id, HWND hwndCtl, UINT codeNotify)
{
        switch (id) {
                case ID_TEST_EXIT:
                   DestroyWindow(hWnd); // post WM_DESTROY message for WndProc to exit
app break;

                case ID_TEST_GO:     // create threads here and gray the "start" menu
                                                // and launch the call control
          threads
                   if (DialogicSysInit(hWnd)) {
                      // initialize Dialogic devices, if error initializing then show
error message and exit
                      MessageBox(hWnd, "Error initializing", "ERROR", MB_OK |
MB_ICONSTOP| MB_APPLMODAL); break;
                   }
                   // ungrey the "stop" menu
                   EnableMenuItem(GetMenu(hWnd), ID_TEST_STOP, MF_ENABLED );
                   EnableMenuItem(GetMenu(hWnd), ID_TEST_GO, MF_DISABLED | MF_GRAYED);
                   break;

                case ID_TEST_STOP:   // "terminate" the call control threads and ungray
menu items
                   DialogicClose(hWnd);
                   // disable the Action/Stop menu item
                   EnableMenuItem(GetMenu(hWnd), ID_TEST_STOP, MF_DISABLED | MF_GRAYED);
                   EnableMenuItem(GetMenu(hWnd), ID_TEST_GO, MF_ENABLED );
                   break;

                default:
                   return;

        } // switch (id)
}
```

```
/*****************************************************************************
 *        NAME : dlgc_OnCommand()
 * DESCRIPTION : Message Handler for WM_SRNOTIFYEVENT
 *    CAUTIONS : none.
 *****************************************************************************/
int dlgc_OnCommand(HWND hWnd)
{
        int rc = 0;
        int chdev;
        int event;
        DX_CST *cstp;
        static iter=0;
        int channum;


        if (sr_waitevt(0) == -1) {
           sprintf(tmpbuf, "sr_waitevt() ERROR %s", ATDV_ERRMSGP( SRL_DEVICE ));
           MessageBox(hWnd,tmpbuf, "ERROR" , MB_OK|MB_APPLMODAL);
           return (1);
        }

        chdev = sr_getevtdev();
        event = sr_getevttype();


        /* Switch according to the event received. */
        switch ( event ) {

           case TDX_SETHOOK:
              cstp = (DX_CST *)sr_getevtdatap();
              switch( cstp->cst_event) {
                 case DX_ONHOOK:
                    /* Go offhook next */
                    if (dx_sethook(chdev, DX_OFFHOOK, EV_ASYNC) == -1) {
                       sprintf(tmpbuf,"FAILED: dx_sethook(%s, DX_OFFHOOK): %s (error
#%d)", ATDV_NAMEP(chdev), ATDV_ERRMSGP(chdev), ATDV_LASTERR(chdev));
                        MessageBox(hWnd, tmpbuf, "ERROR",MB_OK|MB_APPLMODAL);
                        return(1);
                    }
                    break;

                 case DX_OFFHOOK:
                    /* dial next */
                    if (dx_dial(chdev, "12025551212", NULL, EV_ASYNC) == -1) {
                       sprintf(tmpbuf,"FAILED: dx_dial(%s): %s (error #%d)
",ATDV_NAMEP(chdev), ATDV_ERRMSGP(chdev), ATDV_LASTERR(chdev));
                      MessageBox(hWnd, tmpbuf, "ERROR",MB_OK|MB_APPLMODAL);
                      return(1);
                    }
                    break;
              }
              break;

              case TDX_DIAL:
                 /* Next go onhook */
                 if (dx_sethook(chdev, DX_ONHOOK, EV_ASYNC) == -1) {
                    sprintf(tmpbuf,"FAILED: dx_sethook(%s, DX_ONHOOK): %s (error #%d) ",
ATDV_NAMEP(chdev), ATDV_ERRMSGP(chdev), ATDV_LASTERR(chdev));
                    MessageBox(hWnd, tmpbuf, "ERROR",MB_OK|MB_APPLMODAL);
                    return(1);
              }
```

```
            break;

        }

        // retrieve channel number using the USERCONTEXT feature of the SRL
        if (sr_getparm(chdev, SR_USERCONTEXT, (void *)&channum) == -1) {
           sprintf(tmpbuf,"FAILED: sr_getparm(%s): %s (error #%d) ",
ATDV_NAMEP(dxinfo[channum].chdev), ATDV_ERRMSGP(dxinfo[channum].chdev),
ATDV_LASTERR(dxinfo[channum].chdev));
           MessageBox(hWnd, tmpbuf, "ERROR",MB_OK|MB_APPLMODAL);
           return(1);
        }

        sprintf(tmpbuf, "Iteration %d Completed", ++dxinfo[channum].iter);
        disp_status(hWnd, channum, tmpbuf );

         return (0);
}

/*****************************************************************************
 *        NAME : DialogicSysInit()
 * DESCRIPTION : Initialization of Dialogic devices
 *    CAUTIONS : none.
*****************************************************************************/
int DialogicSysInit(HWND hWnd)
{

        int numchan;
        char channame[20];

        /* Initial processing for MAXCHANS */
        for (numchan=1;numchan<=MAXCHAN;numchan++) {
           /* build name of voice channel */
           sprintf(channame, "dxxxB%dC%d", ((numchan-1) / 4) + 1,
((numchan -1)% 4) + 1);

           /* open voice channel */
           if ((dxinfo[numchan].chdev = dx_open(channame, 0)) == -1) {
              sprintf(tmpbuf,"FAILED: dx_open(%s): system error = %d ", channame,
dx_fileerrno());
              MessageBox(hWnd, tmpbuf, "ERROR",MB_OK|MB_APPLMODAL);
              return(1);
           }

           /* set user specific information in the device, in this
           ** case the channel number
           */
           if (sr_setparm(dxinfo[numchan].chdev, SR_USERCONTEXT, (void *)&numchan) ==
1) {
               sprintf(tmpbuf,"FAILED: sr_setparm(%s): %s (error #%d) ",
ATDV_NAMEP(dxinfo[numchan].chdev), ATDV_ERRMSGP(dxinfo[numchan].chdev),
ATDV_LASTERR(dxinfo[numchan].chdev));
               MessageBox(hWnd, tmpbuf, "ERROR",MB_OK|MB_APPLMODAL);
               return(1);
           }


           /* Start the application by putting the channel in onhook state */
           if (dx_sethook(dxinfo[numchan].chdev,DX_ONHOOK,EV_ASYNC) == -1) {
              sprintf(tmpbuf,"FAILED: dx_sethook(%s): %s (error #%d) ",
ATDV_NAMEP(dxinfo[numchan].chdev), ATDV_ERRMSGP(dxinfo[numchan].chdev),
ATDV_LASTERR(dxinfo[numchan].chdev));
```

```
            MessageBox(hWnd, tmpbuf, "ERROR",MB_OK|MB_APPLMODAL); return(1);
         }
      }

      return(0);
}

/*****************************************************************************
 *        NAME : DialogicClose()
 * DESCRIPTION : Tier down of Dialogic devices
 *    CAUTIONS : none.
*****************************************************************************/
void DialogicClose(HWND hWnd)
{

      int numchan;

      /* Close all voice devices before exiting */
      for (numchan=1;numchan<=MAXCHAN;numchan++) {
         // attempt to stop the channel dx_stopch(dxinfo[numchan].chdev,EV_SYNC);
         dx_close(dxinfo[numchan].chdev);
      }


}

/*************************************************************************
 *        NAME: disp_status(hwnd, chnum, stringp)
 *      INPUTS: chno - channel number (1 - 12)
 *              stringp - pointer to string to display
  * DESCRIPTION: display the current activity on the channel in window 2
 *              (the string pointed to by stringp) using chno as a Y offset
*************************************************************************/
void disp_status(HWND hWnd, int channum, char *stringp)
{
      RECT rect;


      // get the entire window rectangle and modify it
      GetClientRect(hWnd, &rect);
      rect.top = (channum+2) * ROWHEIGHT;
      rect.bottom = (channum+3) * ROWHEIGHT;


      // buffer the message
      sprintf(gRowVal[channum], "Channel %d: %s", channum, stringp);

      InvalidateRect(hWnd, &rect, TRUE);
      UpdateWindow(hWnd);

}
```

5.5. Asynchronous with Win32 Synchronization Model

The Asynchronous with Win32 Synchronization model allows an asynchronous application to receive SRL event notification through standard Win32 synchronization mechanisms. The two mechanisms supported are Reset Events

and I/O Completion Ports. In this model, the application informs the SRL to signal the specified Reset Event or I/O Completion Point when an event occurs on a Dialogic device. When the application receives notification, it calls standard Dialogic event retrieval functions to process the event.

Choose the Asynchronous with Win32 Synchronization model for an application that needs the tightest possible integration with other Win32 devices.

Asynchronous with Win32 Synchronization Model Advantages

- Because the Asynchronous with Win32 Synchronization model uses one thread for all Dialogic devices, it requires a lower level of system resources than does the Synchronous model; thus, Asynchronous with Win32 Synchronization model allows for greater scalability in growing systems.
- Provides a single point of event synchronization for devices of separate sources, including DM3 devices.
- Minimizes inter-thread communication because all devices can be controlled within a single thread.
- Provides a high level of scalability to multi-processing platforms. This is especially true for I/O Completion Ports.

Asynchronous with Win32 Synchronization Model Programming Notes

For reset events, an application should perform the following operations.

When the application initializes:

1. Use the Win32 **CreateEvent( )** function to create the event.
2. Fill in the **SRLWIN32INFO** structure to indicate that reset events are being used. The *dwHandleType* field must be set to **SR_RESETEVENT**. The *ObjectHandle* field must be set to the event handle returned by the **CreateEvent( )** function.
3. Call the **sr_setparm( )** function with the parameter number argument set to **SR_WIN32INFO**.

To wait for event notification:

1. Call the Win32 **WaitForSingleObject( )** or **WaitForMultipleObjects( )** function. The **WaitForMultipleObjects( )** function is most commonly used because the application can wait for event notification from multiple sources.
2. At notification, if the event source is a Dialogic device, call the **sr_waitevt()** function, with a zero timeout, to retrieve the event from the Dialogic event queue. To identify the Dialogic device and event, use the **sr_getevtdev( )** and **sr_getevttype( )** functions.

For I/O Completion Ports, an application should perform the following operations:

When the application initializes:

1. Use the Win32 **CreateIoCompletionPort( )** function to create the Completion Port.
2. Fill in the **SRLWIN32INFO** structure to indicate that I/O Completion Points are being used. The *dwHandleType* field must be set to **SR_IOCOMPLETIONPORT**. The *ObjectHandle* field contains the Completion Port handle returned by the **CreateIoCompletionPort( )** function. The *dwUserKey* field must be set to the user-specified key that is returned at the time of event notification. The *lpOverlapped* field may be set to an optional user-specified **OVERLAPPED** structure.
3. Call the **sr_setparm( )** function with the parameter number argument set to **SR_WIN32INFO**.

To wait for event notification:

1. Call the Win32 **GetQueuedCompletionStatus( )** to wait for events from multiple sources, including SRL devices.
2. At notification, if the event source is a Dialogic device, call the **sr_waitevt()** function, with a zero timeout, to retrieve the event from the Dialogic event queue. To identify the Dialogic device and event, use the **sr_getevtdev( )** and **sr_getevttype( )** functions.

Asynchronous with Win32 Synchronization Model Example

An example of the Asynchronous with Win32 Synchronization model is shown below:

```
/*
 * This asynchronous with Win32 synchronization sample application was
 * designed to work with D/41ESC, VFX/40ESC, LSI/81SC, LSI/161SC and
 * D/160SC-LS boards only.  It was compiled using MS-VC++.
 */

/* C includes */
#include <stdio.h>
#include <process.h>
#include <conio.h>
#include <ctype.h>
#include <windows.h>

/* Dialogic includes */
#include <srllib.h>
#include <dxxxlib.h>

/* Defines */
#define MAX_CHAN     4 /* maximun number of voice channels in system */

#define DIALOGIC_KEY      0  /* Index for Dialogic reset event */
#define KEYBOARD_KEY      1  /* Index for keyboard reset event */
#define MAX_RESET_EVENTS  2  /* number of reset events */


/* Globals */
int Kbhit_flag = 0;
HANDLE hEvent[MAX_RESET_EVENTS];

/* Prototypes */
int main();
DWORD WINAPI sample_begin(LPVOID);


/*****************************************************************
*        NAME : int main()
* DESCRIPTION : create reset events, create thread and
*             : poll for keyboard input
*       INPUT : none
*      OUTPUT : none
*     RETURNS : 0 on success; 1 if a failure was encountered
*    CAUTIONS : none
*****************************************************************/
int main()
{
   HANDLE thread_handle;
   DWORD threadID;
   DWORD Index;

   /* show application's title */
   printf("Async with Win32 Sychronization Sample Application - hit any key to
exit...\n");
```

```
   /* create the reset events */
   for (Index = 0; Index < MAX_RESET_EVENTS; Index++) {
      hEvent[Index] = CreateEvent((LPSECURITY_ATTRIBUTES)NULL,
                                  FALSE,
                                  FALSE,
                                  NULL);
   }

   /* create one thread to process all the voice channels */
   if ((thread_handle = (HANDLE)_beginthreadex(NULL,
                                0,
                                sample_begin,
                                (LPVOID)NULL,
                                0,
                                &threadID)) == (HANDLE)-1) {
         /* Perform system error processing */
         exit(1);
   }

   /* wait for Keyboard input to shutdown program */
   getch();

   Kbhit_flag++;  /* let thread know it's time to abort */
   SetEvent( hEvent[KEYBOARD_KEY] );

   /* sleep here until thread has terminated */
   if (WaitForMultipleObjects(1, &thread_handle, TRUE, INFINITE)
         == WAIT_FAILED) {
       printf("ERROR: Failed WaitForMultipleObjects(): error = %ld\n",
              GetLastError());
   }

   return(0);
}


/*****************************************************************
*        NAME : DWORD WINAPI sample_begin(LPVOID argp)
* DESCRIPTION : do all channel specific processing
*       INPUT : LPVOID argp - NULL pointer (not used)
*      OUTPUT : none
*     RETURNS : 0 on success; 1 if a failure was encountered
*    CAUTIONS : none
*****************************************************************/
DWORD WINAPI sample_begin(LPVOID argp)
{
   char channame[20];
   int chdesc;
   int cnt;
   int hDevice[MAX_CHAN];
   long EventCode;
   int Index;
   SRLWIN32INFO SrlWin32Info;

   /*
    * First thing is to inform SRL to signal the reset event
    * when a Dialogic event occurs
    */
   SrlWin32Info.dwTotalSize  = sizeof(SRLWIN32INFO);
   SrlWin32Info.dwHandleType = SR_RESETEVENT;
   SrlWin32Info.ObjectHandle = hEvent[DIALOGIC_KEY];
   if ( sr_setparm(SRL_DEVICE, SR_WIN32INFO, (void *)&SrlWin32Info)
```

```
          == -1) {
    printf("SRL - FAILED sr_setparm( SR_WIN32INFO ): %s (error #%d)\n",
            ATDV_ERRMSGP(SRL_DEVICE), ATDV_LASTERR(SRL_DEVICE));
    return(1);
}

for (cnt = 0; cnt < MAX_CHAN; cnt++) {
   /* build name of voice channel */
   sprintf(channame, "dxxxB%dC%d", (cnt / 4) + 1,
                                   (cnt % 4) + 1);

   /* open voice channel */
   if ((chdesc = dx_open(channame, 0)) == -1) {
      printf("%s - FAILED: dx_open(): system error = %d\n",
              channame, dx_fileerrno());
      return(1);
   }
   hDevice[cnt] = chdesc;
   printf("%s - Voice channel opened\n", ATDV_NAMEP(chdesc));

   /* kick off the state machine by going offhook asynchronously */
   if (dx_sethook(chdesc, DX_OFFHOOK, EV_ASYNC) == -1) {
      printf("%s - FAILED: dx_sethook(DX_OFFHOOK): %s (error #%d)\n",
              ATDV_NAMEP(chdesc), ATDV_ERRMSGP(chdesc), ATDV_LASTERR(chdesc));
      return(1);
   }
   printf("%s - Voice channel off-hook initialized\n",ATDV_NAMEP(chdesc));
}

/* loop until Keyboard input is received */
while (!Kbhit_flag) {
   /*
    * Wait on the reset events
    */
   if ((Index = WaitForMultipleObjects(MAX_RESET_EVENTS,
                                       hEvent,
                                       FALSE,
                                       INFINITE)) == WAIT_FAILED) {
      printf("ERROR: Failed WaitForMultipleObjects(): error = %ld\n",
             GetLastError());
   }

   /* check if it is because of a key hit */
   if (Index == KEYBOARD_KEY) {
      continue;
   }

   /* must be a Dialogic event so process it */
   sr_waitevt(0);

   /*
    * gather data about the event
    */
   chdesc = sr_getevtdev(0);
   EventCode = sr_getevttype(0);

   switch(EventCode) {
   case TDX_SETHOOK:
      if (ATDX_HOOKST(chdesc) == DX_OFFHOOK) {
         printf("%s - Voice channel off-hook\n",ATDV_NAMEP(chdesc));

         /* we went off hook so start dialing */
```

```
            if (dx_dial(chdesc, "12025551212", NULL, EV_ASYNC) == -1) {
               printf("%s - FAILED: dx_dial(): %s (error #%d)\n",
                  ATDV_NAMEP(chdesc), ATDV_ERRMSGP(chdesc), ATDV_LASTERR(chdesc));
               return(1);
            }
            printf("%s - Voice channel dialing initialized\n",ATDV_NAMEP(chdesc));
         } else {
            /* we went on hook so go off hook again */
            printf("%s - Voice channel on-hook\n",ATDV_NAMEP(chdesc));

            /* set the voice channel off-hook */
            if (dx_sethook(chdesc, DX_OFFHOOK, EV_ASYNC) == -1) {
               printf("%s - FAILED: dx_sethook(DX_OFFHOOK): %s (error #%d)\n",
                  ATDV_NAMEP(chdesc), ATDV_ERRMSGP(chdesc), ATDV_LASTERR(chdesc));
               return(1);
            }
            printf("%s - Voice channel off-hook initialized\n",ATDV_NAMEP(chdesc));
         }
         break;

      case TDX_DIAL:
         printf("%s - Voice channel Done dialing\n",ATDV_NAMEP(chdesc));

         /* done dialing so set the voice channel on-hook */
         if (dx_sethook(chdesc, DX_ONHOOK, EV_ASYNC) == -1) {
            printf("%s - FAILED: dx_sethook(DX_ONHOOK): %s (error #%d)\n",
               ATDV_NAMEP(chdesc), ATDV_ERRMSGP(chdesc), ATDV_LASTERR(chdesc));
            return(1);
         }
         printf("%s - Voice channel on-hook initialized\n",ATDV_NAMEP(chdesc));
         break;

      default:
         printf("Received unexpected event 0x%X on device %d\n", EventCode, chdesc);
         break;

      }

   }

   for (cnt = 0; cnt < MAX_CHAN; cnt++) {
      /* close the voice channel */
      chdesc = hDevice[cnt];
      printf("%s - Voice channel closing\n",ATDV_NAMEP(chdesc));
      if (dx_close(chdesc) == -1) {
         printf("%s - FAILED: dx_close(): %s (error #%d)\n",
                ATDV_NAMEP(chdesc), ATDV_ERRMSGP(chdesc), ATDV_LASTERR(chdesc));
         return(1);
      }
   }

   return(0);
}
```

6. SRL Event Management Functions

This chapter provides a complete reference for the Dialogic Standard Runtime Library (SRL) event management functions.

6.1. Event Management Function Overview

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

Event handling	• Enable and disable event handlers or wait a specified amount of time for the next event
Event data retrieval	• Obtain information about the current event
SRL parameter	• Set and request SRL parameters

The event management functions are listed below. *Section 6.4. SRL Event Management Function Reference Overview* describes each of these functions in detail.

6.1.1. Event Handling Functions

sr_enbhdlr( )	• Enable an event handler
sr_dishdlr( )	• Disable an event handler
sr_waitevt( )	• Wait for next event
sr_waitevtEx( )	• Wait for events on certain devices
sr_NotifyEvent( )	• Sends event notification to a window
sr_putevt( )	• Adds an event to the SRL event queue

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 *Section 2.5. SRL Extended Asynchronous Programming* for the hierarchy in which the SRL calls handlers.

In the Asynchronous and Extended Asynchronous programming models, the thread does not receive events until the function **sr_waitevt( )** or **sr_waitevtEx( )** is called. When an event occurs (or has previously occurred) on the device in **sr_waitevt( )**, all appropriate event handlers for the event are called before **sr_waitevt( )** returns.

See *Chapter 3. Basic SRL Programming Models* for information on Asynchronous and Extended Asynchronous programming models.

6.1.2. Event Data Retrieval Functions

sr_getevtdev( )	• Get the Dialogic handle for the current event
sr_getevttype( )	• Get the event type for the current event
sr_getevtlen( )	• Get the length of variable data associated with the current event
sr_getevtdatap( )	• Get a pointer to the variable data associated with the current event

Event data retrieval functions retrieve information about the current event. This allows data extraction and event processing.

6.1.3. SRL Parameter Functions

sr_setparm( )	• Set an SRL parameter
sr_getparm( )	• Get an SRL parameter
sr_getboardcnt( )	• Get the number of boards of a specific type
sr_libinit( )	• Initialize the Standard Runtime Library DLL*
sr_GetDllVersion( )	• Get the Standard Runtime Library DLL Version Number*

*These functions are only used with the Cross-Compatibility library.

SRL parameter functions check the status of and set the value of SRL parameters.

6.2. SRL Error Handling

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

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

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

If a function fails, the error can be retrieved using the **ATDV_LASTERR( )** or **ATDV_ERRMSGP( )** functions described in *Chapter 7.* SRL Standard Attribute Functions. If an SRL function fails, retrieve the error by using the **ATDV_LASTERR( )** function with **SRL_DEVICE** as the argument. To retrieve a text description of the error, use **ATDV_ERRMSGP**.

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

Each function description in this chapter includes a list of the errors that can occur for that function. If the error returned by **ATDV_LASTERR( )** is ESR_SYS, an error from the operating system has occurred; use **dx_fileerrno( )** to obtain the system error value.

The error codes are defined in *srllib.h* and are listed in *Table 3*.

Table 3. Event Management Function Errors

Error Define	Error String
ESR_DATASZ	• Invalid size for default event data memory
ESR_MODE	• Illegal mode for this operation
ESR_NOERR	• No standard runtime library errors
ESR_NOHDLR	• No such handler
ESR_NOTIMP	• Function is not implemented
ESR_PARMID	• Unknown parameter ID

Error Define	Error String
ESR_QSIZE	• Illegal event queue size
ESR_SCAN	• Standard runtime library scanning function returned an error
ESR_SYS	• Error from operating system; use **dx_fileerrno( )** to obtain error value.
ESR_TMOUT	• Returned by ATDV_LASTERR(SRL_DEVICE) when an standard runtime library function timed out.

6.3. SRL Event Management Functions Include Files

The following lines must be included in application code prior to calling any event management functions:

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

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

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

6.4. SRL Event Management Function Reference Overview

This section provides a detailed description of the event management functions in alphabetical order.

NOTE: The functions described in this section use and return device-specific data. See the technology-specific *Programmer's Guide* for further information (such as the *VoiceSoftware Reference: Programmer's Guide*).

Name:	long sr_dishdlr(dev, evt_type, handler)	
Inputs:	long dev	• Dialogic device handle
	long evt_type	• event type
	long (*handler)(unsigned long parm)	• event handling function
Returns:	0 if success -1 if failure	
Includes:	srllib.h	
Type:	Event Handling function	

■ 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 **sr_dishdlr( )** function can be called from within a handler even if the same handler is being disabled.

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 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 technology-specific *Programmer's 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

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 SRL Extended Asynchronous Programming model.

■ Example

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


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;

   /* 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 );
   }
}
```

■ Errors

If this function returns -1 to indicate failure, obtain the reason for the error by calling the SRL 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.

■ See Also

- **sr_enbhdlr( )**
- The appropriate technology-specific *Programmer's Guide*.

Name:	long sr_enbhdlr(dev, evt_type, handler)	
Inputs:	long dev	• Dialogic device handle
	long evt_type	• event type
	long (*handler)(unsigned long parm)	• event handling function
Returns:	0 if success -1 if failure	
Includes:	srllib.h	
Type:	Event Handling function	

■ Description

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

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 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 technology-specific *Programmer's 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.

If more than one handler is enabled for an event when the event is detected, SRL calls all specified handlers. Also, more general handlers can be enabled that handle **any** event on a given device, or handlers can be enabled to handle **any** event on **any** device.

The following actions are valid from within a handler:

- Calling a technology-specific atomic function
- Initiating any technology-specific multitasking function **asynchronously**
- Opening or closing a device
- Enabling an event handler
- Disabling an event handler, including the current one

The following actions are not valid from within a handler:

- Initiating any technology-specific multitasking function **synchronously**
- Calling the **sr_waitevt( )** function

Handlers must return 1 to advise SRL to keep the event, or 0 to advise SRL to release the event. If a handler returns 0 and the event is released, **sr_waitevt( )** does not return for that event. See *Section 2.5. SRL Extended Asynchronous Programming* for the hierarchy in which SRL calls handlers.

The handler is called from within the function **sr_waitevt( )**. For details, see the appropriate technology-specific *Programmer's Guide*.

You can enable a handler from within another handler because SRL'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.

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

■ Cautions

If two handlers are enabled for the same event on the same device, the order in which they are called cannot be controlled.

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; 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.

For more information, see the appropriate technology-specific *Programmer's Guide*.

■ Example

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


long int dx_handler(unsigned long evhandle)
{
     printf( "dx_handler() called, event is 0x%x\n", sr_getevttype(evhandle));
     return( 0 );                              }

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

   /* 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 );
   }
}
```

Errors

If this function returns -1 to indicate failure, obtain the reason for the error by calling the SRL 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.

See Also

- **sr_dishdlr( )**
- The appropriate technology-specific *Programmer's Guide*

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

Type: SRL Parameter

■ Description

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. Valid defines for class name are:

DEV_CLASS_VOICE	• For voice boards
DEV_CLASS_DTI	• For DTI boards
DEV_CLASS_MSI	• For MSI boards

This function returns the number of boards about which an application needs knowledge. In the case of voice, the number of four-channel boards is returned. Standard names are applied as shown below:

DEV_CLASS_VOICE	dxxxB?C?
DEV_CLASS_DTI	dtiB?T?
DEV_CLASS_MSI	msiB?C?

The **?**s above represent board or channel numbers (incremental, starting with 1).

Parameter	Description
class_namep	Pointer to class name
boardcntp	Pointer where to return count of boards of this class

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 */
```

Name:	long sr_GetDllVersion(dwfileverp, dwprodverp)	
Inputs:	LPDWORD dwfileverp	• SRL DLL Version Number
	LPDWORD dwprodverp	• Product version of this release
Returns:	0 if success	
	-1 if failure	
Includes:	srllib.h	

■ Description

The **sr_GetDllVersion( )** function returns the SRL DLL Version Number for the file and product.

This function has the following parameters:

Parameter	Description
dwfileverp	Pointer to where to return file version information
dwprodverp	Pointer to where to return product version information

■ Example

```
/*$ sr_GetDllVersion( ) example $*/

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

int InitDevices( )
{
   DWORD dwfilever, dwprodver;

   /*************************************************************************
   * Initialize all the DLLs required. This will cause the DLLs to be
   * loaded and entry points to be resolved. Entry points not resolved
   * are set up to point to a default not implemented function in the
   * 'C' library. If the DLL is not found all functions are resolved
   * to not implemented.
   *************************************************************************/

   if (sr_libinit(DLGC_MT) == -1) {
      /* Must be already loaded, only reason if sr_libinit( ) was already called */
   }
   /*****************************************************************************
   /* SRL library initialized so all other SRL functions may be called as normal.
   * Display the version number of the DLL
   *****************************************************************************/
   sr_GetDllVersion(&dwfilever, &dwprodver);
   printf("File Version for SRL is %d.%02d\n",
```

```
                              HIWORD(dwfilever), LOWORD(dwfilever));
   printf("Product Version for SRL is %d.%02d\n",
                              HIWORD(dwprodver), LOWORD(dwprodver));

   /* Call technology specific xx_libinit( ) functions */
}
```

■ Caution

This function may only be used with the Cross-Compatibility Library.

■ See Also

- **dx_GetDllVersion( )**
- **fx_GetDllVersion( )**
- **dt_GetDllVersion( )**
- **vr_GetDllVersion( )**

Name: void *sr_getevtdatap(ehandle)

Inputs: 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

Type: Event Data Retrieval function

■ Description

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

NOTE: Information about the data that can be returned by this function can be found in the appropriate technology-specific *Programmer's Guide*

■ Cautions

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

■ 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);
       int *datap;
       switch(sr_getevttype(ehandle)) {
       case TDX_CST:
              *datap  = (int *) sr_getevtdatap(ehandle);
              if (datap->csd_data == DE_RINGS)
              {
               .
               .
               }
              break;

        case TDX_PLAY:
                .
              break;

        }

 }
```

See Also

- **sr_getevtlen( )**
- The appropriate technology-specific *Programmer's Guide*

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

Returns: Dialogic device handle if successful
-1 if no current event
Includes: srllib.h
Type: Event Data Retrieval function

■ Description

The **sr_getevtdev( )** function returns the Dialogic device handle associated with the current event. If 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 technology-specific *Programmer's Guide*.

■ Cautions

■ For applications using sr_waitevt( ), pass a 0 (zero) parameter: sr_getevtdev(0).

■ 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 technology-specific *Programmer's Guide*

Name:	long sr_getevtlen(ehandle)	
Inputs:	unsigned long ehandle	• event handle returned by **sr_waitevtEx( )**, or zero if **sr_waitevt( )** is used
Returns:	length of data if successful -1 if no current event	
Includes:	srllib.h	
Type:	Event Data Retrieval function	

■ Description

The **sr_getevtlen( )** function returns the length of the variable data associated with the current event. If there is no 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 technology-specific *Programmer's Guide.*

■ Cautions

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**.

■ 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 technology-specific *Programmer's Guide*

Name: long sr_getevttype(ehandle)

Inputs: 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

Type: Event Data Retrieval function

■ Description

The **sr_getevttype( )** function returns the event type for the current event. 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 technology-specific *Programmer's Guide.*

■ Cautions

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**.

■ 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 technology-specific *Programmer's Guide*

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	
Type:	SRL Parameter function	

■ Description

The **sr_getparm( )** function returns the value of an SRL parameter. The function parameters are described as follows:

Parameter	Description
dev	Device handle. Generally, you should set this parameter to SRL_DEVICE, the predefined SRL 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 SRL parameter for which values are returned. Possible values are as follows: • SR_MODELTYPE — Get the value for the model type. Returns an integer. • SR_INTERPOLLID — Get the value for the polling granularity (the time between device polls in millisecond units). Returns an integer. • SR_USERCONTEXT — Set user-specific context. This lets you quickly get application-specific context given on a dialogic device handle.
valuep	A 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_MODELTYPE — SR_STASYNC or

Parameter	Description	
		SR_MTASYNC .
	• SR_INTERPOLLID	A number in milliseconds (ms).
	• SR_USERCONTEXT	User-specific context set through the **sr_setparm( )** function.

■ Cautions

Normally, when getting SRL 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.

■ 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" );
}
```

■ Errors

If this function returns -1 to indicate failure, obtain the reason for the error by calling the SRL 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.

Name:	long sr_libinit(flags)	
Inputs:	unsigned short flags	• Specifies the programming model
Returns:	0 if success -1 if failure	
Includes:	srllib.h	

■ Description

The **sr_libinit( )** function initializes the Standard Runtime Library DLL by loading and resolving all entry points in *LIBSRLMT.DLL*.

This function has the following parameter:

Parameter	Description
flags	This flag has two possible values: • DLGC_MT - Specify if using a multi-threaded or window callback model. • DLGC_ST - Specify if using the single threaded model.

This function need only be used if you are linking with the Cross-Compability Library.

■ Cautions

The **sr_libinit( )** function must be called prior to using any other technology specific **xx_libinit( )** functions.

■ Example

```
/* sr_libinit( ) example */

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

int InitDevices( )
{
   DWORD dwfilever, dwprodver;

   /*****************************************************************
   * Initialize all the DLLs required. This will cause the DLLs to be
```

```
   * loaded and entry points to be resolved. Entry points not resolved
   * are set up to point to a default not implemented function in the
   * 'C' library. If the DLL is not found all functions are resolved
   * to not implemented.
   ************************************************************************/

   if (sr_libinit(DLGC_MT) == -1) {
      /* Must be already loaded, only reason if sr_libinit( ) was already called */
   }
   /*****************************************************************************
   * SRL library initialized so all other SRL functions may be called as normal.
   * Display the version number of the DLL
   *****************************************************************************/

   sr_GetDllVersion(&dwfilever, &dwprodver);
   printf("File Version for SRL is %d.%02d\n",
                              HIWORD(dwfilever), LOWORD(dwfilever));
   printf("Product Version for SRL is %d.%02d\n",
                              HIWORD(dwprodver), LOWORD(dwprodver));

   /* Call technology specific xx_libinit( ) functions */

}
```

■ Errors

The **sr_libinit( )** function fails if the library has already been initialized. For example, if you try to make a second call to **sr_libinit( )**, it fails.

■ See Also

- **dx_libinit( )**
- **fx_libinit( )**

Name:	long sr_NotifyEvent(handle, message, flags)	
Inputs:	HWND handle	• Window handle
	unsigned int message	• message number to sent to window
	unsigned int flags	• notification on/off flag
Returns:	0 if success -1 if failure	
Includes:	srllib.h	
Type:	Event handling	

■ Description

The **sr_NotifyEvent( )** function informs the SRL to send event notification to a window. This provides a method of getting notification of Dialogic events through the Window's event queue. Although the actual event does not come in through the Window's 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 Dialogic 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 Dialogic event queue.

NOTES: **1.** The sr_NotifyEvent function should be called before any Dialogic devices are opened.

2. 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.

■ 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:
                              MyPrintf("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 technology-specific *Programmer's Guide*

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	
Type:	Event Management function	
Mode:	Synchronous	

■ Description

The **sr_putevt( )** function allows the application to add an event to the SRL 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 SRL makes of 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. The function parameters are described as follows:

Parameter	Description
dev	Valid device handle of the device opened through either setting the **dev** parameter to SRL_DEVICE or 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 the **evtdatap** parameter to NULL.
evtdatap	Pointer to the variable-length data of the size in the **evtlen** parameter. The SRL makes a copy of this data and attaches it to

Parameter	Description
	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 the **evtlen** parameter 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_waitevt(Ex)** 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 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.

■ Example

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

int        dev;     /* Dialogic 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) != SR_SUCCESS)
{
   printf("_sr_putevt failed - %s", ATDV_ERRMSGP(SRL_DEVICE)));
```

```
    .
    .
    .
}
```

Errors

The **sr_putevt( )** function returns SR_NOMEM if memory cannot be allocated to store the event on the SRL event queue.

See Also

- **sr_waitevt( )**

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	
Type:	SRL Parameter function	

■ Description

The **sr_setparm( )** function allows the application to set the value of an SRL 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 SRL 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	Value for the SRL parameter to be changed. Possible values are as follows:	
	• SR_MODELTYPE	Set the model type to turn off creation of internal thread used to service event handler action
	• SR_INTERPOLLID	Set the polling granularity parameter (the time between device polls expressed in milliseconds).
	• SR_USERCONTEXT	Set user-specific context. This lets you quickly set application-specific context on a given Dialogic device handle.
	• SR_WIN32INFO	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_MODELTYPE	The value is expected to point to an integer that contains SR_STASYNC or SR_MTASYNC.
	• SR_INTERPOLLID	The value is expected to point to an integer that contains the polling granularity expressed in millisecond (ms) units.
	• SR_USERCONTEXT	The value is expected to point to arbitrary user-supplied data.
	• SR_WIN32INFO	The value is expected to point to either an SRLWIN32INFO structure that indicates the Win32 synchronization method or to NULL to disable Win32 notification.

For more information on Win32 integration and on use of the SR_WIN32INFO parameter, see section 5.5. Asynchronous with Win32 Synchronization Model.

■ Cautions

Normally, when setting SRL 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.

■ Example

```
#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 );
   }
}
```

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 Dialogic device handle. DeviceIndex retrieves the
  // 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]);
}
```

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)
}
```

■ Errors

If this function returns -1 to indicate failure, obtain the reason for the error by calling the SRL 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.

■ See Also

- **sr_getparm( )**

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

■ Description

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

Parameter	Description
timeout	A timeout value of -1 instructs **sr_waitevt( )** to wait indefinitely for the next event. Use timeout 0 when the SRL event must return imediately, regardlesss of whether or not an event is present in the SRL event queue.

The **sr_waitevt( )** function returns a value when the next event occurs, or when a timeout is reached. If an event occurs, the value returned is the number of milliseconds before the timeout. If no event occurs before the timeout, the value returned is -1. If the function times out, a timeout event occurs on the SRL_DEVICE. The **sr_getevtdev(**0**)** function returns SRL_DEVICE, the Dialogic device handle associated with the current event. The **sr_getevttype(**0**)** function returns SR_TMOUTEVT, which is the current event type.

■ Cautions

When an application receives an event, the event must be handled immediately and event-specific information should be retrieved before the next call to **sr_waitevt( )** because **sr_waitevt( )** automatically removes the current event before waiting for the next event; previous information is overwritten.

The **sr_waitevt( )** function cannot be called from within a handler.

You cannot use **sr_waitevt( )** and **sr_waitevtEx( )** functions in the same thread.

If you use the **sr_waitevt( )** function to retrieve events, pass 0 as a parameter when using the following functions:

- **sr_getevtdatap(**0**)**
- **sr_getevtdev(**0**)**
- **sr_getevtlen(**0**)**
- **sr_getevttype(**0**)**

■ 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 );
}
```

■ Errors

If this function returns -1 to indicate failure, obtain the reason for the error by calling the SRL 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.

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 if success
-1 if timeout

Includes: srllib.h

Type: Event Control function

■ Description

The **sr_waitevtEx( )** function waits for events on certain devices. This function is an extended version of the standard **sr_waitevt( )** function. It is used to asynchronously wait for any event to occur on any of the handles passed to the functions. It detects events from devices opened through any of the device-specific **xx_open( )** function calls for that device.

Parameter	Description
handlep	Points to an array of opened handles.
handlecnt	Indicates how many devices are contained in the array.
timeout	A timeout value of -1 instructs **sr_waitevt( )** to wait indefinitely for the next event. Use timeout 0 when the SRL event must return imediately, regardlesss of whether or not an event is present in the SRL 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 can be used in a single-threaded or multithreaded application. Furthermore, 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.

■ Cautions

The **sr_waitevt( )** and **sr_waitevtEx( )** functions should not be used in the same application.

The **sr_waitevtEx( )** function can **NOT** be used in conjunction with handlers.

■ Example

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

           }

    }
```

■ Errors

If this function returns -1 to indicate failure, obtain the reason for the error by calling the SRL 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.
ESR_TMOUT	• Timed out waiting for event

■ See Also

- **sr_waitevt( )**
- **sr_getevtdev( )**
- **sr_getevttype( )**
- **sr_getevtlen( )**
- **sr_getevtdatap( )**
- The appropriate technology-specific *Programmer's Guide*

7. SRL Standard Attribute Functions

This chapter provides a complete reference for the Dialogic Standard Runtime Library (SRL) standard attribute functions.

7.1. SRL Standard Attribute Functions Overview

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

All standard attribute function names adhere to the following naming conventions:

- The function name is all capital letters.
  - The function name is prefixed by "ATDV_"
  - The name after the underscore describes the attribute.

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

ATDV_ERRMSGP( )	• pointer to string describing error on last library call
ATDV_IRQNUM( )	• interrupt being used
ATDV_LASTERR( )	• error that occurred on last device library call
ATDV_NAMEP( )	• pointer to device name
ATDV_SUBDEVS( )	• number of subdevices

7.2. SRL Standard Attribute Functions Include Files

The following lines must be included in application code prior to calling any standard attribute functions:

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

*DEVICE*lib.h is the header file for the device being used.

NOTE: *srllib.h* must be included in code before all other Dialogic header files library call. For example, if the device is a D/4x, the *dxxxlib.h* file is included.

7.3. SRL Standard Attribute Function Reference Overview

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

Refer to the appropriate technology-specific *Programmer's Guide* for a list of information returned for specific devices.

Name: char * ATDV_ERRMSGP(dev)
Inputs: int dev • valid Dialogic device handle
Returns: pointer to string
Includes: srllib.h
Category: standard attribute

■ Description

The **ATDV_ERRMSGP( )** function returns a pointer to an ASCIIZ string that describes the last error that occurred on this device. 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	Device handle, the handle returned by the technology-specific **xx_open( )**.

■ Example

```
#include <windows.h>

#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 ));
     }
}
```

■ Errors

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

■ See Also

- The appropriate technology-specific *Programmer's Guide*

Name:	long ATDV_IRQNUM(dev)	
Inputs:	int dev	• valid Dialogic device handle
Returns:	AT_FAILURE if failure otherwise IRQ of device	
Includes:	srllib.h	
Category:	standard attribute	

■ Description

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

Parameter	Description
dev	Device handle, the handle returned by the technology-specific **xx_open( )**.

■ Example

```
#include <windows.h>
#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 ));
}
```

■ 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**.

Name: long ATDV_LASTERR(dev)
Inputs: int dev • valid Dialogic device handle
Returns: AT_FAILURE if an invalid device handle; otherwise a valid error number
Includes: srllib.h
Category: standard attribute

■ Description

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.

Parameter	Description
dev	Device handle, the handle returned by the technology-specific **xx_open( )**.

■ Example

```
#include <windows.h>

#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 ));
   }
}
```

■ Errors

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

■ See Also

- The appropriate technology-specific *Programmer's Guide*

Name: char * ATDV_NAMEP(dev)
Inputs: int dev • valid Dialogic device handle
Returns: pointer to string
Includes: srllib.h
Category: standard attribute

■ Description

The **ATDV_NAMEP( )** function returns a pointer to an ASCIIZ string that specifies a device name, used to open the device.

An example of a device name is:

- dxxxB*b*C*c*

where:

- *b* is the number of the board in the system
- *c* is the number of the channel on the D/4x board

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

Parameter	Description
dev	Device handle, the handle returned by the technology-specific **xx_open( )**.

■ Example

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


main()
{
   int dxxxdev;

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

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

■ Errors

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

■ See Also

- The appropriate technology-specific *Programmer's Guide*

Name: long ATDV_SUBDEVS(dev)
Inputs: int dev • valid Dialogic device handle
Returns: AT_FAILURE if failure, otherwise number of subdevices
Includes: srllib.h
Category: standard attribute

■ Description

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

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

Parameter	Description
dev	Device handle, the handle returned by the technology-specific **xx_open( )**.

■ Example

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

main()
{
   int dxxxdev;

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

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

■ Errors

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

See Also

- The appropriate technology-specific *Programmer's Guide*

8. DV_TPT Termination Parameter Table Structure

DV_TPT structures

The termination parameter table (TPT) specifies termination conditions for Dialogic devices. It is made up of DV_TPT structures contained in the *srllib.h* file and arranged in one of the following ways:

- Linked list of DV_TPT structures
- Array of DV_TPT structures
- Combined linked list and array of DV_TPT structures

The structure is used to set termination conditions for multitasking functions.

Description of the typedef for the DV_TP

The typedef for the DV_TPT structure is shown below:

```
typedef struct DV_TPT {
   unsigned short  tp_type;           /* Flags describing this structure */
   unsigned short  tp_termno;         /* Termination parameter number */
   unsigned short  tp_length;         /* Length of terminator */
   unsigned short  tp_flags;          /* Term. parameter attributes flag */
   unsigned short  tp_data;           /* Optional additional data */
   unsigned short  rfu;               /* Reserved for future use */
   struct DV_TPT  *tp_nextp;         /* Pointer to next term. parameter if */
                                      /* IO_LINK is set */
} DV_TPT;
```

where:

Parameter	Description
• **tp_type**	Describes whether the structure is part of a linked list (IO_LINK), part of an array (IO_CONT), or the last DV_TPT entry in the DV_TPT table (IO_EOT).
• **tp_termno**	Specifies what conditions should terminate functions.

Parameter	Description
• **tp_length**	Specifies the length or size for each termination condition.
• **tp_flags**	Represents various characteristics of the termination condition specified in **termno**.
• **tp_data**	Specifies optional additional data.
• **tp_nextp**	Contains a pointer to the next DV_TPT structure in a linked list.

Depending on the type of device (for example, D/4x, DTI/xxx), these fields can have different values. For a specific example, see the *Voice Software Reference: Programmer's Guide*.

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 through a physical library.

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

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

Dialogic Standard Runtime Library (SRL): Device-independent library that consists of event management functions and standard attribute functions.

Event: Any message sent from the device.

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

Event management functions: SRL functions that connect and disconnect events to application-specified event handlers, allowing the user to retrieve and handle events when they occur on a device.

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

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

Subdevices: Any device that is a direct child of another device, for example, a channel is a subdevice of a board device. Because “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, such as a silence-on or silence-off event on a channel.

Index

A

B

C

D

E

S

T

NOTES

NOTES

NOTES