# Continuous Speech Processing API for Linux and Windows Operating Systems

## Library Reference

*November 2003*

05-1700-003

Publication Date: November 2003

Document Number: 05-1700-003

Intel Converged Communications, Inc.
1515 Route 10
Parsippany, NJ 07054

For **Technical Support**, visit the Intel Telecom Support Resources website at:
*http://developer.intel.com/design/telecom/support*

For **Products and Services Information**, visit the Intel Telecom Products website at:
*http://www.intel.com/design/network/products/telecom*

For **Sales Offices** and other contact information, visit the Where to Buy Intel Telecom Products page at:
*http://www.intel.com/buy/wtb/wtb1028.htm*

# Table of Contents

# Figures

# Revision History

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

| Document No. | Publication Date | Description of Revisions |
|---|---|---|
| 05-1700-003 | November 2003 | ec_getblkinfo( ) function reference: New function.<br>ec_getxmitslot( ) function reference: Revised caution about multiple threads.<br>ec_listen( ) function reference: Revised caution about multiple threads.<br>ec_rearm( ) function reference: Added note that this function is no longer supported and is superseded by the new silence compressed streaming feature.<br>ec_reciottdata( ) function reference: Added paragraph about new silence compressed streaming feature. Revised caution about multiple threads. Added information about new TF_IMMEDIATE flag (DV_TPT structure).<br>ec_setparm( ) function reference: Added new ECCH_SILENCECOMPRESS parameter. Added note about data formats under DXCH_EC_TAP_LENGTH description for Springware boards. Revised caution about multiple threads.<br>ec_stopch( ) function reference: Revised caution about multiple threads.<br>ec_stream( ) function reference: Added paragraph about new silence compressed streaming feature. Revised caution about multiple threads. Added information about new TF_IMMEDIATE flag (DV_TPT structure).<br>Data Structures chapter: New chapter. Added new EC_BLK_INFO data structure.<br>Supplementary Reference Information chapter: Removed chapter because the example code no longer applies. New CSP demos now provide extended example code. |
| 05-1700-002 | August 2002 | ATEC_TERMMSK( ) function reference: New function.<br>ec_getxmitslot( ) function reference: Revised to state that this function is now supported on DM3 boards.<br>ec_reciottdata( ) function reference: Revised note about DX_MAXTIME termination condition.<br>ec_setparm( ) function reference: Revised description for DXCH_EC_TAP_LENGTH.<br>ec_stream( ) function reference: Revised note about DX_MAXTIME termination condition.<br>Supplementary Reference Information chapter: New chapter with extended example code. |
| 05-1700-001 | December 2001 | Initial version of document. Much of the information in this document was previously contained in the *Continuous Speech Processing Software Reference*, document number 05-1398-003. |

# About This Publication

The following topics provide information about this *Continuous Speech Processing API for Linux and Windows Operating Systems Library Reference*:

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

## Purpose

This publication provides a reference to all functions and parameters in the Continuous Speech Processing (CSP) library for Intel® telecom products. These functions and parameters are used to build CSP-enabled applications.

This publication is a companion document to the *Continuous Speech Processing API Programming Guide* which provides guidelines for developing applications using the CSP API.

## Intended Audience

This information is intended for:

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

## How to Use This Publication

Refer to this guide after you have installed the hardware and system software.

This guide assumes that you are familiar with the Linux* or Windows* operating system and the C programming language. It is helpful to keep the voice API documentation handy as you develop your application.

The information in this publication is organized as follows:

- Chapter 1, "Function Summary by Category" introduces you to the various categories of functions in the CSP library.

- Chapter 2, "Function Information" provides an alphabetical reference to the CSP functions.
- Chapter 3, "Data Structures" describes the data structures used by CSP functions.
- Chapter 4, "Events" provides an alphabetical reference to events that may be returned by the CSP software.
- Glossary provides a definition of terms used in this guide.

## Related Information

See the following for more information:

- For information about CSP library features and guidelines for building applications using CSP software, see the *Continuous Speech Processing API Programming Guide*.
- For information on the CSP demo, see the *Continuous Speech Processing API Demo Guide*.
- For information about Voice library features and guidelines for building applications using Voice software, see the *Voice API Programming Guide*.
- For details on all voice functions, parameters, and data structures in the Voice library, see the *Voice API Library Reference*.
- For details on the Standard Runtime Library (a device-independent library that consists of event management functions and standard attribute functions), supported programming models, and programming guidelines for building all applications, see the *Standard Runtime Library API Programming Guide*.
- For details on all functions and data structures in the Standard Runtime Library library, see the *Standard Runtime Library API Library Reference*.
- For information on the system release, system requirements, software and hardware features, supported hardware, and release documentation, see the Release Guide.
- For details on compatibility issues, restrictions and limitations, known problems, and late-breaking updates or corrections to the release documentation, see the Release Update (available on the Technical Support Web site only).
- For details on installing the system software, see the *System Release Installation Guide*.
- For guidelines on building applications using Global Call software (a common signaling interface for network-enabled applications, regardless of the signaling protocol needed to connect to the local telephone network), see the *Global Call API Programming Guide*.
- For details on all functions and data structures in the Global Call library, see the *Global Call API Library Reference*.
- For details on configuration files (including FCD/PCD files) and instructions for configuring products, see the Configuration Guide for your product or product family.
- For technical support, see *http://developer.intel.com/design/telecom/support/*. This Technical Support Web site contains developer support information, downloads, release documentation, technical notes, application notes, a user discussion forum, and more.

# Function Summary by Category 1

This chapter describes the categories into which the Continuous Speech Processing (CSP) library functions can be grouped:

## 1.1 Input/Output Functions

The following functions are used in the transfer of data to and from a CSP-capable channel:

**ec_rearm( )**
Re-enables the voice activity detector (VAD).

**ec_reciottdata( )**
Records echo-cancelled data to a file or memory buffer.

**ec_stopch( )**
Stops activity on a CSP-capable channel.

**ec_stream( )**
Streams echo-cancelled data to a callback function.

## 1.2 Configuration Functions

The following functions are used to configure a CSP-capable channel:

**ec_getparm( )**
Returns the current parameter settings on an open CSP-capable channel device.

**ec_setparm( )**
Configures the parameter of an open CSP-capable channel device.

## 1.3 Routing Functions

The following functions are used for SCbus or CT Bus routing:

**ec_getxmitslot( )**
Returns the time slot on which echo-cancelled data was transmitted.

**ec_listen( )**
Changes the echo-reference signal from the default reference (that is, the same channel as the play) to the specified time slot on the TDM bus.

**ec_unlisten( )**
Changes the echo-reference signal set by **ec_listen( )** back to the default reference (that is, the same channel as the play).

## 1.4 Extended Attribute Functions

The following function is used to obtain information about the device:

**ATEC_TERMMSK( )**
Returns the reason for the last **ec_stream( )** or **ec_reciottdata( )** function termination on a CSP-capable channel.

## 1.5 Status Information

The following function falls under the Status Information category:

**ec_getblkinfo( )**
Returns header information for a block of echo-cancelled data.

# Function Information 2

This chapter provides an alphabetical reference to the functions in the Continuous Speech Processing (CSP) library.

## 2.1 Function Syntax Conventions

The CSP functions use the following syntax:

```
int ec_function(device_handle, parameter1, ... parameterN)
```

where:

int
: refers to the data type integer.

ec_function
: represents the function name. All CSP-specific functions begin with “ec”.

device_handle
: represents the device handle, which is a numerical reference to a device, obtained when a device is opened. The device handle is used for all operations on that device.

parameter1
: represents the first parameter.

parameterN
: represents the last parameter.

## ATEC_TERMMSK( )

| | | |
|---|---|---|
| **Name:** | long ATEC_TERMMSK(chdev) | |
| **Inputs:** | int chdev | • valid channel device handle |
| **Returns:** | channel's last termination bitmap if successful<br>AT_FAILURE if error | |
| **Includes:** | srllib.h<br>dxxxlib.h<br>eclib.h | |
| **Category:** | Extended Attribute | |
| **Mode:** | Synchronous | |
| **Platform:** | DM3, Springware | |

### ■ Description

The **ATEC_TERMMSK( )** function returns a bitmap containing the reason(s) for the last **ec_stream( )** or **ec_reciottdata( )** function termination on the channel **chdev**. The bitmap is set when one of these functions terminates.

| Parameter | Description |
|---|---|
| **chdev** | the channel device handle obtained when the CSP-capable device is opened using **dx_open( )** |

A CSP streaming function terminates when one of the conditions set in the DV_TPT data structure occurs. For more information on this structure, see the *Voice API Library Reference*.

For a list of possible return values, see the **ATDX_TERMMSK( )** function description in the *Voice API Library Reference*.

### ■ Cautions

None.

### ■ Errors

This function will fail and return AT_FAILURE if an invalid channel device handle is specified in **chdev**.

### ■ Example

```
#include <windows.h> /* include in Windows applications only; exclude in Linux */
#include <stdio.h>
#include <fcntl.h>
#include <srllib.h>
#include <dxxxlib.h>
```

```
main()
{
     int chdev;
     long term;
     DX_IOTT iott;
     DV_TPT  tpt[3];
     static DX_XPB xpb = {
          FILE_FORMAT_VOX,
          DATA_FORMAT_PCM,
          DRT_8KHZ,
          8
     };



     /* Open the channel device */
     if ((chdev = dx_open("dxxxB1C1",NULL)) == -1) {
         /* Process error */
     }

     /* Record a voice file. Terminate on receiving a digit, silence,
      * max time, or reaching a byte count of 50000 bytes.
      */
     /* set up DX_IOTT */
     iott.io_type = IO_DEV|IO_EOT;
     iott.io_bufp = 0;
     iott.io_offset = 0;
     iott.io_length = 50000;

     if((iott.io_fhandle = open("file.vox", O_CREAT | O_RDWR, 0666)) == -1)  { (Linux only)
         /* process error */
     }

     if((iott.io_fhandle = dx_fileopen("file.vox", O_RDWR)) == -1)  { (Windows only)
         /* process error */
     }

     /* set up DV_TPTs for the required terminating conditions */
     dx_clrtpt(tpt,3);
     tpt[0].tp_type   = IO_CONT;
     tpt[0].tp_termno = DX_MAXDTMF;           /* Maximum digits */
     tpt[0].tp_length = 1;                     * terminate on the first digit */
     tpt[0].tp_flags  = TF_MAXDTMF;           /* Use the default flags */
     tpt[1].tp_type   = IO_CONT;
     tpt[1].tp_termno = DX_MAXTIME;           /* Maximum time */
     tpt[1].tp_length = 100;                  /* terminate after 10 secs */
     tpt[1].tp_flags  = TF_MAXTIME;           /* Use the default flags */
     tpt[2].tp_type   = IO_EOT;               /* last entry in the table */
     tpt[2].tp_termno = DX_MAXSIL;            /* Maximum Silence */
     tpt[2].tp_length = 30;                   /* terminate on 3 sec silence */
     tpt[2].tp_flags  = TF_MAXSIL;            /* Use the default flags */


     /* Now record to the file using ec_reciottdata() */
     if (ec_reciottdata(chdev, &iott, tpt, &xpb, EV_SYNC) == -1) {
           /* process error */
     }

     /* Examine bitmap to determine if digits caused termination */
     if((term = ATEC_TERMMSK(chdev)) == AT_FAILURE) {
         /* Process error */
     }
```

```
        if(term & TM_MAXDTMF) {
            printf("Terminated on digits\n");
              .
              .
        }
}
```

## ■ See Also

- **ATDX_TERMMSK( )** in the *Voice API Library Reference*

## ec_getblkinfo( )

|  |  |  |
|---|---|---|
| **Name:** | int ec_getblkinfo(blkInfop) | |
| **Inputs:** | EC_BLK_INFO *blkInfop | • points to the EC_BLK_INFO data structure |
| **Returns:** | 0 for success<br>-1 for failure | |
| **Includes:** | srllib.h<br>dxxxlib.h<br>eclib.h | |
| **Category:** | Status Information | |
| **Mode:** | Synchronous | |

### ■ Description

The **ec_getblkinfo( )** function returns header information for a block of echo-cancelled data.

This function is typically used in conjunction with the silence compressed streaming feature. To enable this feature, set ECCH_SILENCECOMPRESS in **ec_setparm( )**. This feature is disabled by default.

This function allows you to get information about a block of data and perform various analyses. For example, by looking at the time stamp information for each block, you can calculate the amount of silence that was compressed. Or you can retrieve the block of initial data.

| Parameter | Description |
|---|---|
| **blkInfop** | points to the EC_BLK_INFO data structure, which specifies block information. See EC_BLK_INFO, on page 58 for more information. |

Invoke user I/O callback functions in your application to receive unbuffered data blocks. You can do so using **ec_stream( )** after you have installed user I/O functions. Each block is delivered to the application *write* callback function as it is received from the board.

By invoking **ec_getblkinfo( )** from within the *write* callback function, the EC_BLK_INFO structure to which it passes a pointer is filled with details regarding the block for which the callback is executed.

### ■ Cautions

- You must call the **ec_getblkinfo( )** function from within the *write* callback function; otherwise, an error will occur.
- You cannot use **ec_getblkinfo( )** in conjunction with **ec_reciottdata( )** because the user application layer does not have access to the internal callback function.
- You should process your data before calling **ec_getblkinfo( )** otherwise bits of noise will be interjected into the silence compressed recording.

## ■ Errors

When this function returns -1 to indicate failure, the most likely reason is that this function was not called from within a *write* callback function.

## ■ Example

This example code illustrates how a callback function processes streamed data.

```
int stream_cb(int chDev, char *buffer, UINT length)
{
     int          rc;
     /* Silence Compressed Streaming*/
     EC_BLK_INFO *blkInfo;

     /* process data first before calling ec_getblkinfo() so as not to disrupt the transfer
      * of data
      */

     /* Write recorded streaming data to file. */
     rc = _write(RecordFile[ecdev_to_channel[chDev]], buffer, length);
     if (rc <0) {
          printf( "[%d]: Error writing streaming data to file.\n", ecdev_to_channel[chDev]);
     }


     if(ChanNum == 1){
          blkInfo = (EC_BLK_INFO *)buffer;

      /* Note: It is not recommended to use of printf(),  scanf(),
      *        and other similar functions within the callback function.
      *        For demonstration purposes output to the screen
      *        occurs when using a single channel.
      */

          printtime();
          printf("[%d]:In stream_cb(). Received %d bytes of data. Voice data is
                     processed.\n",ecdev_to_channel[chDev], length);
          if(compress == TRUE){

               /* Silence Compressed Streaming */
               rc = ec_getblkinfo(blkInfo);
               if (rc == 0){
                    printtime();
                    printf("[%d]: type %d: flags %d: size %d: elapsed time %d\n",
                    ecdev_to_channel[chDev],blkInfo->type,blkInfo->flags,blkInfo->size,
                          blkInfo->timestamp);
               }else{printf("Error in ec_getblkinfo(). Err Msg = %s, Lasterror = %d\n",
                    ATDV_ERRMSGP(chDev), ATDV_LASTERR(chDev));  }
          }
     }//end if ChanNum

     return(length);

}
```

## ■ See Also

- **ec_reciottdata( )**
- **ec_stream( )**
- EC_BLK_INFO data structure

## ec_getparm( )

| | | |
|---|---|---|
| **Name:** | int ec_getparm(chdev, parmNo, lpValue) | |
| **Inputs:** | int chdev | • valid channel device handle |
| | unsigned long parmNo | • parameter value |
| | void *lpValue | • pointer to memory where parameter value is stored |
| **Returns:** | 0 for success<br>-1 for failure | |
| **Includes:** | srllib.h<br>dxxxlib.h<br>eclib.h | |
| **Category:** | Configuration | |
| **Mode:** | Synchronous | |
| **Platform:** | DM3, Springware | |

### ■ Description

The **ec_getparm( )** function returns the current parameter settings for an open device that supports continuous speech processing (CSP).

| Parameter | Description |
|---|---|
| **chdev** | the channel device handle obtained when the CSP-capable device is opened using **dx_open( )** |
| **parmNo** | the define for the parameter whose value is returned in the variable pointed to by **lpValue** |
| **lpValue** | a pointer to the variable where the **parmNo** value is stored on return |

The same parameter IDs are available for **ec_setparm( )** and **ec_getparm( )**. For details on these parameters, see the **ec_setparm( )** function description.

### ■ Cautions

- The address of the variable passed to receive the value of the requested parameter must be cast as void* as shown in the example. It is recommended that you clear this variable prior to calling **ec_getparm( )**.
- Allocate sufficient memory to receive the value of the parameter specified. Note that some parameters require only two bytes while other parameters may be ASCII strings. The data type of the variable that will receive the parameter value must match the data type for the specific parameter being queried.
- On DM3 boards, you must issue **ec_getparm( )** in the same process as **ec_setparm( )**; otherwise, the values returned for **ec_getparm( )** will be invalid.

## ■ Errors

If the function returns -1, use **ATDV_LASTERR( )** to return the error code and **ATDV_ERRMSGP( )** to return a descriptive error message.

One of the following error codes may be returned:

EDX_BADDEV
: Device handle is NULL or invalid.

EDX_BADPARM
: Parameter error.

EDX_BUSY
: Channel is busy (when channel device handle is specified) or first channel is busy (when board device handle is specified).

EDX_SYSTEM
: Operating system error.

EEC_UNSUPPORTED
: Device handle is valid but device does not support CSP.

## ■ Example

```
#include <windows.h>  /* include in Windows applications only; exclude in Linux */
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <eclib.h>
#include <errno.h> /* include in Linux applications only; exclude in Windows */

main()
{

int chdev
short parmval; /* DXCH_BARGEIN parameter is 2 bytes long */
int srlmode;   /* Standard Runtime Library mode */

     /* Set SRL to run in polled mode. */
     srlmode = SR_POLLMODE;
     if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1) {
         /* process error */
     }

     /* Open the channel and get channel device handle in chdev */
     if ((chdev = dx_open("dxxxB1C1",0)) == -1) {
          /* process error */
     }

     /* Clear parameter variable */
     parmval = 0;

     /* Get parameter settings */
     if (ec_getparm(chdev, DXCH_BARGEIN, (void *)&parmval) == -1) {
          /* process error */
     }
     /* Get additional parameter settings as needed */
  . . .
}
```

## ■ See Also

- **ec_setparm( )**
- **dx_setparm( )** in the *Voice API Library Reference*

## ec_getxmitslot( )

| | | |
|---|---|---|
| **Name:** | int ec_getxmitslot(chDev, lpSlot) | |
| **Inputs:** | int chDev | • valid channel device handle |
| | SC_TSINFO *lpSlot | • pointer to time slot data structure |
| **Returns:** | 0 for success<br>-1 for failure | |
| **Includes:** | srllib.h<br>dxxxlib.h<br>eclib.h | |
| **Category:** | Routing | |
| **Mode:** | Synchronous | |
| **Platform:** | DM3, Springware | |

### ■ Description

The **ec_getxmitslot( )** function returns the echo-cancelled transmit time slot number of a CSP-capable full-duplex channel. It returns the number of the TDM bus time slot which transmits the echo-cancelled data. This information is returned in an SC_TSINFO structure.

This function (together with xx_**listen( )** and xx_**unlisten( )** routing functions such as **dx_getxmitslot( )** and **gc_GetXmitSlot( )**) allows you to send echo-cancelled data over the TDM bus to another board, regardless of whether this board is CSP-capable or not.

*Note:* The **ec_getxmitslot( )** function supports the feature streaming echo-cancelled data to the TDM bus. This feature is supported on select DM3 boards only. For current information on hardware support, see the Release Guide for the system release you are using.

The SC_TSINFO structure is declared as follows:

```
typedef struct {
        unsigned long  sc_numts;
        long           *sc_tsarrayp;
} SC_TSINFO;
```

The **sc_numts** field must be initialized with the number of TDM bus time slots requested (1 for a voice channel). The **sc_tsarrayp** field must be initialized with a pointer to a valid array. Upon return from the function, the array contains the time slot on which the voice channel transmits.

| Parameter | Description |
|---|---|
| **chDev** | the channel device handle obtained when the CSP-capable device is opened using **dx_open( )** |
| **lpSlot** | a pointer to the SC_TSINFO data structure |

## Cautions

- This function fails if you specify an invalid channel device handle.
- The **nr_scroute( )** and **nr_scunroute( )** convenience functions do not support CSP. To route echo-cancelled data, use xx_**listen( )** and xx_**unlisten( )** functions where xx represents the type of device, such as "dx" for voice. See the *Voice API Library Reference* for more information.
- On Springware boards, in Linux applications that use multiple threads, you must avoid having two or more threads call functions that use the same device handle; otherwise, the replies can be unpredictable and cause those functions to time out or create internal concurrency that can lead to a segmentation fault. If you must do this, use semaphores to prevent concurrent access to a particular device handle.

## Errors

If the function returns -1, use **ATDV_LASTERR( )** to return the error code and **ATDV_ERRMSGP( )** to return a descriptive error message.

One of the following error codes may be returned:

| | |
|---|---|
| EDX_BADDEV | Device handle is NULL or invalid. |
| EDX_BADPARM | Time slot pointer information is NULL or invalid. |
| EDX_SH_BADCMD | Command is not supported in current bus configuration. |
| EDX_SH_BADINDX | Invalid Switch Handler index number. |
| EDX_SH_BADLCLTS | Invalid channel number. |
| EDX_SH_BADMODE | Function is not supported in current bus configuration. |
| EDX_SH_BADTYPE | Invalid channel type (voice, analog, etc.) |
| EDX_SH_CMDBLOCK | Blocking command is in progress. |
| EDX_SH_LCLDSCNCT | Channel is already disconnected from TDM bus. |
| EDX_SH_LIBBSY | Switch Handler library is busy. |
| EDX_SH_LIBNOTINIT | Switch Handler library is uninitialized. |
| EDX_SH_MISSING | Switch Handler is not present. |

EDX_SH_NOCLK
: Switch Handler clock fallback failed.

EDX_SYSTEM
: Operating system error.

EEC_UNSUPPORTED
: Device handle is valid but device does not support CSP.

## Example

```
#include <windows.h>  /* include in Windows applications only; exclude in Linux */
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <eclib.h>
#include <errno.h>  /* include in Linux applications only; exclude in Windows */

main()
{
     int chdev;                  /* Channel device handle */
     SC_TSINFO sc_tsinfo;        /* Time slot information structure */
     long scts;                  /* TDM bus time slot */
     int srlmode;                /* Standard Runtime Library mode */

    /* Set SRL to run in polled mode. */
     srlmode = SR_POLLMODE;
     if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1) {
          /* process error */
     }

    /* Open board 1 channel 1 device */
     if ((chdev = dx_open("dxxxB1C1", 0)) == -1) {
          printf("Cannot open channel dxxxB1C1. Check to see if board is started");
          exit(1);
     }

    /* Fill in the TDM bus time slot information */
     sc_tsinfo.sc_numts = 1;
     sc_tsinfo.sc_tsarrayp = &scts;

    /* Get TDM bus time slot connected to transmit of voice channel 1 on board 1 */
     if (ec_getxmitslot(chdev, &sc_tsinfo) == -1) {
          printf("Error message = %s", ATDV_ERRMSGP(chdev));
          exit(1);
     }

     printf("%s is transmitting on TDM bus time slot %ld", ATDV_NAMEP(chdev), scts);
}
```

## See Also

- **ag_getxmitslot( )** in the *Voice API Library Reference*
- **dt_getxmitslot( )** in the *Digital Network Interface Software Reference*
- **dx_getxmitslot( )** in the *Voice API Library Reference*

# ec_listen( )

| | | |
|---|---|---|
| **Name:** | int ec_listen(chDev, lpSlot) | |
| **Inputs:** | int chDev | • valid channel device handle |
| | SC_TSINFO *lpSlot | • pointer to time slot data structure |
| **Returns:** | 0 for success<br>-1 for failure | |
| **Includes:** | srllib.h<br>dxxxlib.h<br>eclib.h | |
| **Category:** | Routing | |
| **Mode:** | Synchronous | |
| **Platform:** | DM3, Springware | |

## ■ Description

The **ec_listen( )** function changes the echo-reference signal from the default reference (that is, the same channel as the play) to the specified time slot on the TDM bus.

The SC_TSINFO structure is declared as follows:

```
typedef struct {
        unsigned long  sc_numts;
        long           *sc_tsarrayp;
} SC_TSINFO;
```

The **sc_numts** field must be initialized with a value of 1. The **sc_tsarrayp** field must be initialized with a pointer to a long value that holds the time slot number that should be used as reference signal by the CSP-capable channel. Note that an SC_TSINFO structure populated by one of the xx_getxmitslot( ) functions (such as **dx_getxmitslot( )**, **dt_getxmitslot( )**, and **gc_GetXmitSlot( )**) is correctly formatted for use with **ec_listen( )**.

| Parameter | Description |
|---|---|
| **chDev** | the channel device handle obtained when the CSP-capable device is opened using **dx_open( )** |
| **lpSlot** | a pointer to the SC_TSINFO data structure |

## Cautions

- This function fails if you specify an invalid channel device handle.
- On Springware boards, in Linux applications that use multiple threads, you must avoid having two or more threads call functions that use the same device handle; otherwise, the replies can be unpredictable and cause those functions to time out or create internal concurrency that can lead to a segmentation fault. If you must do this, use semaphores to prevent concurrent access to a particular device handle.

## Errors

If the function returns -1, use **ATDV_LASTERR( )** to return the error code and **ATDV_ERRMSGP( )** to return a descriptive error message.

One of the following error codes may be returned:

EDX_BADDEV
: Device handle is NULL or invalid.

EDX_BADPARM
: Time slot pointer information is NULL or invalid.

EEC_UNSUPPORTED
: Device handle is valid but device does not support CSP.

## Example

```
#include <windows.h>  /* include in Windows applications only; exclude in Linux */
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <eclib.h>
#include <errno.h>  /* include in Linux applications only; exclude in Windows */

main()
{

     int chdev1,chdev2;   /* Channel device handles */
     SC_TSINFO sc_tsinfo; /* Time slot information structure */
     long  scts;          /* TDM bus time slot */

     /* Open board 1 channel 1 device */
     chdev1 = dx_open("dxxxB1C1", NULL);
     if (chdev1 < 0) {
          printf("Error %d in dx_open(dxxxB1C1)\n",chdev1);
          exit(-1);
     }

     /* Open board 1 channel 2 device */
     chdev2 = dx_open("dxxxB1C2", NULL);
     if (chdev2 < 0) {
          printf("Error %d in dx_open(dxxxB1C1)\n",chdev2);
          exit(-1);
     }

     /* Set DXCH_EC_TAP_LENGTH as needed */
     ret = ec_setparm( ... );
```

```
        /* Get second channel’s vox transmit time slot */
        sc_tsinfo.sc_numts = 1;
        sc_tsinfo.sc_tsarrayp = &scts;
        if (dx_getxmitslot(chdev2, &sc_tsinfo) == -1) {
             printf("Error in dx_getxmitslot(chdev2, &sc_tsinfo). Err Msg =  %s\n",
                       ATDV_ERRMSGP(chdev2));
        }

        /* Make first channel's ec listen to second channel's vox transmit time slot */
        if (ec_listen(chdev1, &sc_tsinfo) == -1) {
             printf("Error in ec_listen(chdev1, &sc_tsinfo). Err Msg = %s\n",
                       ATDV_ERRMSGP(chdev1));
        }

        /* Set ECCH_NLP off and set other desired parameters */
        ret = ec_setparm( ... );
   }
```

## ■ See Also

- **ag_listen( )** in the *Voice API Library Reference*
- **dt_listen( )** in the *Digital Network Interface Software Reference*
- **dx_listen( )** in the *Voice API Library Reference*

## ec_rearm( )

| | | |
|---|---|---|
| **Name:** | int ec_rearm(chDev) | |
| **Inputs:** | int chDev | • valid channel device handle |
| **Returns:** | 0 for success<br>-1 for failure | |
| **Includes:** | srllib.h<br>dxxxlib.h<br>eclib.h | |
| **Category:** | I/O | |
| **Mode:** | Synchronous | |
| **Platform:** | DM3, Springware | |

### ■ Description

The **ec_rearm( )** function temporarily stops streaming of echo-cancelled data from the board and rearms or re-enables the voice activity detector (VAD). The prompt is not affected by this function.

If a VAD event is received and the recognizer determines that the energy was non-speech such as a cough, use this function to re-activate the VAD for the next burst of energy.

The **ec_rearm( )** function is intended to be used with VAD enabled and barge-in disabled.

***Note:*** In System Release 6.0, the **ec_rearm( )** function is not supported. The rearming functionality is superseded by the silence compressed streaming feature. Use the ECCH_SILENCECOMPRESS parameter in **ec_setparm( )** to enable this feature.

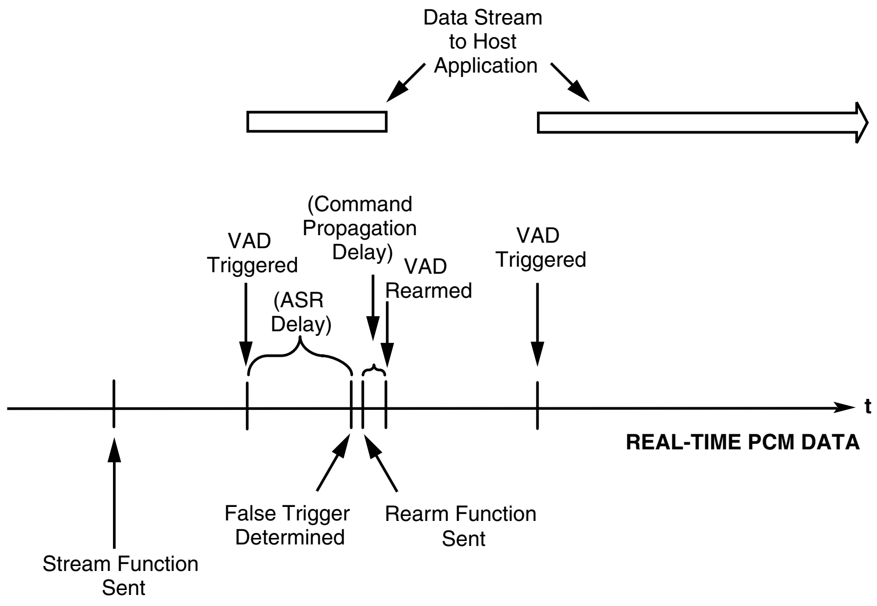
| Parameter | Description |
|---|---|
| **chDev** | the channel device handle obtained when the CSP-capable device is opened using **dx_open( )** |

The following scenario describes how the **ec_rearm( )** function works. After the VAD is triggered, it starts streaming data to the host application. The host application determines that this is a false trigger and calls the **ec_rearm( )** function. Streaming is halted and the VAD is rearmed for the next burst of energy.

***Caution:*** During the time that the VAD is being rearmed, you will not get a VAD event if an energy burst comes in. The time it takes for the VAD to be rearmed varies depending on hardware and operating system used.

Figure 1 illustrates the rearming concept.

**Figure 1. Rearming the Voice Activity Detector (VAD)**



## Cautions

You must explicitly clear the digit buffer if digits remaining in the buffer will not be processed. Before calling **ec_rearm**( ), you must call **dx_clrdigbuf**( ) to clear the buffer. For more information on this function, see the *Voice API Library Reference*.

## Errors

If the function returns -1, use **ATDV_LASTERR**( ) to return the error code and **ATDV_ERRMSGP**( ) to return a descriptive error message.

One of the following error codes may be returned:

EDX_BADDEV
Device handle is NULL or invalid.

EDX_BADPARM
Parameter error.

EEC_UNSUPPORTED
Device handle is valid but device does not support CSP.

## Example

```
#include <windows.h>  /* include in Windows applications only; exclude in Linux */
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <eclib.h>
#include <errno.h>  /* include in Linux applications only; exclude in Windows */
```

```
main()
{

   char  csp_devname[9];
   int   ret, csp_dev, parmval=0;
   SC_TSINFO  sc_tsinfo;   /* Time slot information structure */
   long scts;              /* TDM bus time slot */
   int srlmode;            /* Standard Runtime Library mode */
   DX_IOTT iott;           /* I/O transfer table */
   DV_TPT  tptp[1], tpt;   /* termination parameter table */
   DX_XPB xpb;             /* I/O transfer parameter block */

  /* Set SRL to run in polled mode. */
   srlmode = SR_POLLMODE;
   if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1) {
      /* process error */
   }

   sprintf(csp_devname,"dxxxB1C1");

  /* Open a voice device. */
   csp_dev = dx_open(csp_devname, 0);
   if (csp_dev < 0) {
      printf("Error %d in dx_open()\n",csp_dev);
      exit(-1);
   }

  /* Set up ec parameters as needed.
   * ECCH_VADINITIATED is enabled by default.
   * Barge-in should be disabled (DXCH_BARGEIN=0) so that prompt
   * continues to play after energy is detected.
   */
   ret = ec_setparm( ... );
   if (ret == -1) {
      printf("Error in ec_setparm(). Err Msg = %s, Lasterror = %d\n",
      ATDV_ERRMSGP(csp_dev), ATDV_LASTERR(csp_dev));
   }

  /* Set up DV_TPT for record */
   tpt.tp_type   = IO_EOT;
   tpt.tp_termno = DX_MAXTIME;
   tpt.tp_length = 60;
   tpt.tp_flags  = TF_MAXTIME;

  /* Record data format set to 8-bit PCM, 8KHz sampling rate */
   xpb.wFileFormat = FILE_FORMAT_VOX;
   xpb.wDataFormat = DATA_FORMAT_PCM;
   xpb.nSamplesPerSec = DRT_8KHZ;
   xpb.wBitsPerSample = 8;

   ret = ec_stream(csp_dev, &tpt, &xpb, &stream_cb, EV_ASYNC | MD_NOGAIN);
   if (ret == -1) {
      printf("Error in ec_reciottdata(). Err Msg = %s, Lasterror = %d\n",
      ATDV_ERRMSGP(csp_dev), ATDV_LASTERR(csp_dev));
   }

  /* Set channel off-hook */
   ret = dx_sethook(csp_dev, DX_OFFHOOK, EV_SYNC);
   if (ret == -1) {
      printf("Error in dx_sethook(). Err Msg = %s, Lasterror = %d\n",
      ATDV_ERRMSGP(csp_dev), ATDV_LASTERR(csp_dev));
   }
```

```
/* Set up DX_IOTT */
 iott.io_type   = IO_DEV|IO_EOT;
 iott.io_bufp   = 0;
 iott.io_offset = 0;
 iott.io_length = -1;

/* Set up DV_TPT for play */
 dx_clrtpt(&tptp,1);
 tptp[0].tp_type   = IO_EOT;
 tptp[0].tp_termno = DX_MAXDTMF;
 tptp[0].tp_length = 1;
 tptp[0].tp_flags  = TF_MAXDTMF;

/* Open file to be played */
 #ifdef WIN32
 if ((iott.io_fhandle = dx_fileopen("sample.vox",O_RDONLY|O_BINARY)) == -1) {
    printf("Error opening sample.vox.\n");
    exit(1);
 }
 #else
 if (( iott.io_fhandle = open("sample.vox",O_RDONLY)) == -1) {
    printf("File open error\n");
    exit(2);
 }
 #endif

/* Play prompt message. */
 ret = dx_play(csp_dev, &iott, &tptp, EV_ASYNC);
 if ( ret == -1) {
    printf("Error playing sample.vox.\n");
    exit(1);
 }

/* In the ASR engine section -- pseudocode */
   while (utterance is undesirable) {
   /* Wait for TEC_VAD event */
    while (TEC_VAD event is not received) {
       sr_waitevt(-1);
       ret = sr_getevttype( );
       if (ret == TEC_VAD) {
         /* After TEC_VAD event is received, determine if utterance is desirable */
          if (utterance is undesirable) {
            /* Use ec_rearm( ) to pause streaming and rearm the VAD trigger*/
             ret = ec_rearm(csp_dev);
             if (ret == -1) {
                printf("Error in ec_rearm(). Err Msg = %s, Lasterror = %d\n",
                ATDV_ERRMSGP(csp_dev), ATDV_LASTERR(csp_dev));
             } /* end if (ret == -1) */
          } /* end if (utterance is undesirable) */
       } /* end if (ret == TEC_VAD) */
    } /* end while (TEC_VAD event not received)
 } /* end while (utterance is undesirable) */

.
.
.
```

## ■ See Also

None

## ec_reciottdata( )

| | | |
|---|---|---|
| **Name:** | int ec_reciottdata(chDev, iottp, tptp, xpbp, mode) | |
| **Inputs:** | int chDev | • valid channel device handle |
| | DX_IOTT *iottp | • pointer to I/O transfer table |
| | DV_TPT *tptp | • pointer to termination parameter table |
| | DX_XPB *xpbp | • pointer to I/O transfer parameter block table |
| | unsigned short mode | • record mode |
| **Returns:** | 0 for success<br>-1 for failure | |
| **Includes:** | srllib.h<br>dxxxlib.h<br>eclib.h | |
| **Category:** | I/O | |
| **Mode:** | Synchronous/Asynchronous | |
| **Platform:** | DM3, Springware | |

### ■ Description

The **ec_reciottdata( )** function starts an echo-cancelled record to a file or memory buffer on a CSP-capable full-duplex channel.

You can perform a record at all times or a voice-activated record. To perform a voice-activated record, where recording begins only after speech energy has been detected, use ECCH_VADINITIATED in **ec_setparm( )**. This parameter is enabled by default.

You can record or stream speech energy with silence removed from the data stream. To use the silence compressed streaming feature, enable ECCH_SILENCECOMPRESS in **ec_setparm( )**. This parameter is disabled by default.

| Parameter | Description |
|---|---|
| **chDev** | the channel device handle obtained when the CSP-capable device is opened using **dx_open( )** |
| **iottp** | pointer to the DX_IOTT data structure that specifies the order and media on which the echo-cancelled data is recorded |

| Parameter | Description |
|---|---|
| **tptp** | pointer to the DV_TPT data structure that sets the termination conditions for the device handle<br>***Note:*** On Springware boards, the only supported DV_TPT terminating conditions are DX_MAXTIME, DX_MAXSIL, and DX_MAXNOSIL. The Voice Activity Detector (VAD) timeout period is set by the **tp_length** parameter in the DV_TPT structure. For more information on DV_TPT, see the *Voice API Library Reference*.<br>***Note:*** On DM3 boards, all DV_TPT terminating conditions are supported except for DX_LCOFF, DX_PMON and DX_PMOFF.<br>***Note:*** In CSP, DV_TPT terminating conditions are edge-sensitive. |
| **xpbp** | pointer to the DX_XPB data structure that specifies the file format, data format, sampling rate and sampling size |
| **mode** | a logical OR bit mask that specifies the record mode:<br>• EV_SYNC – synchronous mode<br>• EV_ASYNC – asynchronous mode<br>• MD_GAIN – automatic gain control (AGC)<br>• MD_NOGAIN – no automatic gain control<br>***Note:*** For ASR applications, turn AGC off. |

## ■ Cautions

- This function fails if an unsupported data format is specified. For a list of supported data formats, see the *Continuous Speech Processing API Programming Guide*.
- To set the proper parameters, the **ec_setparm( )** function must be called for every **ec_reciottdata**( ) occurrence in your application.
- If you use this function in synchronous mode, you must use multithreading in your application.
- All files specified in the DX_IOTT data structure are of the file format described in DX_XPB.
- All files recorded to have the data encoding and rate as described in DX_XPB.
- The DX_IOTT data area must remain in scope for the duration of the function if running asynchronously.
- The DX_XPB data area must remain in scope for the duration of the function if running asynchronously.
- On Springware boards, we recommend that you use the same data format for play and recording/streaming.
- On Springware boards, in Linux applications that use multiple threads, you must avoid having two or more threads call functions that use the same device handle; otherwise, the replies can be unpredictable and cause those functions to time out or create internal concurrency that can lead to a segmentation fault. If you must do this, use semaphores to prevent concurrent access to a particular device handle.
- The DX_MAXSIL and DX_MAXNOSIL termination conditions are not supported on CSP-enabled boards that have an analog front-end and Japan Caller ID enabled with polarity reversal. This limitation affects D/120JCT Rev 2 and D/41JCT-LS boards. Thus, these termination conditions cannot be specified in the DV_TPT structure for **ec_reciottdata**( ) and **ec_stream**( ).

- When DX_MAXSIL and DX_MAXNOSIL termination conditions are used, the silence timer starts after a **dx_play( )** has completed. To start the silence timer immediately after the onset of **ec_stream**( ) or **ec_reciottdata( )**, use the bit flag TF_IMMEDIATE (tp_flags field in the DV_TPT structure). This flag is not supported on Springware boards.

## ■ Errors

If the function returns -1, use **ATDV_LASTERR( )** to return the error code and **ATDV_ERRMSGP( )** to return a descriptive error message.

One of the following error codes may be returned:

EDX_BADDEV
Device handle is NULL or invalid.

EDX_BADIOTT
Invalid DX_IOTT setting.

EDX_BADPARM
Parameter is invalid or not supported.

EDX_BADWAVFILE
Invalid WAVE file.

EDX_BUSY
Channel is busy.

EDX_SH_BADCMD
Unsupported command or WAVE file format.

EDX_SYSTEM
Operating system error.

EDX_XPBPARM
Invalid DX_XPB setting.

EEC_UNSUPPORTED
Device handle is valid but device does not support CSP.

## ■ Example

```
#include <windows.h>  /* include in Windows applications only; exclude in Linux */
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <eclib.h>
#include <errno.h>  /* include in Linux applications only; exclude in Windows */

main ()
{

     int chdev;             /* channel descriptor */
     int fd;                /* file descriptor for file to be played */
     DX_IOTT iott, iott1;   /* I/O transfer table */
     DV_TPT tpt, tpt1;      /* termination parameter table */
     DX_XPB xpb;            /* I/O transfer parameter block */
     int parmval = 1;
     int srlmode;           /* Standard Runtime Library mode */
```

```
/* Set SRL to run in polled mode. */
srlmode = SR_POLLMODE;
if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1)
{
     /* process error */
}
.
.
.

/* Open channel */
if ((chdev = dx_open("dxxxB1C1",0)) == -1)
{
     /* process error */
}

/* Set event mask to send the VAD events to the application */
dx_setevtmsk(chdev, DM_VADEVTS);

/* To use barge-in and voice-activated recording, you must enable
 * ECCH_VADINITIATED (enabled by default) and DXCH_BARGEIN using
 * ec_setparm( )
 */
ec_setparm (chdev, DXCH_BARGEIN, &parmval);
ec_setparm (chdev, ECCH_VADINITIATED, &parmval);
.
.
.

/* Set up DV_TPT for record */
tpt.tp_type   = IO_EOT;
tpt.tp_termno = DX_MAXTIME;
tpt.tp_length = 60;  /* max time for record is 6 secs */
tpt.tp_flags  = TF_MAXTIME;

/* Open file */
#ifdef WIN32
if (( iott.io_fhandle = dx_fileopen("MESSAGE.VOX",O_RDWR| O_BINARY| O_CREAT| O_TRUNC,
                         0666)) == -1)
{
     printf("File open error\n");
     exit(2);
}
#else
if (( iott.io_fhandle = open("MESSAGE.VOX",O_RDWR| O_CREAT| O_TRUNC, 0666)) == -1)
{
     printf("File open error\n");
     exit(2);
}
#endif

/* Set up DX_IOTT */
iott.io_bufp    = 0;
iott.io_offset  = 0;
iott.io_length  = -1;
iott.io_type = IO_DEV | IO_EOT;

/*
 * Specify VOX file format for PCM at 8KHz.
 */
xpb.wFileFormat = FILE_FORMAT_VOX;
xpb.wDataFormat = DATA_FORMAT_PCM;
xpb.nSamplesPerSec = DRT_8KHZ;
xpb.wBitsPerSample = 8;
```

```
/* Start recording. */
if (ec_reciottdata(chdev, &iott, &tpt, &xpb, EV_ASYNC | MD_NOGAIN) == -1)
{
     printf("Error recording file - %s\n",    ATDV_ERRMSGP(chdev));
     exit(4);
}

/* Open file to be played */
#ifdef WIN32
if ((iott1.io_fhandle = dx_fileopen("SAMPLE.VOX",O_RDONLY|O_BINARY)) == -1)
{
     printf("Error opening SAMPLE.VOX\n");
     exit(1);
}
#else
if (( iott1.io_fhandle = open("SAMPLE.VOX",O_RDONLY)) == -1)
{
     printf("File open error\n");
     exit(2);
}
#endif

iott1.io_bufp    = 0;
iott1.io_offset  = 0;
iott1.io_length  = -1;
iott1.io_type = IO_DEV | IO_EOT;

/* Set up DV_TPT for play */
tpt1.tp_type   = IO_EOT;
tpt1.tp_termno = DX_MAXDTMF;
tpt1.tp_length = 1;
tpt1.tp_flags  = TF_MAXDTMF;

/* Play intro message; use same file format as record */
if (dx_playiottdata(chdev,&iott1,&tpt1,&xpb,EV_ASYNC) == -1)
{
     printf("Error playing file - %s\n",      ATDV_ERRMSGP(chdev));
     exit(4);
}

/* Wait for barge-in and echo-cancelled record to complete */
while (1)
{
     sr_waitevt(-1);
     ret = sr_getevttype();
     if (ret == TDX_BARGEIN)
     {
          printf("TDX_BARGEIN event received\n");
     }
     else if (ret == TDX_PLAY)
     {
          printf("Play Completed event received\n");
          break;
     }
     else if (ret == TEC_STREAM)
     {
          printf("TEC_STREAM - termination event");
          printf("for ec_stream and ec_reciottdata received\n");
          break;
     }
     else if (ret == TDX_ERROR)
     {
          printf("ERROR event received\n");
```

```
            }
            else
            {
                 printf("Event 0x%x received.\n", ret);
            }

       } /* end while */

       /* Close record file */
       #ifdef WIN32
       if (dx_fileclose(iott.io_fhandle) == -1)
       {
            printf("Error closing MESSAGE.VOX \n");
            exit(1);
       }
       #else
       if (close(iott.io_fhandle) == -1)
       {
            printf("Error closing MESSAGE.VOX \n");
            exit(1);
       }
       #endif

       /* Close play file */
       #ifdef WIN32
       if (dx_fileclose(iott1.io_fhandle) == -1
       {
            printf("Error closing SAMPLE.VOX \n");
            exit(1);
       }
       #else
       if (close(iott1.io_fhandle) == -1) {
            printf("Error closing SAMPLE.VOX \n");
            exit(1);
       }
       #endif

       /* Close channel */
       dx_close(chdev);

  }
```

## ■ See Also

- **ec_stream( )**
- **dx_reciottdata( )** in the *Voice API Library Reference*

## ec_setparm( )

| | | |
|---|---|---|
| **Name:** | int ec_setparm(chDev, parmNo, lpValue) | |
| **Inputs:** | int chDev | • valid channel device handle |
| | unsigned long parmNo | • parameter value |
| | void *lpValue | • pointer to memory where parameter value is stored |
| **Returns:** | 0 for success<br>-1 for failure | |
| **Includes:** | srllib.h<br>dxxxlib.h<br>eclib.h | |
| **Category:** | Configuration | |
| **Mode:** | Synchronous | |
| **Platform:** | DM3, Springware | |

### ■ Description

The **ec_setparm( )** function configures the parameter of an open device that supports continuous speech processing (CSP). This function sets one parameter value at a time on an open channel.

| Parameter | Description |
|---|---|
| **chDev** | the channel device handle obtained when a CSP-capable device is opened using **dx_open( )** |
| **parmNo** | the define for the parameter to be set |
| **lpValue** | a pointer to the variable that specifies the parameter to be set<br>***Note:*** You must pass the value of the parameter to be set in a variable cast as (void *) as shown in the example. |

The same parameter IDs are available for **ec_setparm( )** and **ec_getparm( )**.

The *eclib.h* contains definitions (#define) for these parameter IDs. All **ec_setparm( )** parameter IDs have default values. If you don't use **ec_setparm( )** to change the parameter values, the default values are used.

The following summarizes the parameter IDs and their purpose. The defines for parameter IDs are described in more detail following this table, in alphabetical order.

| **Board Level Parameters:** | |
|---|---|
| DXBD_RXBUFSIZE | • sets size of firmware record buffer |
| DXBD_TXBUFSIZE | • sets size of firmware play buffer |
| **Continuous Speech Processing Channel Parameters:** | |
| ECCH_XFERBUFFERSIZE | • sets size of driver record buffer |
| ECCH_VADINITIATED | • sets VAD-initiated calls |
| ECCH_ECHOCANCELLER | • enables/disables echo canceller |
| DXCH_BARGEIN | • enables barge-in during prompt play |
| DXCH_BARGEINONLY | • enables TDX_BARGEIN and TDX_PLAY events |
| DXCH_EC_TAP_LENGTH | • sets tap length of echo canceller |
| ECCH_ADAPTMODE | • sets adaptation mode (slow or fast convergence) |
| ECCH_NLP | • turns NLP (comfort noise) on or off |
| ECCH_SILENCECOMPRESS | • turn silence compressed streaming (SCS) on or off |
| **Voice Activity Detector (VAD) Parameters:** | |
| ECCH_SVAD | • enables/disables zero-crossing mode |
| DXCH_SPEECHPLAYTHRESH | • energy level that triggers VAD during play |
| DXCH_SPEECHNONPLAYTHRESH | • energy level that triggers VAD during non-play |
| DXCH_SPEECHPLAYTRIGG | • energy level greater than speech threshold that triggers VAD during play |
| DXCH_SPEECHNONPLAYTRIGG | • energy level greater than speech threshold that triggers VAD during non-play |
| DXCH_SPEECHPLAYWINDOW | • window surveyed to detect speech energy during prompt play |
| DXCH_SPEECHNONPLAYWINDOW | • window surveyed to detect speech energy during non-play |
| DXCH_SPEECHSNR | • reciprocal signal to noise ratio |

| Define | Description |
|---|---|
| DXBD_RXBUFSIZE (Springware boards only)<br>Bytes: 2<br>Default: 512<br>Attribute: R/W<br>Units: bytes<br>Range: 128-512 | Supported on Springware boards only. These buffers are used to transfer data between the firmware and the driver.<br>For more information on setting buffer sizes, see the *Continuous Speech Processing API Programming Guide*.<br>**Note:** Decreasing the size of the buffers increases the number of interrupts between the host application and the board, thereby increasing the load on both the host and on-board control processors.<br>**Note:** On Windows, to modify the default value of 512, you must edit the *voice.prm* file. For details, see the configuration guide. On Linux, the value is fixed at 512. |
| DXBD_TXBUFSIZE (Springware boards only)<br>Bytes: 2<br>Default: 512<br>Attribute: R/W<br>Units: bytes<br>Range: 128-512 | Supported on Springware boards only. Sets the size of the firmware transmit (or play) buffers in shared RAM. These buffers are used to transfer data between the firmware and the driver.<br>Be sure that all channels on the board are idle before using this parameter; otherwise, unpredictable behavior may result.<br>For more information on setting buffer sizes, see the *Continuous Speech Processing API Programming Guide*.<br>**Note:** Decreasing the size of the buffers increases the number of interrupts between the host application and the board, thereby increasing the load on both the host and on-board control processors.<br>**Note:** On Windows, to modify the default value of 512, you must edit the *voice.prm* file. For details, see the configuration guide. On Linux, the value is fixed at 512. |
| DXCH_BARGEIN<br>Bytes: 2<br>Default: 0<br>Attribute: R/W<br>Values: 0 or 1 | Enables or disables barge-in in the application during prompt play when a CSP-supported data format is used. For a list of supported data formats, see the *Continuous Speech Processing API Programming Guide*.<br>The value 1 turns the feature on, and 0 turns the feature off. |
| DXCH_BARGEINONLY<br>Bytes: 2<br>Default: 1<br>Attribute: R/W<br>Values: 0 or 1 | Enables or disables generation of TDX_BARGEIN and TDX_PLAY events when a barge-in condition occurs. See Chapter 4, "Events" for a list of events.<br>The value 0 enables generation of both TDX_BARGEIN and TDX_PLAY events. (In doing so, you receive the TDX_PLAY event upon barge-in and can simply ignore the TDX_BARGEIN event in your playback state machine.)<br>**Note:** When playing a prompt in synchronous mode, you must set DXCH_BARGEINONLY to 0.<br>The value 1, the default, enables generation of TDX_BARGEIN event only.<br>This parameter does not affect the setting of barge-in itself; see DXCH_BARGEIN. |

| Define | Description |
|---|---|
| DXCH_EC_TAP_LENGTH (for Springware boards)<br>Bytes: 2<br>Default: 48<br>Attribute: R/W<br>Units: 0.125 ms<br>Values: 48 to 128 | For Springware boards. Specifies the tap length for the echo canceller. The longer the tap length, the more echo is cancelled from the incoming signal. However, this means more processing power is required.<br>The default value is 48 taps which corresponds to 6 ms. One tap is 125 microseconds (0.125 ms).<br>**Note:** To use CSP in ASR applications, set this value to 128 taps (16 ms).<br>**Note:** If you use this parameter, you must specify this parameter BEFORE any other CSP parameter. Any time you specify DXCH_EC_TAP_LENGTH, other CSP parameters are reset to their default values.<br>**Note:** Do not specify ECCH_ECHOCANCELLER and DXCH_EC_TAP_LENGTH in your application for the same stream. Each parameter resets the other to its default value.<br>**Note:** On Springware boards, after completion of the CSP section of your application, set DXCH_EC_TAP_LENGTH to the default value of 48. This allows you to use other data formats that are not supported by CSP. |
| DXCH_EC_TAP_LENGTH (for DM3 boards)<br>Bytes: 2<br>Default: depends on media load<br>Attribute: R/W<br>Units: 0.125 ms<br>Value: depends on media load | For DM3 boards. Specifies the tap length for the echo canceller. The longer the tap length, the more echo is cancelled from the incoming signal. However, this means more processing power is required.<br>One tap is 125 microseconds (0.125 ms).<br>The media load determines what values are supported on a DM3 board. On some DM3 boards, the default value and only supported value is 128 taps (16 ms). On other DM3 boards, the default value is 512 taps (64 ms) and can be modified. To modify this value, you must edit and download the appropriate PCD/FCD file. For more information, see the appropriate Configuration Guide.<br>For information on DM3 boards that support a tap length greater than 128 taps, see the latest Release Guide. |
| DXCH_SPEECHNONPLAYTHRESH (Springware boards only)<br>Bytes: 2<br>Default: -40<br>Attribute: R/W<br>Units: decibel milliwatts (dBm)<br>Range: +3 to -54 dBm | Supported on Springware boards only. Specifies the minimum energy level of incoming speech necessary to trigger the voice activity detector. This value is used when a prompt has completed playing. **You must supply the plus or minus sign with this value.** |
| DXCH_SPEECHNONPLAYTRIGG (Springware boards only)<br>Bytes: 2<br>Default: 10<br>Attribute: R/W<br>Units: integer of 12 ms blocks<br>Range: 5-10 | Supported on Springware boards only. Specifies the number of 12 ms blocks whose speech energy is greater than the speech threshold required to trigger the voice activity detector (VAD). This value is used when a prompt has completed playing.<br>**Note:** This value must be less than or equal to the value of DXCH_SPEECHNONPLAYWINDOW. |

| Define | Description |
|---|---|
| DXCH_SPEECHNONPLAYWINDOW (Springware boards only)<br>Bytes: 2<br>Default: 10<br>Attribute: R/W<br>Units: integer of 12 ms blocks<br>Range: 5-10 | Supported on Springware boards only. Specifies the number of 12 ms blocks or frames which are surveyed to detect speech energy. This value is used when a prompt has completed playing.<br>**Note:** This value must be greater than or equal to the value of DXCH_SPEECHNONPLAYTRIGG. |
| DXCH_SPEECHPLAYTHRESH<br>Bytes: 2<br>Default: -40<br>Attribute: R/W<br>Units: decibel milliwatts (dBm)<br>Range: +3 to -54 | Specifies the minimum energy level of incoming speech necessary to trigger the voice activity detector. This value is used while a prompt is playing. **You must supply the plus or minus sign with this value.**<br>For more information on modifying these voice activity detector parameters, see the *Continuous Speech Processing API Programming Guide*.<br>On DM3 boards, specifying this parameter means that the VAD uses energy mode to determine the start of speech. For the VAD to use a combination of zero-crossing mode and energy mode, do not use this parameter; rather, set the ECCH_SVAD parameter to 0. In this case, the threshold value is set automatically by the VAD.<br>**Note:** On Springware boards and DM3 boards, you can modify this parameter while recording or streaming is active. |
| DXCH_SPEECHPLAYTRIGG (for Springware boards)<br>Bytes: 2<br>Default: 10<br>Attribute: R/W<br>Units: integer of 12 ms blocks<br>Range: 5-10 | For Springware boards. Specifies the number of 12 ms blocks whose speech energy is greater than the speech threshold required to trigger the voice activity detector. This value is used while a prompt is playing.<br>**Note:** This value must be less than or equal to the value of DXCH_SPEECHPLAYWINDOW.<br>**Note:** You can modify this parameter while recording or streaming is active. |
| DXCH_SPEECHPLAYTRIGG (for DM3 boards)<br>Bytes: 2<br>Default: 10<br>Attribute: R/W<br>Units: integer of 10 ms blocks<br>Range: 5-10 | For DM3 boards. Specifies the number of 10 ms blocks whose speech energy is greater than the speech threshold required to trigger the voice activity detector. This value is used while a prompt is playing.<br>**Note:** This value must be less than or equal to the value of DXCH_SPEECHPLAYWINDOW.<br>**Note:** You can modify this parameter while recording or streaming is active. |
| DXCH_SPEECHPLAYWINDOW (for Springware boards)<br>Bytes: 2<br>Default: 10<br>Attribute: R/W<br>Units: integer of 12 ms blocks<br>Range: 5-10 | For Springware boards. During the playing of a prompt, this parameter specifies the number of 12 ms blocks or frames which are surveyed to detect speech energy.<br>**Note:** This value must be greater than or equal to the value of DXCH_SPEECHPLAYTRIGG.<br>**Note:** You can modify this parameter while recording or streaming is active. |
| DXCH_SPEECHPLAYWINDOW (for DM3 boards)<br>Bytes: 2<br>Default: 10<br>Attribute: R/W<br>Units: integer of 10 ms blocks<br>Range: 5-10 | For DM3 boards. During the playing of a prompt, this parameter specifies the number of 10 ms blocks or frames which are surveyed to detect speech energy.<br>**Note:** This value must be greater than or equal to the value of DXCH_SPEECHPLAYTRIGG.<br>**Note:** You can modify this parameter while recording or streaming is active. |

| Define | Description |
|---|---|
| DXCH_SPEECHSNR<br>Bytes: 2<br>Default: -12<br>Attribute: R/W<br>Units: decibels (dB)<br>Range: 0 to -20 dB | Specifies the reciprocal of the signal to noise ratio (SNR) between the incoming speech energy and the estimated residual noise at the output of the echo canceller circuit.<br>SNR is the relationship of the magnitude of a transmission signal to the noise of a channel. It is a measurement of signal strength compared to error-inducing circuit noise.<br>In environments where the incoming signal is weak or has residual noise, you may want to adjust this value higher to reduce noise in the signal.<br>**You must supply the minus sign with this value.**<br>**Note:** You can modify this parameter while recording or streaming is active. |
| ECCH_ADAPTMODE<br>(Springware boards only)<br>Bytes: 2<br>Default: 0<br>Attribute: R/W<br>Values: 0 or 1 | Supported on Springware boards only. Specifies the adaptation mode of operation for the echo canceller.<br>The echo canceller uses two operating modes, fast mode and slow mode. Regardless of the parameter value, the echo canceller always starts in fast mode (higher automatic gain factor) after it is reset, then switches to a slow mode (lower automatic gain factor).<br>When this parameter is set to 0, two factors are used in determining the switch from fast to slow mode: (1) Echo Return Loss Enhancement (ERLE) and (2) adaptation time.<br>When this parameter is set to 1, only the adaptation time factor is used. For more information on the echo canceller and adaptation mode, see the *Continuous Speech Processing API Programming Guide*. |
| ECCH_ECHOCANCELLER<br>Bytes: 2<br>Default: 1<br>Attribute: R/W<br>Values: 0 or 1 | Enables or disables the echo canceller in the application.<br>The value 1 turns on the echo canceller, and the value 0 turns it off.<br>**Note:** Because the echo canceller is enabled by default, you do not need to use this parameter to turn on the echo canceller in your application. Only use this parameter to turn off the echo canceller. You may want to turn off the echo canceller for evaluation purposes.<br>**Note:** For Springware boards, if you use this parameter, you must specify this parameter BEFORE any other CSP parameter. Any time you specify ECCH_ECHOCANCELLER, the tap length and other parameters are reset to their default values.<br>**Note:** For Springware boards, do not specify ECCH_ECHOCANCELLER and DXCH_EC_TAP_LENGTH in your application for the same stream. Each parameter resets the other to its default value. |
| ECCH_NLP<br>Bytes: 2<br>Default: 0<br>Attribute: R/W<br>Values: 0 or 1 | Turns non-linear processing (NLP) on or off. The value 0 (not 1) turns on the NLP feature, and 1 (not 0) turns it off.<br>NLP refers to comfort noise; that is, a background noise used in dictation applications to let the user know that the application is working.<br>**For ASR applications, you must turn this feature off; that is, set ECCH_NLP = 1.** |

| Define | Description |
|---|---|
| ECCH_SILENCECOMPRESS (DM3 boards only)<br>Bytes: 2<br>Default: 0<br>Attribute: R/W<br>Values: 0 or 1 | Supported on DM3 boards only. This parameter enables or disables silence compressed streaming (SCS) in the application.<br>The value 1 turns this feature on, and the value 0 turns it off.<br>For more information on silence compressed streaming, see the *Continuous Speech Processing API Programming Guide*.<br>When turned on, SCS uses default parameter values that are set in the configuration file (CONFIG, PCD, FCD files). These values are downloaded to the board when it is started. Examples of SCS parameters in the configuration file are: initial data, trailing silence, speech probability threshold, silence probability threshold, and background noise thresholds. For more information on these modifying default parameter values, see the Configuration Guide for your product or product family. |
| ECCH_SVAD (DM3 boards only)<br>Bytes: 2<br>Default: 0<br>Attribute: R/W<br>Values: 0 or 1 | Supported on DM3 boards only. Specifies how the voice activity detector (VAD) detects the start of speech.<br>The value 0, the default, means that the VAD uses a combination of energy and zero-crossing mode (where energy level goes to zero for a time period) to determine the start of speech. The threshold is determined automatically by the VAD.<br>The value 1 means that the VAD uses energy mode only and the threshold value is set by the DXCH_SPEECHPLAYTHRESH parameter. |
| ECCH_VADINITIATED<br>Bytes: 2<br>Default: 1<br>Attribute: R/W<br>Values: 0 or 1 | Enables or disables voice-activated record in the application. If enabled, recording or streaming of echo-cancelled data to the host application begins only after speech is detected.<br>The value 1 turns the feature on, and 0 turns the feature off. |
| ECCH_XFERBUFFERSIZE (for Springware boards)<br>Bytes: 2<br>Default: 16 kbytes<br>Attribute: R/W<br>Units: bytes<br>Range: 128 bytes to 16 kilobytes (in multiples of 128) | For Springware boards. The size of the driver buffers on the receive side of a CSP-capable full-duplex channel. These buffers are used to transfer data between the driver and the host application.<br>This value is configurable per channel at run-time.<br>For voice-mail applications, the default of 16 kbytes is sufficient.<br>For ASR applications, you may need to set the buffer size lower to improve real-time processing and reduce latency. For more information on setting buffer sizes, see the *Continuous Speech Processing API Programming Guide*.<br>**Note:** The smaller the buffer size, the more interrupts are generated to handle the buffers, and consequently the greater the system load.<br>In Linux, this parameter has limitations. The possible values are 1, 2, 4, 8 and 16 kbytes. By default, the amount of data passed to the user-defined callback function is fixed at 16 kbytes. You can only override this default per process by calling **ec_setparm( )** BEFORE opening a channel:<br>`int size = 1024; /* or 2, 4, 8, 16 kbytes */`<br>`...`<br>`ec_setparm(SRL_DEVICE, ECCH_XFERBUFFERSIZE, &size)`<br>**Note:** You must use SRL_DEVICE as the device name.<br>For more information on buffers and data flow, see the *Continuous Speech Processing API Programming Guide*. |

| Define | Description |
|---|---|
| ECCH_XFERBUFFERSIZE (for DM3 boards)<br>Bytes: 2<br>Default: 16 kbytes<br>Attribute: R/W<br>Units: bytes<br>Range: 240 bytes to 16 kbytes | For DM3 boards. The size of the host application buffers on the receive side of a CSP-capable full-duplex channel. These buffers are used to transfer data between the firmware and the host application.<br>On DM3, the firmware buffer size is adjusted based on the value of ECCH_XFERBUFFERSIZE (the transfer buffer).<br>If the transfer buffer is less than or equal to 2 kbytes, then the firmware buffer is set to the same size as the transfer buffer.<br>If the transfer buffer is greater than 2 kbytes, then the firmware buffer is set to 2 kbytes. The content of multiple firmware buffers is accumulated in the transfer buffer before being written to file or provided to the application callback function.<br>The firmware buffer size cannot be greater than 2 kbytes.<br>This value is configurable per channel at run-time.<br>For ASR applications, you will need to set the buffer size lower to improve real-time processing and reduce latency.<br>**Note:** The smaller the buffer size, the more interrupts are generated to handle the buffers, and consequently the greater the system load.<br>**Note:** On DM3 boards, ECCH_XFERBUFFERSIZE is a channel-level parameter for both Linux and Windows. Unlike Springware boards, there are no limitations on its use for Linux. |

## ■ Cautions

- You must pass the value of the parameter to be set in a variable cast as (void *) as shown in the example.
- Be sure to use the correct data type when setting the parameter value. Some parameters require two bytes while other parameters may be ASCII strings. The data type of the variable that will receive the parameter value must match the data type for the specific parameter being queried.
- Before you use **ec_setparm( )**, the channel must be open.
- On DM3 boards, you must issue **ec_getparm( )** in the same process as **ec_setparm( )**; otherwise, the values returned for **ec_getparm( )** will be invalid.
- Certain parameters can be modified while an **ec_reciottdata( )** or **ec_stream( )** is in progress. These parameters are: DXCH_SPEECHPLAYTHRESH, DXCH_SPEECHPLAYTRIGG, DXCH_SPEECHPLAYWINDOW, DXCH_SPEECHSNR, ECCH_SVAD.
- On Springware boards, in Linux applications that use multiple threads, you must avoid having two or more threads call functions that use the same device handle; otherwise, the replies can be unpredictable and cause those functions to time out or create internal concurrency that can lead to a segmentation fault. If you must do this, use semaphores to prevent concurrent access to a particular device handle.

## ■ Errors

If the function returns -1, use **ATDV_LASTERR( )** to return the error code and **ATDV_ERRMSGP( )** to return a descriptive error message.

One of the following error codes may be returned:

EDX_BADDEV
: Device handle is NULL or invalid.

EDX_BADPARM
: Parameter is invalid or not supported.

EDX_SYSTEM
: Operating system error.

EEC_UNSUPPORTED
: Device handle is valid but device does not support CSP.

## Example

```
#include <windows.h>  /* include in Windows applications only; exclude in Linux */
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <eclib.h>
#include <errno.h> /* include in Linux applications only; exclude in Windows */

main()
{
int chdev;
short parmval;
int srlmode;         /* Standard Runtime Library mode */

    /* Set SRL to run in polled mode. */
     srlmode = SR_POLLMODE;
     if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1) {
          /* process error */
     }

    /* Open the board and get channel device handle in chdev */
     if ((chdev = dx_open("dxxxB1C1",0)) == -1) {
          /* process error */
     }

    /* Set up parameters */

    /* Enable barge-in for this channel */
     parmval = 1;
     if (ec_setparm(chdev, DXCH_BARGEIN, (void *)&parmval) == -1) {
          /* process error */
     }
     /* Set up additional parameters as needed */
     . . .
}
```

## See Also

- **ec_getparm( )**
- **dx_getparm( )** in the *Voice API Library Reference*
- **dx_setparm( )** in the *Voice API Library Reference*

## ec_stopch( )

| | | |
|---|---|---|
| **Name:** | int ec_stopch(chDev, StopType, mode) | |
| **Inputs:** | int chDev | • valid channel device handle |
| | unsigned long StopType | • type of channel stop |
| | unsigned short mode | • mode flags |
| **Returns:** | 0 for success<br>-1 for failure | |
| **Includes:** | srllib.h<br>dxxxlib.h<br>eclib.h | |
| **Category:** | I/O | |
| **Mode:** | Synchronous/Asynchronous | |
| **Platform:** | DM3, Springware | |

### ■ Description

The **ec_stopch( )** function forces termination of currently active I/O functions on a CSP-capable full-duplex channel, as defined by **StopType**.

This function can terminate CSP or voice library I/O functions.

The **StopType** value determines whether the play, receive or both sides of the channel are terminated. By contrast, the **dx_stopch( )** function only terminates the prompt play side.

| Parameter | Description |
|---|---|
| **chDev** | the channel device handle obtained when the CSP-capable device is opened using **dx_open( )** |
| **StopType** | the type of stop channel to perform:<br>• SENDING – stops the prompt play side<br>• RECEIVING – stops the side receiving echo-cancelled data<br>• FULLDUPLEX – stops both play/receive sides |
| **mode** | specifies the mode:<br>• EV_SYNC – synchronous mode<br>• EV_ASYNC – asynchronous mode |

## Cautions

- The **ec_stopch( )** has no effect on a channel that has either of the following functions issued: **dx_dial( )** without call progress analysis enabled or **dx_wink( )**. These functions continue to run normally, and **ec_stopch( )** returns a success. For **dx_dial( )**, the digits specified in the **dialstrp** parameter are still dialed.
- If **ec_stopch( )** is called on a channel dialing with call progress analysis enabled, the call progress analysis process stops but dialing is completed. Any call progress analysis information collected prior to the stop is returned by extended attribute functions.
- If an I/O function terminates (due to another reason) before **ec_stopch( )** is issued, the reason for termination does not indicate **ec_stopch( )** was called.
- When calling **ec_stopch( )** from a signal handler, you must set **mode** to EV_ASYNC.
- On Springware boards, in Linux applications that use multiple threads, you must avoid having two or more threads call functions that use the same device handle; otherwise, the replies can be unpredictable and cause those functions to time out or create internal concurrency that can lead to a segmentation fault. If you must do this, use semaphores to prevent concurrent access to a particular device handle.

## Errors

If the function returns -1, use **ATDV_LASTERR( )** to return the error code and **ATDV_ERRMSGP( )** to return a descriptive error message.

One of the following error codes may be returned:

| | |
|---|---|
| EDX_BADDEV | Device handle is NULL or invalid. |
| EDX_BADPARM | Stop Type or mode is invalid. |
| EDX_SYSTEM | Operating system error. |
| EEC_UNSUPPORTED | Device handle is valid but device does not support CSP. |

## Example

```
#include <windows.h>  /* include in Windows applications only; exclude in Linux */
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <eclib.h>
#include <errno.h> /* include in Linux applications only; exclude in Windows */

main()
{
     int chdev, srlmode;

    /* Set SRL to run in polled mode. */
     srlmode = SR_POLLMODE;
     if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1) {
          /* process error */
     }
```

```
        /* Open the channel using dx_open( ). Get channel device descriptor in
         * chdev.
         */
         if ((chdev = dx_open("dxxxB1C1",0)) == -1) {
              /* process error */
         }

        /* continue processing */
                  .
                  .

        /* Force the channel idle.  The I/O function that the channel is
         * executing will be terminated, and control passed to the handler
         * function previously enabled, using sr_enbhdlr(), for the
         * termination event corresponding to that I/O function.
         * In the asynchronous mode, ec_stopch() returns immediately,
         * without waiting for the channel to go idle.
         */
         if (ec_stopch(chdev, FULLDUPLEX, EV_ASYNC) == -1) {
              /* process error */
         }
    }
```

## ■ See Also

- **dx_stopch( )** in the *Voice API Library Reference*

## ec_stream( )

| | | |
|---|---|---|
| **Name:** | int ec_stream(chDev, tptp, xpbp, callback, mode) | |
| **Inputs:** | int chDev | • valid channel device handle |
| | DV_TPT *tptp | • pointer to termination parameter table |
| | DX_XPB *xpbp | • pointer to I/O transfer parameter block table |
| | int (*callback) (int, char*, uint) | • address of a function to receive recorded data buffers |
| | unsigned short mode | • stream mode |
| **Returns:** | 0 for success<br>-1 for failure | |
| **Includes:** | srllib.h<br>dxxxlib.h<br>eclib.h | |
| **Category:** | I/O | |
| **Mode:** | Synchronous/Asynchronous | |
| **Platform:** | DM3, Springware | |

### ■ Description

The **ec_stream( )** function streams echo-cancelled data to a callback function on a CSP-capable full-duplex channel. This user-defined callback function is called every time the driver fills the driver buffer with data. See ECCH_XFERBUFFERSIZE in the **ec_setparm( )** function description for information on setting the driver buffer size.

This function is designed specifically for use in ASR applications where echo-cancelled data must be streamed to the host application in real time for further processing, such as comparing the speech utterance against an employee database and then connecting the caller to the intended audience.

You can perform voice streaming at all times or voice-activated streaming. The ECCH_VADINITIATED parameter in **ec_setparm( )** controls voice-activated streaming, where recording begins only after speech energy has been detected. This parameter is enabled by default.

You can record or stream speech energy with silence removed from the data stream. To use the silence compressed streaming feature, enable ECCH_SILENCECOMPRESS in **ec_setparm( )**. This parameter is disabled by default.

| Parameter | Description |
|---|---|
| **chDev** | the channel device handle obtained when the CSP-capable device is opened using **dx_open( )** |
| **tptp** | pointer to the DV_TPT data structure that sets the termination conditions for the device handle<br>***Note:*** On Springware boards, the only supported DV_TPT terminating conditions are DX_MAXTIME, DX_MAXSIL, and DX_MAXNOSIL. The voice activity detector timeout period is set by the **tp_length** parameter in the DV_TPT structure. For more information on DV_TPT, see the *Voice API Library Reference*.<br>***Note:*** On DM3 boards, all DV_TPT terminating conditions are supported except for DX_LCOFF, DX_PMON and DX_PMOFF.<br>***Note:*** In CSP, DV_TPT terminating conditions are edge-sensitive. |
| **xpbp** | pointer to the DX_XPB data structure that specifies the file format, data format, sampling rate and resolution |
| **callback** | the user-defined callback function that receives the echo-cancelled stream. For more information on the user-defined callback function, see the description following this table. |
| **mode** | a logical OR bit mask that specifies the stream mode:<br>• EV_SYNC – synchronous mode<br>• EV_ASYNC – asynchronous mode<br>• MD_GAIN – automatic gain control (AGC)<br>• MD_NOGAIN – no automatic gain control<br>***Note:*** For ASR applications, turn AGC off. |

The user-defined callback function is similar to the C library **write( )** function. Its prototype is:

```
int callback (int chDev, char *buffer, uint length)
```

The definition of the arguments is:

chDev
: CSP channel on which streaming is performed

buffer
: the buffer that contains streamed data

length
: the length of the data buffer in bytes

This user-defined callback function returns the number of bytes contained in **length** upon success. Any other value is viewed as an error and streaming is terminated. We do not recommend terminating the streaming activity in this way; instead, use **ec_stopch( )**.

We recommend that inside the user-defined callback function you:

- do **not** call another Intel Dialogic function
- do **not** call a blocking function such as sleep

- do **not** call an I/O function such as printf, scanf, and so on (although you may use these for debugging purposes)

We recommend that inside the user-defined callback function you do the following:

- Copy the **buffer** contents for processing in another context.
- Signal the other context to begin processing.

## ■ Cautions

- This function fails if an unsupported data format is specified. For a list of supported data formats, see *Continuous Speech Processing API Programming Guide*.
- We recommend that you specify the **ec_stream( )** function **before** a voice play function in your application.
- To set the proper parameters, the **ec_stream( )** function must be called for every **ec_stream( )** occurrence in your application.
- If you use this function in synchronous mode, you must use multithreading in your application.
- All files recorded to have the data encoding and rate as described in DX_XPB.
- The DX_XPB data area must remain in scope for the duration of the function if running asynchronously.
- On Springware boards, we recommend that you use the same data format for play and recording/streaming.
- On Springware boards, in Linux applications that use multiple threads, you must avoid having two or more threads call functions that use the same device handle; otherwise, the replies can be unpredictable and cause those functions to time out or create internal concurrency that can lead to a segmentation fault. If you must do this, use semaphores to prevent concurrent access to a particular device handle.
- The DX_MAXSIL and DX_MAXNOSIL termination conditions are not supported on CSP-enabled boards that have an analog front-end and Japan Caller ID enabled with polarity reversal. This limitation affects D/120JCT Rev 2 and D/41JCT-LS boards. Thus, these termination conditions cannot be specified in the DV_TPT structure for **ec_reciottdata( )** and **ec_stream( )**.
- When DX_MAXSIL and DX_MAXNOSIL termination conditions are used, the silence timer starts after a **dx_play( )** has completed. To start the silence timer immediately after the onset of **ec_stream( )** or **ec_reciottdata( )**, use the bit flag TF_IMMEDIATE (tp_flags field in the DV_TPT structure). This flag is not supported on Springware boards.

## ■ Errors

If the function returns -1, use **ATDV_LASTERR( )** to return the error code and **ATDV_ERRMSGP( )** to return a descriptive error message.

One of the following error codes may be returned:

EDX_BADDEV
Device handle is NULL or invalid.

EDX_BADPARM
Parameter is invalid.

EDX_SYSTEM
Operating system error.

EEC_UNSUPPORTED
Device handle is valid but device does not support CSP.

## ■ Example

```
#include <windows.h>  /* include in Windows applications only; exclude in Linux */
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <eclib.h>
#include <errno.h>    /* include in Linux applications only; exclude in Windows */

main()
{

     char  csp_devname[9];
     int   ret, csp_dev, parmval=0;
     SC_TSINFO  sc_tsinfo;   /* Time slot information structure */
     long scts;              /* TDM bus time slot */
     int srlmode;            /* Standard Runtime Library mode */
     DX_IOTT iott;           /* I/O transfer table */
     DV_TPT  tptp[1], tpt;   /* termination parameter table */
     DX_XPB xpb;             /* I/O transfer parameter block */

    /* Set SRL to run in polled mode. */
     srlmode = SR_POLLMODE;
     if (sr_setparm(SRL_DEVICE, SR_MODEID, (void *)&srlmode) == -1)
     {
          /* process error */
     }

     /* Barge-in parameters */
     short BargeIn= 1;

     sprintf(csp_devname,"dxxxB1C1");

     /* Open a voice device. */
     csp_dev = dx_open(csp_devname, 0);
     if (csp_dev < 0)
     {
          printf("Error %d in dx_open()\n",csp_dev);
          exit(-1);
     }

     /* Set up ec parameters (use default if not being set).
      * Enable barge-in. ECCH_VADINITIATED is enabled by default.
      */
     ret = ec_setparm(csp_dev, DXCH_BARGEIN, (void *) &BargeIn);
     if (ret == -1)
     {
          printf("Error in ec_setparm(DXCH_BARGEIN). Err Msg = %s, Lasterror = %d\n",
          ATDV_ERRMSGP(csp_dev), ATDV_LASTERR(csp_dev));
     }

     /* Set up DV_TPT for record */
     tpt.tp_type   = IO_EOT;
     tpt.tp_termno = DX_MAXTIME;
     tpt.tp_length = 60;
     tpt.tp_flags  = TF_MAXTIME;
```

```
/* Record data format set to 8-bit PCM, 8KHz sampling rate */
xpb.wFileFormat = FILE_FORMAT_VOX;
xpb.wDataFormat = DATA_FORMAT_PCM;
xpb.nSamplesPerSec = DRT_8KHZ;
xpb.wBitsPerSample = 8;

ret = ec_stream(csp_dev, &tpt, &xpb, &stream_cb, EV_ASYNC | MD_NOGAIN);
if (ret == -1)
{
     printf("Error in ec_reciottdata(). Err Msg = %s, Lasterror = %d\n",
               ATDV_ERRMSGP(csp_dev), ATDV_LASTERR(csp_dev));
}

/* Set channel off-hook */
ret = dx_sethook(csp_dev, DX_OFFHOOK, EV_SYNC);
if (ret == -1)
{
     printf("Error in dx_sethook(). Err Msg = %s, Lasterror = %d\n",
               ATDV_ERRMSGP(csp_dev), ATDV_LASTERR(csp_dev));
}

/* Set up DX_IOTT */
iott.io_type   = IO_DEV|IO_EOT;
iott.io_bufp   = 0;
iott.io_offset = 0;
iott.io_length = -1;

/* Set up DV_TPT for play */
dx_clrtpt(&tptp,1);
tptp[0].tp_type   = IO_EOT;
tptp[0].tp_termno = DX_MAXDTMF;
tptp[0].tp_length = 1;
tptp[0].tp_flags  = TF_MAXDTMF;

/* Open file to be played */
#ifdef WIN32
if ((iott.io_fhandle = dx_fileopen("sample.vox",O_RDONLY|O_BINARY)) == -1)
{
     printf("Error opening sample.vox.\n");
     exit(1);
}
#else
if (( iott.io_fhandle = open("sample.vox",O_RDONLY)) == -1)
{
     printf("File open error\n");
     exit(2);
}
#endif

/* Play prompt message. */
ret = dx_play(csp_dev, &iott, &tptp, EV_ASYNC);
if ( ret == -1)
{
     printf("Error playing sample.vox.\n");
     exit(1);
}

/* Wait for barge-in and echo-cancelled record to complete  */
while (1)
{
     sr_waitevt(-1);
     ret = sr_getevttype();
     if (ret == TDX_BARGEIN)
     {
          printf("TDX_BARGEIN event received\n");
     }
     else if (ret == TDX_PLAY)
```

```
            {
                printf("Play Completed event received\n");
                break;
            }
            else if (ret == TEC_STREAM)
            {
                printf("TEC_STREAM - termination event ");
                printf("for ec_stream and ec_reciottdata received.\n");
                break;
            }

            else if (ret == TDX_ERROR)
            {
                printf("ERROR event received\n");
            }
            else
            {
                printf("Event 0x%x received.\n", ret);
            }

        } /* end while */

        // Set channel on hook
        dx_sethook(csp_dev, DX_ONHOOK, EV_SYNC);

        /* Close play file */
        #ifdef WIN32
        if (dx_fileclose(iott.io_fhandle) == -1)
        {
            printf("Error closing file.\n");
            exit(1);
        }
        #else
        if (close(iott.io_fhandle) == -1)
        {
            printf("Error closing file. \n");
            exit(1);
        }
        #endif

        // Close channel
        dx_close(csp_dev);
    }

    int stream_cb(int chDev, char *buffer, int length)
    {
        /* process recorded data here ... */
        return(length);
    }
```

## ■ See Also

- **ec_reciottdata( )**
- **ec_stopch( )**
- **dx_reciottdata( )** in the *Voice API Library Reference*

## ec_unlisten( )

| | | |
|---|---|---|
| **Name:** | int ec_unlisten(chDev) | |
| **Inputs:** | int chDev | • valid channel device handle |
| **Returns:** | 0 for success<br>-1 for failure | |
| **Includes:** | srllib.h<br>dxxxlib.h<br>eclib.h | |
| **Category:** | Routing | |
| **Mode:** | Synchronous | |
| **Platform:** | DM3, Springware | |

### ■ Description

The **ec_unlisten( )** function changes the echo-reference signal set by **ec_listen( )** back to the default reference (that is, the same channel as the play).

| Parameter | Description |
|---|---|
| **chDev** | the channel device handle obtained when the CSP-capable device is opened using **dx_open( )** |

### ■ Cautions

This function fails if you specify an invalid channel device handle.

### ■ Errors

If the function returns -1, use **ATDV_LASTERR( )** to return the error code and **ATDV_ERRMSGP( )** to return a descriptive error message.

One of the following error codes may be returned:

EDX_BADDEV
    Device handle is NULL or invalid.

EDX_BADPARM
    Time slot pointer information is NULL or invalid.

EEC_UNSUPPORTED
    Device handle is valid but device does not support CSP.

## Example

```
#include <windows.h>  /* include in Windows applications only; exclude in Linux */
#include <stdio.h>
#include <srllib.h>
#include <dxxxlib.h>
#include <eclib.h>
#include <errno.h>    /* include in Linux applications only; exclude in Windows */

main()
{
     int chdev;                 /* voice channel device handle */
     /* Open board 1 channel 1 device */
     if ((chdev = dx_open("dxxxB1C1", 0)) == -1)
     {
          printf("Cannot open channel dxxxB1C1." );
          exit(1);
     }
     /* Disconnect receive of board 1 channel 1 from all TDM bus time slots */
     if (ec_unlisten(chdev) == -1)
     {
          printf("Error message = %s", ATDV_ERRMSGP(chdev));
          exit(1);
     }
}
```

## See Also

- **ag_listen( )** in the *Voice API Library Reference*
- **dt_listen( )** in the *Voice API Library Reference*
- **dx_listen( )** in the *Voice API Library Reference*

# Data Structures 3

This chapter provides an alphabetical reference to the data structures used by Continuous Speech Processing library functions. The following data structure is discussed:

## EC_BLK_INFO

```
typedef struct EC_BLK_INFO
{
     EC_BlockType   type;      /* type of block */
     unsigned int   flags;     /* bit mask with EC_BlockFlags value */
     unsigned int   size;      /* size of block  */
     unsigned long  timestamp; /* time stamp of first sample of the block */
} EC_BLK_INFO;

typedef enum {
     EC_INITIAL_BLK,          /* A block with the initial data */
                              /* (for ASR engine adjustments) [optional] */
     EC_SPEECH_BLK,           /* A block with speech */
     EC_SPEECH_LAST_BLK       /* Last block of speech before silence. */
                              /* The block size could actually be 0, */
                              /* in which case the block simply indicates */
                              /* that silence is now being compressed until */
                              /* EC_SPEECH_BLK blocks are sent again */
} EC_BlockType;

typedef enum {
     EC_LAST_BLK    = 0x1     /* Last block of a streaming session */
} EC_BlockFlags;
```

### ■ Description

The EC_BLK_INFO data structure contains information for a block of echo-cancelled data. Information includes whether this block contains the initial data, whether it contains speech, and whether it is the last block of speech before silence. This data structure is used by the **ec_getblkinfo( )** function. This data structure is defined in *eclib.h*.

### ■ Field Descriptions

The fields of the EC_BLK_INFO data structure are described as follows:

type
: Specifies whether this is a block with initial data, a block with speech, or the last block of speech before silence.

  The EC_BlockType data type is an enumeration that defines the following values:
  - EC_INITIAL_BLK – a block with initial data
  - EC_SPEECH_BLK – a block with speech
  - EC_SPEECH_LAST_BLK – the last block of speech before a silence period

flags
: Specifies a bit mask of EC_BlockFlags values.

  The EC_BlockFlags data type is an enumeration that defines the following value:
  - EC_LAST_BLK – The last block of speech before streaming ends. The fact that streaming ends is also indicated by the TEC_STREAM termination event being sent. The value 1 indicates that this is the last speech block of a streaming session.

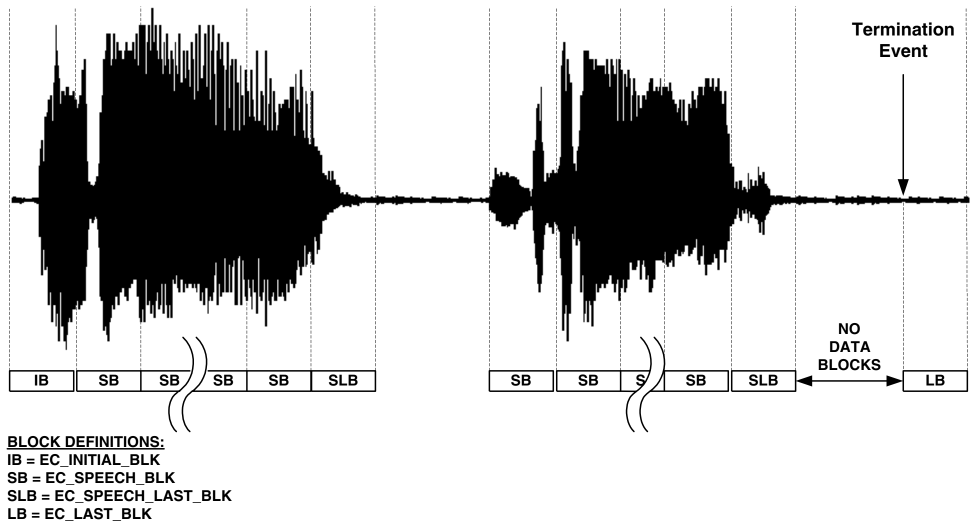
size
: Specifies the size of the block indicated in the type field. This value is given in bytes. The last block size may be 0, which indicates that silence is now being compressed until EC_SPEECH_BLK blocks are sent again.

timestamp

Specifies the time stamp of the first sample of the block indicated in the type field. This value is given as the number of frames, with each frame equal to 10 milliseconds.

The following figure illustrates the concept of the data blocks in the EC_BLK_INFO data structure.

**Figure 2. Example of Data Blocks in EC_BLK_INFO**



## ■ Example

For an example, see the Example section of the **ec_getblkinfo( )** function description.

# Events 4

This chapter provides information on events that can be returned by the Continuous Speech Processing (CSP) software.

An event indicates that a specific activity has occurred on a channel. The voice driver reports channel activity to the application program in the form of events, which allows the program to identify and respond to a specific occurrence on a channel. Events provide feedback on the progress and completion of functions and indicate the occurrence of other channel activities. Events are sometimes referred to in general as termination events, because most of them indicate the end of an operation.

The following events, listed in alphabetical order, can be returned by the CSP software. Use **sr_waitevt( )**, **sr_enbhdlr( )** or other SRL functions to collect an event code, depending on the programming model in use. For more information, see the *Standard Runtime Library API Library Reference* and the *Standard Runtime Library API Programming Guide*.

TEC_CONVERGED
: Termination event. Occurs when the echo canceller sends a message to the host application that the incoming signal has been echo-cancelled (converged). Echo-cancelled convergence notification is enabled using the DM_CONVERGED parameter in **dx_setevtmsk( )**.

TEC_STREAM
: Termination event. Indicates that an echo-cancelled record function, **ec_reciottdata( )**, or echo-cancelled stream function, **ec_stream( )**, has ended.

TEC_VAD
: Termination event. Occurs when the voice activity detector (VAD) sends a message to the host application that significant speech energy has been detected. VAD event notification is enabled using the DM_VADEVTS parameter in **dx_setevtmsk( )**.

TDX_BARGEIN
: Termination event. Indicates that play was terminated by the VAD due to barge-in. Barge-in is enabled using the DXCH_BARGEIN parameter in **ec_setparm( )**.

TDX_CST
: Unsolicited event. Indicates a firmware buffer overrun when the event appears with the REC_BUF_OVERFLOW subcode. To retrieve the DX_CST structure and this code, use **sr_getevtdatap( )**.

TDX_PLAY
: Termination event. Can optionally be received (in addition to TDX_BARGEIN) to indicate that play was terminated by the VAD. Specify using DXCH_BARGEINONLY parameter in **ec_setparm( )**.

> ***Note:*** When the VAD is not enabled, the TDX_PLAY event indicates termination for **dx_play( )** in asynchronous mode. For more information, see the *Voice API Library Reference*.

# Glossary

**application programming interface (API):** A set of standard software interrupts, calls, and data formats that application programs use to initiate contact with network services or other program-to-program communications.

**asynchronous function:** A function that allows program execution to continue without waiting for a task to complete. To implement an asynchronous function, an application-defined event handler must be enabled to trap and process the completed event. See synchronous function.

**automatic speech recognition (ASR):** A set of algorithms that processes speech utterances.

**barge-in:** The act of a party beginning to speak while a prompt is being played. When the VAD detects significant energy in the voice channel, CSP can optionally terminate prompts playing on that channel. Thus the party on the other end of the line is said to have “barged in” on the prompt.

**buffer:** A block of memory or temporary storage device that holds data until it can be processed. It is used to compensate for the difference in the rate of the flow of information when transmitting data from one device to another.

**comfort noise generation (CNG):** The ability to produce a background noise when there is no speech on the telephone line.

**convergence:** The point at which the echo canceller processes enough data to be able to identify the echo component in the incoming signal and thereby reduce it to provide echo-cancelled data to the host.

**device:** A computer peripheral or component controlled through a software device driver. A voice and/or network interface expansion board is considered a physical board containing one or more logical board devices, where each channel or time slot on the board is a device.

**DM3:** Mediastream processing architecture from Intel. It is open, layered, and flexible, encompassing hardware as well as software components. A whole set of products are built on DM3 architecture.

**driver:** A software module that provides a defined interface between a program and the hardware.

**echo:** The component of an outgoing signal (that is, the play prompt) reflected in the incoming signal. The echo occurs when the signal passes through an analog device or other interface somewhere in the circuit.

**echo-cancelled signal:** The incoming signal whose echo component has been significantly reduced by the echo canceller.

**echo cancellation (EC):** A technique used to significantly reduce traces of an outgoing prompt in the incoming signal. These traces are referred to as echo. The **echo canceller** is the component in CSP responsible for performing echo cancellation.

**firmware:** A set of program instructions that are resident (usually in EPROM) on an expansion board.

**fixed routing:** In this configuration, the resource devices (voice/fax) and network interface devices are permanently coupled together in a fixed configuration. Only the network interface time slot device has access to the CT Bus.

**flexible routing:** In this configuration, the resource devices (voice/fax) and network interface devices are independent, which allows exporting and sharing of the resources. All resources have access to the CT Bus.

**incoming signal or incoming speech signal:** The speech uttered by the caller, or the DTMF tone entered by the caller. Also known as the **echo-carrying signal**. This signal contains an echo component only if an outgoing prompt is played while the incoming signal is generated.

**latency:** The lag time experienced as a result of audio energy traveling over the telephone or data network from the sender to the receiver.

**library:** A collection of precompiled routines that a program can use. The routines, sometimes called modules, are stored in object format. Libraries are particularly useful for storing frequently used routines because you do not need to explicitly link them to every program that uses them. The linker automatically looks in libraries for routines that it does not find elsewhere.

**non-linear processing (NLP):** A process used to block or suppress the residual (echo-cancelled) signal, when there is no near end speech. This process can be used with **comfort noise generation** (CNG) to produce background noise. Background noise energy estimation is used to adjust the level of comfort noise generated. This allows the speaker to listen to the same level of background noise when the non-linear processor is switched on and off due to double-talk situations or near end speech. A typical usage of this feature is background noise used in dictation applications to let the user know that the application is working.

**outgoing prompt or outgoing signal:** The speech in a computer telephony application that is played to a caller. Also known as the **echo-generating signal**.

**pre-speech buffer:** A circular buffer that stores the incoming speech signal and is used to reduce the problem of clipped speech. This data, which includes the incoming speech signal prior to the VAD trigger, is then sent to the host application for processing. This action ensures that minimal incoming data is lost due to VAD latency.

**reference signal or echo-reference signal:** The outgoing signal that is free of echo before it is passed to the network device. This signal is used by the echo canceller to characterize the echo to be removed from the incoming signal.

**Standard Attribute functions:** Class of functions that take one input parameter (a valid device handle) and return generic information about the device. For instance, Standard Attribute functions return IRQ and error information for all device types. Standard Attribute function names are case-sensitive and must be in capital letters. Standard Attribute functions for all devices are contained in the SRL. See Standard Runtime Library.

**Springware:** Downloadable signal- and call-processing firmware from Intel. Also refers to boards whose device family is not DM3.

**Standard Runtime Library (SRL):** Software resource containing Event Management and Standard Attribute functions and data structures used by all devices, but which return data unique to the device.

**synchronous function:** A function that blocks program execution until a value is returned by the device. Also called a blocking function. See asynchronous function.

**tap length:** Also called tail length or length. Refers to the number of milliseconds of echo that is eliminated from the incoming signal. The length of an echo canceller is sometimes given as “taps,” where each tap is 125 microseconds long.

**TDM bus:** time division multiplexing bus. A resource sharing bus such as the SCbus or CT Bus that allows information to be transmitted and received among resources over multiple data lines.

**utterance:** Speech made by the user.

**voice activity detector (VAD):** Also called voice energy detector. This detector identifies the presence of speech energy and determines when significant energy is detected in the voice channel. It notifies the host application that speech energy is detected.

**zero-crossing:** Refers to energy that crosses over the zero mark many times over a short period of time and thus has a high probability of being speech.

# Index