# Continuous Speech Processing API for Linux and Windows

## Library Reference

*December 2001*

05-1700-001

## COPYRIGHT NOTICE



Publication Date: December 2001

Document Number: 05-1700-001

Intel Corporation
Telecommunications and Embedded Group
1515 Route 10
Parsippany, NJ 07054

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

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

# Table of Contents

# Figures

# About This Publication

The following topics provide information about this publication:

- 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.

It 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
- 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 which includes the CSP 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 Software Reference: Programmer's Guide* and *Voice Software Reference: Standard Runtime Library* 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 Reference" provides an alphabetical reference to the CSP functions.

- Chapter 3, “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:

- *Continuous Speech Processing API Programming Guide*
- *Continuous Speech Processing Demo Guide*
- *System Release Guide*
- *System Release Update* (available on the Dialogic Technical Support Web site only)
- *System Release Installation and Configuration Guide*
- *Voice Software Reference,* which includes the *Voice Features Guide, Standard Runtime Library Programmer’s Guide* and *Voice Programmer’s Guide*
- *Compatibility Guide for the Dialogic R4 API on DM3 Products*
- *SCbus Routing Function Reference*
- *GlobalCall™ API Software Reference*
- *ISDN Software Reference*
- *DM3 Configuration File Reference*
- *http://support.dialogic.com*

# Function Summary by Category 1

The Continuous Speech Processing (CSP) library provides functions for building CSP-enabled applications. The CSP library functions can be grouped into the following categories:

- Input/Output Functions
- Configuration Functions
- Routing Functions

## 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.

**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 transmit time slot of a CSP-capable channel.

**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).

# Function Reference 2

An alphabetical reference to the functions in the Continuous Speech Processing (CSP) library is provided.

## 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.

# 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
-1 for failure

**Includes:** srllib.h
dxxxlib.h
eclib.h

**Category:** configuration

**Mode:** synchronous

## 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. **You must also 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 2 bytes while other parameters may be ASCII strings.

## 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, 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 */
  }

/* 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 */
  . . .
}
```

## ■ 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:

| Error code | Reason |
|---|---|
| EDX_BADDEV | Device handle is NULL or invalid. |
| EDX_BADPARM | Parameter is invalid or not supported. |
| 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. |

## ■ See Also

- **ec_setparm( )**
- **dx_setparm( )**

# ec_getxmitslot( )

**Name:** int ec_getxmitslot(chDev, lpSlot)

**Inputs:** int chDev • valid channel device handle

SC_TSINFO *lpSlot • pointer to time slot information structure

**Returns:** 0 for success
-1 for failure

**Includes:** srllib.h
dxxxlib.h
eclib.h

**Category:** routing

**Mode:** synchronous

## ■ 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 SCbus time slot which transmits the echo-cancelled data. This information is contained in an SC_TSINFO structure.

*Note:* On DM3 boards, the **ec_getxmitslot( )** function is not supported.

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.

A voice channel on an SCbus-based board can transmit on only one SCbus time slot.

For more information, see the *SCbus Routing Function Reference*.

| 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 "ag" for analog. See the *SCbus Routing Function Reference* for more information.
- In Linux applications that use multiple threads, you must avoid having two or more threads call functions that send commands to the same channel; otherwise, the replies can be unpredictable and cause those functions to time out. If you must do this, use semaphores to prevent concurrent access to a particular channel.

## 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;                  /* SCbus 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 SCbus time slot information */
   sc_tsinfo.sc_numts = 1;
   sc_tsinfo.sc_tsarrayp = &scts;

  /* Get SCbus 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 SCbus time slot %ld", ATDV_NAMEP(chdev), scts);
}
```

## 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:

| Error code | Reason |
|---|---|
| EDX_BADDEV | Device handle is invalid or NULL. |
| EDX_BADPARM | Time slot pointer information is NULL or invalid. |
| EDX_SH_BADCMD | Command is not supported in current bus configuration. |
| EDX_SH_BADINDX | Switch Handler index number is NULL. |
| EDX_SH_BADLCLTS | Channel number is invalid. |
| EDX_SH_BADMODE | Function is not supported in current bus configuration. |
| EDX_SH_BADTYPE | Channel type is invalid. |
| EDX_SH_CMDBLOCK | Blocking command is in progress. |
| EDX_SH_LCLDSCNCT | Channel is already disconnected from SCbus. |
| 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 occurred. |
| EEC_UNSUPPORTED | Device handle is valid but device does not support CSP. |

## See Also

- **ag_getxmitslot( )**
- **dt_getxmitslot( )**
- **dx_getxmitslot( )**

# ec_listen( )

**Name:** int ec_listen(chDev, lpSlot)

**Inputs:** int chDev • valid channel device handle

SC_TSINFO *lpSlot • pointer to time slot array

**Returns:** 0 for success
-1 for failure

**Includes:** srllib.h
dxxxlib.h
eclib.h

**Category:** routing

**Mode:** synchronous

## ■ 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 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.

For more information, see the *SCbus Routing Function Reference*.

| 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.
- For SpringWare boards, if you set **ec_setparm( )** parameters for use with **ec_listen( )**, you must do so in a specific order:
  - Set DXCH_EC_TAP_LENGTH as needed. (The echo canceller, ECCH_ECHOCANCELLER, is enabled by default, so you do not need to specify this parameter in your application.)
  - Call **ec_listen( )**.
  - Set ECCH_NLP off (ECCH_NLP = 1).
  - Specify other parameters as needed.
- In Linux applications that use multiple threads, you must avoid having two or more threads call functions that send commands to the same channel; otherwise, the replies can be unpredictable and cause those functions to time out. If you must do this, use semaphores to prevent concurrent access to a particular channel.

## 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;          /* SCbus 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( ... );

   }
}
```

## ■ 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:

| Error code | Reason |
|---|---|
| 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. |

## ■ See Also

- **ag_listen( )**
- **dt_listen( )**
- **dx_listen( )**

# ec_rearm( )

**Name:** int ec_rearm(chDev)

**Inputs:** int chDev
- valid channel device handle

**Returns:** 0 for success
-1 for failure

**Includes:** srllib.h
dxxxlib.h
eclib.h

**Category:** I/O

**Mode:** synchronous

## 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.

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

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

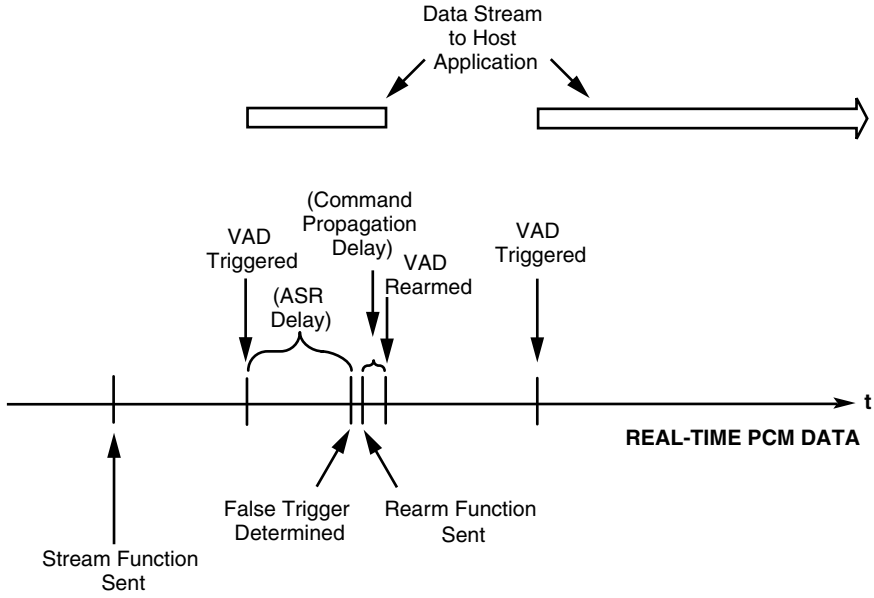
The following scenario describes how the **ec_rearm( )** function works:

- 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 2-1 illustrates the rearming concept.

**Figure 2-1. Rearming the Voice Activity Detector (VAD)**




## ■ 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;              /* SCbus 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 Dialogic 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) */

.
.
.
```

## ■ 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:

| Error code | Reason |
|---|---|
| EDX_BADDEV | Device handle is NULL or invalid. |
| EDX_BADPARM | Parameter is invalid. |
| EEC_UNSUPPORTED | Device handle is valid but device does not support CSP. |

# 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
-1 for failure

**Includes:** srllib.h
dxxxlib.h
eclib.h

**Category:** I/O

**Mode:** synchronous/asynchronous

## ■ 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, enable ECCH_VADINITIATED in **ec_setparm( )**.

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

| Parameter | Description |
|---|---|
| **tptp** | Pointer to the DV_TPT table 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 Software Reference: Programmer's Guide*.<br>***Note:*** On DM3 boards, all DV_TPT terminating conditions are supported except for DX_MAXTIME, DX_LCOFF, DX_PMON and DX_PMOFF. For more information on DV_TPT limitations on DM3 boards, see the *Compatibility Guide for the Dialogic R4 API on DM3 Products*.<br>***Note:*** In CSP, DV_TPT terminating conditions are edge-sensitive. |
| **xpbp** | Pointer to the DX_XPB table that specifies the file format, data format, sampling rate and sampling size. |
| **mode** | A 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*.
- On SpringWare boards, we recommend that you use the same data format for play and recording/streaming.
- 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.
- In Linux applications that use multiple threads, you must avoid having two or more threads call functions that send commands to the same channel; otherwise, the replies can be unpredictable and cause those functions to time out. If you must do this, use semaphores to prevent concurrent access to a particular channel.
- All files specified in the DX_IOTT table 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.

## 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);
```

## ■ 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:

| Error code | Reason |
|---|---|
| EDX_BADDEV | Device handle is NULL or invalid. |
| EDX_BADPARM | Parameter is invalid or not supported. |
| EDX_BUSY | Channel is busy. |
| EDX_XPBPARM | DX_XPB setting is invalid. |
| EDX_BADIOTT | DX_IOTT setting is invalid. |
| EDX_SYSTEM | Operating system error occurred. |
| EDX_BADWAVFILE | WAVE file is invalid. |
| EDX_SH_BADCMD | Command or WAVE file format is not supported. |
| EEC_UNSUPPORTED | Device handle is valid but device does not support CSP. |

## See Also

- **ec_stream( )**
- **dx_reciottdata( )**

# 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
-1 for failure

**Includes:** srllib.h
dxxxlib.h
eclib.h

**Category:** configuration

**Mode:** synchronous

## 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 the 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 |
| **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<br>(SpringWare boards only)<br>Bytes: 2<br>Default: 512<br>Attribute: R/W<br>Units: bytes<br>Range: 128-512 | For 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:** To modify the default value of 512, you must edit the *voice.prm* file. For details, see the installation and configuration guide. |
| DXBD_TXBUFSIZE<br>(SpringWare boards only)<br>Bytes: 2<br>Default: 512<br>Attribute: R/W<br>Units: bytes<br>Range: 128-512 | For 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:** To modify the default value of 512, you must edit the *voice.prm* file. For details, see the installation and configuration guide. |
| 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 3, “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<br>(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. |
| DXCH_EC_TAP_LENGTH<br>(for DM3 boards)<br>Bytes: 2<br>Default: 128<br>Attribute: R/W<br>Units: 0.125 ms<br>Value: 128 | 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>The default value is 128 taps which corresponds to 16 ms. This value can not be modified. |
| DXCH_SPEECHNONPLAYTHRESH<br>(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<br>(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. |
| DXCH_SPEECHNONPLAYWINDOW<br>(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. |

| Define | Description |
|---|---|
| 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<br>(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<br>(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<br>(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<br>(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_SVAD<br>(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. |

| Define | Description |
|---|---|
| ECCH_XFERBUFFERSIZE<br>(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*. |
| ECCH_XFERBUFFERSIZE<br>(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. |

## Cautions

- You must pass the value of the parameter to be set in a variable cast as (void *) as shown in the example.
- Before you use **ec_setparm( )**, the channel must be open.
- 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.
- In Linux applications that use multiple threads, you must avoid having two or more threads call functions that send commands to the same channel; otherwise, the replies can be unpredictable and cause those functions to time out. If you must do this, use semaphores to prevent concurrent access to a particular channel.

## 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, 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 */
   . . .
}
```

## 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:

| Error code | Reason |
|---|---|
| EDX_BADDEV | Device handle is NULL or invalid. |
| EDX_BADPARM | Parameter is invalid or not supported. |
| EDX_SYSTEM | Operating system error occurred. |
| EEC_UNSUPPORTED | Device handle is valid but device does not support CSP. |

## See Also

- **ec_getparm( )**
- **dx_getparm( )**
- **dx_setparm( )**

# 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
-1 for failure

**Includes:** srllib.h
dxxxlib.h
eclib.h

**Category:** I/O

**Mode:** synchronous/asynchronous

## Description

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

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. In 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 receive side<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 Analysis enabled
  - **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 Analysis enabled, the Call Analysis process stops but dialing is completed. Any Call 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.
- In Linux, applications that use multiple threads, you must avoid having two or more threads call functions that send commands to the same channel; otherwise, the replies can be unpredictable and cause those functions to time out. If you must do this, use semaphores to prevent concurrent access to a particular channel.

## ■ 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 */
   }
}
```

## 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:

| Error code | Reason |
|---|---|
| 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. |

## See Also

- **dx_stopch( )**

# 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
-1 for failure

**Includes:** srllib.h
dxxxlib.h
eclib.h

**Category:** I/O

**Mode:** synchronous/asynchronous

## ■ 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.

| Parameter | Description |
|---|---|
| **chDev** | The channel device handle obtained when the CSP-capable device is opened using **dx_open( )**. |
| **tptp** | Pointer to the DV_TPT table 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 Software Reference*.<br>***Note:*** On DM3 boards, all DV_TPT terminating conditions are supported except for DX_MAXTIME, DX_LCOFF, DX_PMON and DX_PMOFF. For more information on DV_TPT limitations on DM3 boards, see the *Compatibility Guide for the Dialogic R4 on DM3 Products*.<br>***Note:*** In CSP, DV_TPT terminating conditions are edge-sensitive. |
| **xpbp** | Pointer to the DX_XPB table 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 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)
```

where:

**chDev:** is the CSP channel on which streaming is performed

**buffer:** is the buffer that contains streamed data

**length:** is 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 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.
- On SpringWare boards, we recommend that you use the same data format for play and recording/streaming.
- To set the proper parameters, the **ec_setparm( )** 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.
- In Linux applications that use multiple threads, you must avoid having two or more threads call functions that send commands to the same channel; otherwise, the replies can be unpredictable and cause those functions to time out. If you must do this, use semaphores to prevent concurrent access to a particular channel.
- 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.

## ■ 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;              /* SCbus 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 */
int 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 dx_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 Dialogic 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);
}
```

## 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:

| Error code | Reason |
|---|---|
| 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. |

## See Also

- **ec_reciottdata( )**
- **dx_reciottdata( )**

# ec_unlisten( )

**Name:** int ec_unlisten(chDev)

**Inputs:** int chDev
- valid channel device handle

**Returns:** 0 for success
-1 for failure

**Includes:** srllib.h
dxxxlib.h
eclib.h

**Category:** routing

**Mode:** synchronous

## 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.

## 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 SCbus time slots */
   if (ec_unlisten(chdev) == -1) {
       printf("Error message = %s", ATDV_ERRMSGP(chdev));
       exit(1);
   }
}
```

## 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:

| Error code | Reason |
|---|---|
| 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. |

## See Also

- **ag_listen( )**
- **dt_listen( )**
- **dx_listen( )**

# Events 3

This chapter provides information on events that may 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, may 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 Programmer's Guide* (in the*Voice Software Reference*).

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( )**.

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_setevtmask( )**.

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 Software Reference*.

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_setevtmask( )**. This event is only generated when data is being recorded or streamed.

# Index

## F



## L



## M



## N



## P



## R



## S



## T



## U