Dialogic®

Application Note

Low-Density Audio Streaming with Voice Resources on Dialogic® Products

Executive Summary

This application note describes methods for streaming audio between the host and voice resources on Dialogic® boards in low-density environments. It is intended for solution designers and application developers who need to use such streaming functions for speech technology integration, host-based voice processing, or customized voice streaming needs.

Implementation details are provided for using “play-and-record” and “play-and-CSP” using Dialogic® Continuous Speech Processing Technology (CSP) API functions to accomplish the audio streaming. This application note also presents a detailed implementation for two methods of streaming audio between the telecom server (host) and voice resources on Dialogic boards. It provides implementation details and sample code to demonstrate a sample implementation of this feature.

Table of Contents

Introduction 2

Dialogic Continuous Speech Processing Technology 2

User-Defined I/O 3

Buffering 4

Streaming from the Host 5

Streaming to the Host with Voice Devices Based on Dialogic Continuous Speech Processing Technology 6

Streaming to the Host with Voice Devices Not Based on Dialogic Continuous Speech Processing Technology 8

Sample Code 10

Product List 10

Sample Code Download 10

Acronyms 11

For More Information 11

Introduction

Audio streaming between Dialogic boards and the host differs from traditional play and record functions in that it is a continuous source and sink of audio data. Also, the audio data is transferred using relatively small buffers. This method provides lower latency in starting and stopping the audio and is adaptable to VoIP applications.

Full-duplex streaming requires transferring buffers of audio data in both directions. Using very small buffers results in high interrupt rates, which limit the use of these techniques to low-density applications. The actual density limits depend upon the platform and the application.

Streaming from the host to voice resources on Dialogic boards is accomplished using the voice API dx_playiottdata() function call with user-defined input/output (I/O) functions enabled. This allows the application to take over the file I/O normally associated with play functions and instead access application-defined buffering.

Streaming from voice resources on Dialogic boards to the host can be accomplished in two ways. If the voice resources are Dialogic Continuous Speech Processing capable, the speech API ec_stream() function is used to establish a callback to the application that delivers the audio data. For voice resources that are not Dialogic Continuous Speech Processing capable, a separate device and the voice API dx_reciottdata() function call are used with user-defined I/O functions in streaming from the host to the board.

In all cases, the audio data is buffered between the source and the sink. The buffering can take many forms, such as managed, fixed-size buffers or circular buffers that are managed either first-in, first-out (FIFO) or by timestamps (for latency control). For the purpose of this application note, and for simplicity, the examples use a simple FIFO circular buffer.

Dialogic Continuous Speech Processing Technology

Dialogic Continuous Speech Processing Technology enhances existing speech technologies by providing board-level firmware that processes real-time voice signals to identify human speech input and present it to the host platform for speech recognition. The real-time functions include both echo cancellation and voice activity detection (VAD). This approach offloads host platform resources for more complex speech recognition tasks, such as analyzing and recognizing the speech input in support of the application.

User-Defined I/O

User-defined I/O is a mechanism that allows the application to take over what are normally file I/O operations for play and record. This is accomplished by enabling callback functions that mimic file I/O.

The voice API dx_setdevuio() function and DX_UIO structure are used to set the user-defined I/O function callbacks for read, write, and seek operations for a device. Once enabled, these I/O callback routines are used for all play and record operations. Alternatively, dx_setuio() function can be used to globally set the user-defined I/O routines. The semantics of these callback routines are the same as for file I/O. An example follows:

```
int uio_read(int fileh, char* bufferPtr, unsigned count) {
   int bytes_read;
   // Read count bytes from application buffering
   // and write to the buffer pointed to by bufferPtr.
   // Return the actual number of bytes transferred.
   // EOF signified if bytes_read != count.
   return bytes_read;
}

int uio_write(int fileh, char* bufferPtr, unsigned count) {
   int bytes_written;
   // Read count bytes from the buffer pointed to by bufferPtr
   // and write to the application buffering
   // Return the actual number of bytes transferred
   // EOF signified if bytes_written != count.
   return bytes written;
}

long uio_seek(int fileh, long offset, int whence) {
   // Simulate file seek I/O operation.
   // Return offset or -1 for an error.
   return offset;
}

int InitializeDeviceUio(int deviceHandle, DX_UIO* uioBlock) {
   uioBlock.u_read = uio_read;
   uioBlock.u_write = uio_write;
   uioBlock.u_seek = uio_seek;

   if (dx_setdevuio(deviceHandle, uioblock, 0) == -1) {
         // handle error
         return -1;
   }
   return 0;
}
```

The user-defined I/O routines include a parameter for a file handle, which is supplied to the driver when a play or record is initiated. In non-file-based I/O, this data element is used as an application context value. Its requirement is that it is unique for each concurrent voice resource operation.

Buffering

Several levels of buffering are involved with audio streaming:

- Buffering under application control
- Buffers in the firmware
- Buffers used to transfer between the driver and the firmware

Buffering is required within the application between the application layer audio source/sink and the voice/speech API interface. This can take a number of forms, depending on application requirements. For the purpose of this application note and the accompanying sample code, a FIFO circular buffer is used.

An application layer buffer object has at least the following interface:

- **int Read**(**char* ptr, const int cnt**) — Reads data from the buffer, where cnt bytes are read from the internal store and written to the location pointed to by ptr.
- **int Write**(**const char* ptr, const int cnt**) — Writes data to the buffer, where cnt bytes are written to the internal store from the location pointed to by ptr.
- **int Available**() — Returns the number of internal store bytes available to be read.
- **int Free**() — Returns the number of internal store bytes free that can be written.

There are other considerations for the firmware buffers and the buffers used to transfer data between the driver and the firmware. These buffer sizes are adjusted to meet audio streaming latency requirements and balanced against system resource utilization. The transfer buffer size must be at least twice the size of the firmware buffer. The default firmware buffer size is 512 bytes; the default transfer buffer size is 32K. Audio streaming applications typically use a smaller transfer buffer size (that is, 2048, 1024, or 512 bytes).

The transfer buffer size for user-defined I/O operations is adjusted by setting voice channel parameters as follows:

```
int SetTxXferBufferSize( int deviceHandle, int xferBufferSize) {
        int parmval = xferBufferSize;
        if (dx_setparm( deviceHandle,
                        DXCH_TXDATABUFSIZE,
                        (void *)&parmval) == -1) {
            // handle error
            return -1;
        }
        return 0;
}

int SetRxXferBufferSize( int deviceHandle, int xferBufferSize) {
        int parmval = xferBufferSize;
        if (dx_setparm( deviceHandle,
                        DXCH_RXDATABUFSIZE,
                        (void *)&parmval) == -1) {
            // handle error
            return -1;
        }
        return 0;
}
```

The transfer buffer size for Dialogic Continuous Speech Processing stream operations is adjusted by setting a Continuous Speech Processing channel parameter:

```
int SetCspXferBufferSize( int deviceHandle, int xferBufferSize) {
   int parmval = xferBufferSize;
   if (ec_setparm( deviceHandle,
                    ECCH_XFERBUFFERSIZE,
                    (void *)&parmval) == -1) {
         // handle error
         return -1;
   }
   return 0;
}
```

Streaming from the Host

Streaming audio data from the host to voice resources based on the Dialogic board is accomplished using the voice API dx_playiottdata() function with user-defined I/O. Normally, a play operation would begin playing from a file or memory buffer until a terminating condition occurs (for example, until end of file [EOF] or dual tone multi-frequency [DTMF] digits are detected). A continuous play of data from an application-managed buffer is required for audio streaming, ending only on a stop channel command.

The voice API dx_playiottdata() function call is used as a continuous streaming mechanism by specifying no termination conditions and always returning success from the user-defined I/O callback routines. An example follows:

```
int StartStreamFromHost(int deviceHandle, DX_IOTT* iott, DX_XPB* xpb)
{
   iott->io_fhandle = deviceHandle;
                           // application context token
   iott->io_type = IO_UIO|IO_DEV|IO_EOT;
   iott->io_offset = 0;
   iott->io_length = -1;
   iott->io_bufp = 0;
   iott->io_nextp = 0;
   iott->io_prevp = 0;

   xpb->wFileFormat = FILE_FORMAT_VOX;
   xpb->wDataFormat = DATA_FORMAT_MULAW;
   xpb->nSamplesPerSec = DRT_8KHZ;
   xpb->wBitsPerSample = 8;

   if (dx_playiottdata( deviceHandle,
                              iott,
                              0,
                              xpb,
                              EV_ASYNC) == -1) {
         // handle error
         return -1;
   }
   return 0;
}
```

Immediately after initiating the streaming, the driver begins making callbacks to the user-defined I/O uio_seek and uio_read routines. This continues until:

- **uio_seek** returns an error
- **uio_read** returns a value other than the number of bytes requested
- Or **dx_stopch**() is called for the device

The situation can arise where the uio_read routine is called and the application has no audio data immediately available. If that happens, audio data equating to silence or comfort level noise should be provided:

```
int uio_read(int fileh, char* bufferPtr, unsigned count) {
   // Assuming existence of an array of global buffer objects
   if( GlobalBuffer[ fileh] .HasData()) {
         int bytes_read;
         // copy data from the application buffer
         // and update butes_read
         return bytes_read;
   }
   memset(bufferPtr, 0xFF, count); // insert silence
   return count;
}
```

Streaming is stopped by issuing a dx_stopch() function call. A TDX_PLAY event is generated when the streaming stops:

```
int StopStream(int deviceHandle) {
   if (dx_stopch(deviceHandle, EV_ASYNC) == -1) {
         // handle error
         return -1;
   }
   return 0;
}
```

Streaming to the Host with Voice Devices Based on Dialogic Continuous Speech Processing Technology

Streaming audio data to the host from a Dialogic board with voice resources based on Dialogic Continuous Speech Processing Technology, or with Dialogic® Host Media Processing Software Release 1.1 for Windows® or later, is accomplished using the Dialogic Continuous Speech Processing API ec_stream() function. The ec_stream() function is used to establish a user-defined callback function that is called every time the driver fills the driver buffer with data. The application then copies the data to the application-provided buffering.

The VAD component of Dialogic Continuous Speech Processing Technology is used to generate events and control the start of audio streaming. These modes of operation are disabled for continuous streaming. (See the *Continuous Speech Processing API for Linux and Windows Operating System: Programming Guide* for more information.) An example follows:

```
    int InitializeCSPStreamingToHost(int deviceHandle) {
    int parmval;

    // disable barge-in
    parmval = 0;
    if (ec_setparm( deviceHandle,
                    DXCH_BARGEIN,
                    (void*)&parmval) == -1) {
          // handle error
          return -1;
    }

    // disable voice activation
    parmval = 0;
    if (ec_setparm( deviceHandle,
                    ECCH_VADINITIATED,
                    (void*)&parmval) == -1) {
          // handle error
          return -1;
    }
    return 0;
}
```

The ec_stream callback function is called when the driver fills the driver buffer with data. The application is responsible for copying the data and returning success:

```
int csp_stream(char* bufferPtr, unsigned count) {
   // Read count bytes from the buffer pointed to by bufferPtr
   // and write to the application buffering
   // Return 0 for success and -1 for failure
   return 0;
}
```

The ec_stream speech API call is used to initiate a continuous streaming mechanism by specifying no termination conditions and always returning success from the callback routine. (See the *Dialogic Continuous Speech Processing API Library Reference* for additional information and details on the ec_stream function and associated data structures.)
An example follows:

```
int StartCSPStreamToHost(int deviceHandle, DX_XPB* xpb, int
(*callback) (int, char*, uint)) {
   xpb->wFileFormat = FILE_FORMAT_VOX;
   xpb->wDataFormat = DATA_FORMAT_MULAW;
   xpb->nSamplesPerSec = DRT_8KHZ;
   xpb->wBitsPerSample = 9;

   if (dx_ec_stream( deviceHandle,
                      0,
                      xpb,
                      callback,
                      EV_ASYNC) == -1) {
         // handle error
         return -1;
   }
   return 0;
}
```

Streaming is stopped by issuing an ec_stopch() function call. A TEC_STREAM event is generated when the streaming stops:

```
int StopCSPStream(int deviceHandle) {
   if (dx_stopch(deviceHandle, RECEIVING, EV_ASYNC) == -1) {
         // handle error
         return -1;
   }
   return 0;
}
```

Streaming to the Host with Voice Devices Not Based on Dialogic Continuous Speech Processing Technology

Streaming audio data to the host from Dialogic boards with voice resources not based on Dialogic Continuous Speech Processing Technology is accomplished using the voice API dx_reciottdata() function with user-defined I/O. Normally, a record operation would begin recording to a file or memory buffer until some terminating condition occurs (for example, DTMF digits are detected, a maximum record time size is reached). A continuous record of data from an application-managed buffer is required for audio streaming, ending only on a stop channel command.

The voice API dx_reciottdata() function call is used as a continuous streaming mechanism by specifying no termination conditions and always returning success from the user-defined I/O callback routines.

```
int StartStreamToHost(int deviceHandle, DX_IOTT* iott, DX_XPB* xpb) {
   iott->io_fhandle = deviceHandle;
                       // application context token
   iott->io_type = IO_UIO|IO_DEV|IO_EOT;
   iott->io_offset = 0;
   iott->io_length = -1;
   iott->io_bufp = 0;
```

```
    iott->io_nextp = 0;
    iott->io_prevp = 0;

    xpb->wFileFormat = FILE_FORMAT_VOX;
    xpb->wDataFormat = DATA_FORMAT_MULAW;
    xpb->nSamplesPerSec = DRT_8KHZ;
    xpb->wBitsPerSample = 9;

    if (dx_reciottdata( deviceHandle,
                        iott,
                        0,
                        xpb,
                        EV_ASYNC) == -1) {
          // handle error
          return -1;
    }
    return 0;
}
```

After initiating the streaming, the driver begins making callbacks to the user-defined I/O uio_write routine and will continue until uio_write returns a value other than the number of bytes requested, or dx_stopch() is called for the device. The situation can arise where the uio_write routine is called and the application has no free buffer space immediately available. If that happens, the audio data must be discarded.

```
int uio_write(int fileh, char* bufferPtr, unsigned count) {
    // Assuming existence of an array of global buffer objects
    if( GlobalBuffer[ fileh] .HasFreeSpace()) {
          int bytes_written;
          // copy data to the application buffer
          // and update butes_read
          return bytes_read;
    }
    return count;
}
```

Streaming is stopped by issuing a dx_stopch() function call. A TDX_REC event is generated when the streaming stops. Sample code is shown in the “Streaming from the Host” section.

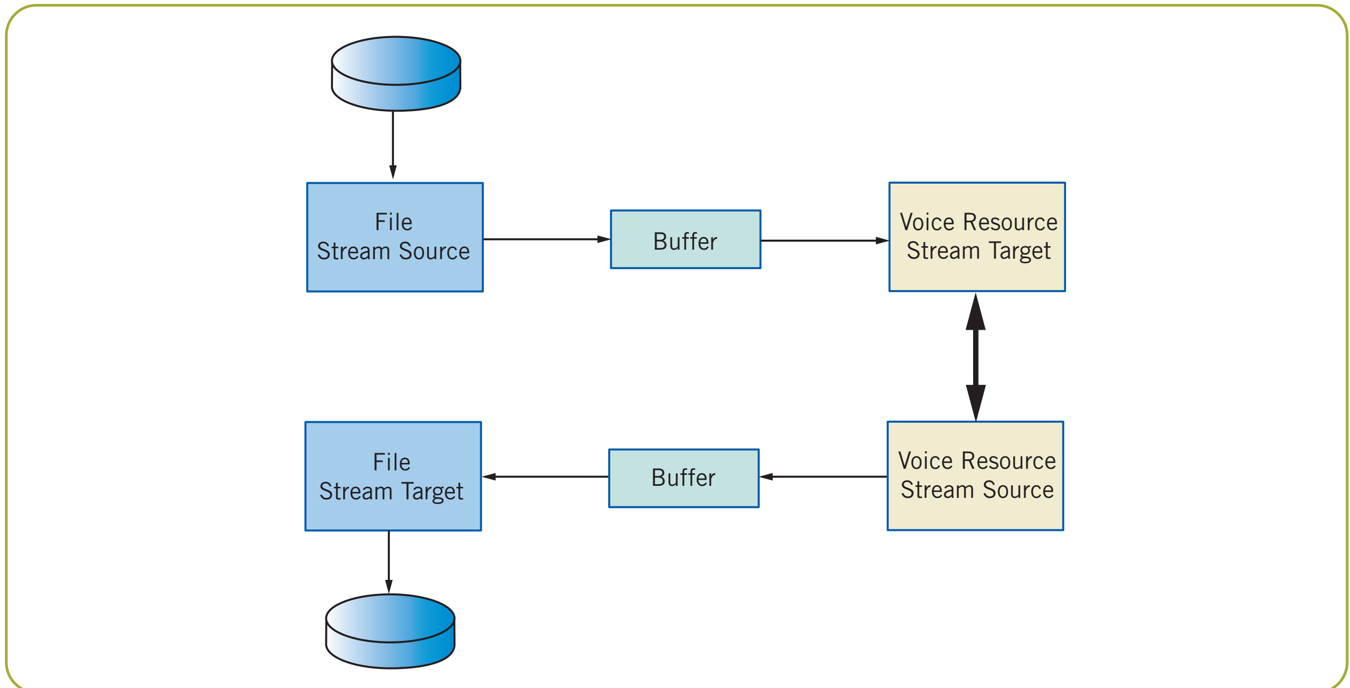

Figure 1. Audio Data Stream in Sample Applications

Sample Code

The sample code demonstrates audio streaming from the host by reading audio data from a file and streaming it to a voice resource connected to the SCbus. The reverse path is demonstrated by streaming the audio from a voice resource on the SCbus to the host and writing it to a second file. The results can be observed by playing the resulting file or examining it with an audio file editor. The audio data stream flow is shown in Figure 1.

The design of the sample code includes a buffer class, file stream source and target classes, a voice resource stream target class, and voice resource stream source classes that are both capable and not capable of Dialogic Continuous Speech Processing.

There are two example applications. One uses a pair of voice resources; the other uses a single voice resource that is capable of Dialogic Continuous Speech Processing.

The stream classes, provided as a buffer object when they are created, all have the same interface:

- **Open()** — Opens the underlying resources
- **Start()** — Starts streaming
- **Stop()** — Stops streaming
- **Close()** — Closes the underlying resources

Product List

This sample code was tested with the following Dialogic products. It is expected that the sample code will work with other Dialogic products that contain API functions that are both compatible and not compatible with Dialogic Continuous Speech Processing streaming:

- Dialogic® D/41JCT-LS Combined Media Board
- Dialogic® D/240JCT-T1 Combined Media Board
- Dialogic® System Release 6.0 PCI for Windows
- Dialogic® Host Media Processing Software Release 3.0 for Windows

Sample Code Download

The sample code in this application note may be downloaded at http://www.dialogic.com/goto/?10564

Acronyms

API	Application Programming Interface
CSP	Dialogic Continuous Speech Processing Technology
DTMF	Dual Tone Multi-frequency
EOF	End of File
FIFO	First-In, First-Out
I/O	Input/Output
IVR	Interactive Voice Response
UIO	User-defined Input/Output
VAD	Voice Activity Detection
VoIP	Voice over Internet Protocol

For More Information

Dialogic® D/240JCT-T1 Combined Media Board (http://www.dialogic.com/products/tdm_boards/media_processing/D240-300JCT_Boards.htm) — Provides one span of digital network interfaces in a H.100-compliant universal PCI form factor. Contains rich media features including voice processing, speech recognition software, fax, tone signaling, global tone detection, global tone generation, and call progress analysis.

Dialogic® D/41JCT-LS Combined Media Board (http://www.dialogic.com/products/tdm_boards/media_processing/D41JCT_Boards.htm) — Provides four telephone line interface circuits for direct connection to analog loop start lines.

Dialogic® System Release 6.x (http://www.dialogic.com/products/tdm_boards/system_release_software/default.htm)

Dialogic® Host Media Processing Software (http://www.dialogic.com/products/ip_enabled/HMPWindows.htm) — This software enables customers to build full-featured, scalable, and cost-effective software-only IP media servers for interactive voice response (IVR), voice mail, unified messaging, and conferencing. It offers flexibility in configuring media processing features and cost of ownership savings, because it only requires a standard computer without involving any proprietary hardware.

Continuous Speech Processing API for Linux and Windows Operating System: Programming Guide (http://www.dialogic.com/manuals/sr61win/default.htm)

Dialogic®

To learn more, visit our site on the World Wide Web at **http://www.dialogic.com**.

Dialogic Corporation
9800 Cavendish Blvd., 5th floor
Montreal, Quebec
CANADA H4M 2V9

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

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

Dialogic may make changes to specifications, product descriptions, and plans at any time, without notice.

Dialogic is a registered trademark of Dialogic Corporation. Dialogic’s trademarks may be used publicly only with permission from Dialogic. Such permission may only be granted by Dialogic’s legal department at 9800 Cavendish Blvd., 5th Floor, Montreal, Quebec, Canada H4M 2V9. Any authorized use of Dialogic’s trademarks will be subject to full respect of the trademark guidelines published by Dialogic from time to time and any use of Dialogic’s trademarks requires proper acknowledgement.

Windows is a registered trademark of Microsoft Corporation in the United States and/or other countries. Other names of actual companies and products mentioned herein are the trademarks of their respective owners. Dialogic encourages all users of its products to procure all necessary intellectual property licenses required to implement their concepts or applications, which licenses may vary from country to country.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Dialogic products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

Copyright © 2007 Dialogic Corporation All rights reserved.

06/07 9085-02