Channel Sorting with the EDMA

Introduction

In this chapter we are going to explore how to use a very powerful feature of the EDMA called Channel Sorting. We are going to start with the code that we wrote in the previous chapters and see how to use some of the other capabilities of the EDMA to sort data. These capabilities can be used for many other types of transfers, as we will see.

Outline

Outline

◆ Background: More EDMA Examples
◆ Packed Data vs Sorted Data
◆ EDMA Channel Sorting
◆ Counter Reload
◆ Channel Sorting Procedure
◆ Using BSL
◆ Exercise
◆ Lab 7
Chapter Topics

Channel Sorting with the EDMA ................................................................. 7-1

Chapter Topics ......................................................................................... 7-2

More EDMA Examples .............................................................................. 7-3

Packed Data vs. Sorted Data ..................................................................... 7-6

EDMA Channel Sorting ........................................................................... 7-9

Counter Reload ....................................................................................... 7-13

Channel Sorting Configuration ............................................................... 7-17

Using Board Support Library (BSL) ........................................................ 7-18

Exercise .................................................................................................... 7-19

Lab 7 ........................................................................................................ 7-23

Part A ....................................................................................................... 7-28

Multiple Channels (Optional) ................................................................. 7-29
More EDMA Examples

Let's start out by reviewing what we did back in the EDMA chapter.

### Single-Frame Transfer (Review)

**Goal:**
Transfer 4 elements from loc_8 to myDest

<table>
<thead>
<tr>
<th>Addr Update Mode (SUM/DUM)</th>
<th>ESIZE</th>
<th>SUM</th>
<th>DUM</th>
<th>FS</th>
</tr>
</thead>
<tbody>
<tr>
<td>00: fixed (no modification)</td>
<td>00: 32-bits</td>
<td>01: 16-bits</td>
<td>10: 8-bits</td>
<td>11: rsvd</td>
</tr>
<tr>
<td>01: inc by element size</td>
<td>Frame Sync: 0: Off</td>
<td>1: On</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: dec by element size</td>
<td>00: fixed (no modification)</td>
<td>01: inc by element size</td>
<td>10: dec by element size</td>
<td>11: index</td>
</tr>
<tr>
<td>11: index</td>
<td>ESIZE: 00: 32-bits</td>
<td>01: 16-bits</td>
<td>10: 8-bits</td>
<td>11: rsvd</td>
</tr>
</tbody>
</table>

Here is the same type of example using the indexing capability of the EDMA.

### Indexed Single Frame Transfer

**Procedure**
- Source & Dest Addr
- Transfer Count
- Element Size
- Increment src/dest Frame Sync

<table>
<thead>
<tr>
<th>Addr Update Mode (SUM/DUM)</th>
<th>ESIZE</th>
<th>SUM</th>
<th>DUM</th>
<th>FS</th>
</tr>
</thead>
<tbody>
<tr>
<td>00: fixed (no modification)</td>
<td>00: 32-bits</td>
<td>01: 16-bits</td>
<td>10: 8-bits</td>
<td>11: rsvd</td>
</tr>
<tr>
<td>01: inc by element size</td>
<td>Frame Sync: 0: Off</td>
<td>1: On</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: dec by element size</td>
<td>00: fixed (no modification)</td>
<td>01: inc by element size</td>
<td>10: dec by element size</td>
<td>11: index</td>
</tr>
<tr>
<td>11: index</td>
<td>ESIZE: 00: 32-bits</td>
<td>01: 16-bits</td>
<td>10: 8-bits</td>
<td>11: rsvd</td>
</tr>
</tbody>
</table>

8-bit Pixels

<table>
<thead>
<tr>
<th>Codec:</th>
<th>Codec</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRC</td>
<td>Codec</td>
</tr>
<tr>
<td>DUM</td>
<td>8 bits</td>
</tr>
</tbody>
</table>

Codec: 8 bits

<table>
<thead>
<tr>
<th>Codec</th>
<th>Codec</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRC</td>
<td>Codec</td>
</tr>
<tr>
<td>DUM</td>
<td>8 bits</td>
</tr>
</tbody>
</table>

8-bit Pixels

00: fixed (no modification) | 00: 32-bits | 01: 16-bits | 10: 8-bits | 11: rsvd |
| Frame Sync: 0: Off | 1: On |

8-bit Pixels

00: fixed (no modification) | 00: 32-bits | 01: 16-bits | 10: 8-bits | 11: rsvd |
| Frame Sync: 0: Off | 1: On |

8-bit Pixels

00: fixed (no modification) | 00: 32-bits | 01: 16-bits | 10: 8-bits | 11: rsvd |
| Frame Sync: 0: Off | 1: On |
As you can see, we simply change the update mode of the source to use and index, and fill in the index register with the appropriate value. Note that this value is in bytes.

We used an element index above. To move blocks of data, you may need a frame index as well.
The frame index allows you to modify the address after each frame. This capability is one of the primary enablers to channel sorting with the EDMA.

### Multi-Frame (Block) Transfer

**Procedure**
- Source & Dest Addr
- Transfer Count
- Element Size
- Increment src/dest Frame Sync
- Index

<table>
<thead>
<tr>
<th>16-bit Pixels</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6</td>
</tr>
<tr>
<td>7 8 9 10 11 12</td>
</tr>
<tr>
<td>13 14 15 16 17 18</td>
</tr>
<tr>
<td>19 20 21 22 23 24</td>
</tr>
<tr>
<td>25 26 27 28 29 30</td>
</tr>
<tr>
<td>31 32 33 34 35 36</td>
</tr>
</tbody>
</table>

**Codec:**
- 16 bits

**Options**
- Source
- Transfer Count
- Destination
- Index
- Cnt Reload
- Link Addr

<table>
<thead>
<tr>
<th>ESIZE</th>
<th>SUM</th>
<th>DUM</th>
<th>FS</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>11</td>
<td>00</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># Frames</th>
<th># Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Frame Index (bytes)</th>
<th>Element Index (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

Here's a more detailed explanation of how to calculate the frame index. One important thing to remember is that the index register treats everything as bytes.

### Why Does Frame Index = 6 bytes

The frame index calculation involves understanding the byte alignment and indexing within the EDMA context. The diagram illustrates the sequence of frames and the indexing process in detail.
In order to understand what channel sorting is, we need to understand the different ways that data can come in to a system. Data is packed if multiple channels (L and R) are next to each other, or packed, into memory.

- After A/D conversion, the AIC23 shifts out data from alternating channels:
  - Left, then
  - Right
- This leaves data **packed** in memory. (You might also say it’s *interleaved* in memory.)

The AIC23 codec has been sending us packed data up until this point.
Sorted data is separated out into buffers which contain data for only one channel. So, you would have one buffer full of left data, and one buffer full of right data. Are there any advantages to this approach? Most people would say yes. When the data is sorted, you can write your algorithms so that they simply process a buffer. If you want to add another channel, you simply call the algorithm again with a new buffer of data. If the data is packed, the algorithm would have to be specific to the way the data is organized, and therefore less flexible.

- Sorting data splits data up by Left or Right channel
- Often, this is called **Channel Sorting**
Given the advantages of sorted data, how do we do it efficiently? We could do it with the CPU, but that takes valuable time.

- You could use the CPU to sort data, or
- It is more efficient to use the EDMA to sort the data

Why not do it with the EDMA as it is moving the data from the serial port? It has to do this anyway, and it doesn't take any time away from the CPU. So, how do we set this up?
**EDMA Channel Sorting**

In order to have the EDMA sort data, we need to re-think how we do our transfers. Instead of thinking of the data as a continuous stream, we need to think of it as $M$ frames of $N$ elements of data. Each frame is a collection of the corresponding elements of each channel. For example, the $0^{th}$ frame is all of the $0^{th}$ elements from each channel. So, how many frames do we need? We need one for each channel.

### How Channel Sorting Works

<table>
<thead>
<tr>
<th>Frame</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 (=10-1)</td>
<td>2</td>
</tr>
</tbody>
</table>

Given:
- Two buffers: Left, Right
- Buffers each 10 elements long
- ESIZE = 16-bits

EDMA setup:
- To sort L/R data, we need to set up EDMA with 10 frames, each with 2 elements

In the example above, there are 2 channels of data and we want to grab 10 samples from each channel. So, we have 10 frames of 2 elements each.
Now we need to figure out how to modify the addresses after each transfer. If each element is 2 bytes wide, how many bytes do we need to add to the address after transferring the first element to transfer the second to the right place?

### How Channel Sorting Works

<table>
<thead>
<tr>
<th>Frame</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>9 (=10-1)</td>
</tr>
<tr>
<td>Index</td>
<td>20</td>
</tr>
<tr>
<td>Source</td>
<td>McBSP</td>
</tr>
<tr>
<td>Destination</td>
<td>Left</td>
</tr>
</tbody>
</table>

Given:
- Two buffers: Left, Right
- Buffers each 10 elements long
- ESIZE = 16-bits

EDMA setup:
- To sort L/R data, we need to set up EDMA with 10 frames, each with 2 elements

Well, if there are 10 2 byte elements, we need to add 20 bytes. Take a closer look at the example above. When we write the first element to the Left channel, we need to move down to the first element of the Right channel. If the address of the first element in the Left channel is 0 and it has 10 2 byte elements, then the address of the first element of the Right channel is 20 (don't forget that addresses on the C6000 are in bytes). So, we need to skip from 0 to 20 between elements in a frame. That's why the element index above is set to 20.
Now the question becomes, what do we need to do to the addresses after we transfer the first element of the Right channel? We need to go back up in memory to the second element of the Left channel. After each frame, we need to go back up. How can we do this?

![Diagram of how channel sorting works]

- **Given:**
  - Two buffers: Left, Right
  - Buffers each 10 elements long
  - ESIZE = 16-bits

- **EDMA setup:**
  - To sort L/R data, we need to set up EDMA with 10 frames, each with 2 elements

- **Table:**

<table>
<thead>
<tr>
<th>Frame</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>9 (=10-1)</td>
</tr>
<tr>
<td>Index</td>
<td>-18</td>
</tr>
<tr>
<td>Source</td>
<td>McBSP</td>
</tr>
<tr>
<td>Destination</td>
<td>Left</td>
</tr>
</tbody>
</table>

- **Question:** How many bytes to go back to Left[2]?

We can use the frame index to move us back to the Left channel. So, if the starting address of the Right channel is 20, and the second element of the Left channel is at 2, we need to go back (the value is negative) by 18.
Here's a summary of the values and how we got to them. Don't forget that the addresses have to be normalized to bytes before the indexes are calculated.

### How Channel Sorting Works

<table>
<thead>
<tr>
<th>Frame Count</th>
<th>Element Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Frame Index</th>
<th>Element Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>-18</td>
<td>20</td>
</tr>
</tbody>
</table>

Forward 10 to next element
Back 9 to next frame

2 bytes
Counter Reload

When the EDMA transfers a frame of data, the element count goes to 0. It needs a place to remember how many elements are in a frame. In this topic, we'll look at how this is done.

Notice how the element count goes to 1 after the first transfer.
After the second transfer (or the last element transfer in a frame) the element count sits at 0.

What happens when the element count goes to zero? 
There's a register for this

When setting up the EDMA transfer parameters, the "Count Reload" field can be set to the same value as the original element count. Then the element count can be reloaded before the next frame transfer. This allows the EDMA to keep up with the number of elements in each frame.
This process of reloading the element count after each frame is transferred repeats over and over until the frame count goes to 0.

What happens when the element count goes to zero? There's a register for this.
What happens when the element count goes to zero?

There's a register for this.
Channel Sorting Configuration

Here is a simple outline to follow when you want to implement channel sorting with the EDMA (i.e. this may be good info. to refer back to in the lab).

**Channel Sorting Configuration**

To enable EDMA channel sorting, reconfigure the EDMA as shown below:

<table>
<thead>
<tr>
<th>Options:</th>
<th>DUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source:</td>
<td></td>
</tr>
<tr>
<td>Transfer Count:</td>
<td>BUFFSIZE - 1</td>
</tr>
<tr>
<td>Destination:</td>
<td>1st Buffer's Address</td>
</tr>
<tr>
<td>Index:</td>
<td>- (BUFFSIZE -1) * NBYTES</td>
</tr>
<tr>
<td>Count Reload / Link:</td>
<td># of Buffers</td>
</tr>
</tbody>
</table>

- **NBYES** = # of bytes per element
- **Destination Update Mode (DUM):**
  - 00: fixed (no modification)
  - 01: inc by element size
  - 10: dec by element size
  - 11: index

**Channel Sorting Configuration**

Provided:
- Two buffers, each of "BUFFSIZE" number of elements
- Each element consists of "NBYTES"
- Buffers follow one after the other in memory

Calculate:
- Element Count = # of Buffers
- Frame Count = BUFFSIZE - 1
- Element Index = BUFFSIZE * NBYTES
- Frame Index = - (BUFFSIZE * NBYTES) + NBYTES

From our previous "How Sorting Works" example:

BUFFSIZE = 10
NBYTES = 2

Therefore:
- Elem Count = 2
- Frame Count = 10 - 1 = 9
- Element Idx = 10 * 2 = 20
- Frame Idx = -(10*2) + 2 = -18

**Note:** For the channel sorting configuration described here to work properly, the two buffers must be aligned properly and contiguous in memory. In ANSI C, declaring two arrays one after the other does not necessarily guarantee they will be contiguous, though if you look at the map file created during the lab exercises, you will see that by "luck" they are contiguous.
Using Board Support Library (BSL)

The DSKs come with a very helpful set of functions to access all of their capabilities. These functions are organized into a library for each board.

### Board Support Library

- **Board Support Library (BSL)**
  - Board-level routines supporting DSK-specific hardware
  - Higher level of abstraction than CSL
  - BSL functions make use of CSL

- **Codec**
- **Leds**
- **Switches**
- **Flash**

### Chip Support Library (CSL)

Low-level routines supporting on-chip peripherals

- **Serial Ports**
- **EDMA**
- **EMIF**
- **Cache**
- **Timers**
- **Etc.**

Here are the three quick steps necessary to use a module in the BSL.

### Interfacing with the DSK’s DIP Switches

1. **Add these include files:**
   ```
   #include <dsk6713.h>
   #include <dsk6713_dip.h>
   ```

2. **Add this library to your project:**
   ```
   C:\CCStudio_v3.1\c6000\dsk6713\lib\dsk6713bsl.lib
   ```

3. **Use the DIP_get API to read the DSK switches (0-3):**
   ```
   if (DSK6713_DIP_get(0) == 0){
       mySample = 0;
   }
   ```

<table>
<thead>
<tr>
<th>Switch Position</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Down</td>
<td>0</td>
</tr>
<tr>
<td>Up</td>
<td>1</td>
</tr>
</tbody>
</table>

Note: If you’re using the 6416 DSK, just change 6713 to 6416.
Exercise

Exercise: Background

- Update the destination EDMA configuration for channel sorting. This exercise should take 10 minutes.
- These are the data declarations and references used in Lab 7:

```c
// ======== Declarations ========
#define BUFFSIZE 32

// ======== References ========
extern short gBufRcvL[BUFFSIZE];
extern short gBufRcvR[BUFFSIZE];
extern short gBufXmtL[BUFFSIZE];
extern short gBufXmtR[BUFFSIZE];
extern SINE_Obj sineObjL;
extern SINE_Obj sineObjR;

// ======== Global Variables ========
EDMA_Handle hEdmaRcv;
EDMA_Handle hEdmaReloadRcv;
EDMA_Handle hEdmaXmt;
EDMA_Handle hEdmaReloadXmt;
short gXmtTCC;
short gRcvTCC;
```

Exercise: Step 1

Modify the configuration from our previous lab exercise:

```c
EDMA_Config gEdmaConfigRcv = {
    EDMA_OPT_RMK(
        EDMA_OPT_PRI_LOW,       // Priority?
        EDMA_OPT_ESIZE_16BIT,   // Element size?
        EDMA_OPT_2DS_NO,        // 2 dimensional source?
        EDMA_OPT_SUM_NONE,      // Src update mode?
        EDMA_OPT_2DD_NO,        // 2 dimensional dest?
        EDMA_OPT_DUM_INC,       // Dest update mode?
        EDMA_OPT_TCINT_YES,     // Cause EDMA interrupt?
        EDMA_OPT_TCC_OF(0),     // Transfer complete code?
        EDMA_OPT_LINK_YES,      // Enable link parameters?
        EDMA_OPT_FS_NO          // Use frame sync?
    ),
    ...
};
```
Exercise: Steps 2-3

- Using the declarations and variables from the previous slide, fill in the correct values. Use the symbol `BUFFSIZE` rather than just the value, in case we change the buffer size later.
- Refer back to page 7-17 for hints on how to fill in the blanks.

2. Set Transfer Counter:

   ```
   EDMA_CNT_RMK(
       EDMA_CNT_FRMCNT_OF(),
       EDMA_CNT_ELECNT_OF()
   ),
   ```

3. Set Destination to first buffer’s address

   ```
   EDMA_DST_OF()
   ```

Exercise: Step 4

- Using the declarations and variables from the previous slide, fill in the correct values. Use the symbol `BUFFSIZE` rather than just the value, in case we change the buffer size later.
- Refer back to page 7-17 for hints on how to fill in the blanks.

Set Index register:

4. `EDMA_IDX_RMK`

   ```
   // Negative Frame Index to move us back to the previous channel
   EDMA_IDX_FRMIDX_OF(),
   
   // Positive Element Index to move us to the next channel
   EDMA_IDX_ELEIDX_OF()
   ```
Exercise: Step 5

Element Reload:

```c
EDMA_RLD_RMK(
    // Number of elements, should be the same as Element Count
    EDMA_RLD_ELERLD_OF( ),
    // We'll replace "0" later using EDMA_link()
    EDMA_RLD_LINK_OF(0)
)
```

Exercise: Step 6

Complete the “if” condition below using BSL:
If DIP switch 0 is on (down), then add sine-wave values to the Left and Right receive buffers

```c
if ( )
{
    SINE_add(&sineObjL, gBufRcvL, BUFFSIZE);
    SINE_add(&sineObjR, gBufRcvR, BUFFSIZE);
}
```
*** another place to stare for no reason at all ***
Lab 7

In this lab, we are going to set up the EDMA to sort the packed left/right data stream into separate buffers of all left data and right data.

**Copy Files and Rename the Project**

1. **Copy Lab6 folder to the audioapp folder**
   
   Because lab65 used completely different code than we’ve been building up, we want to revert back to our solution for lab6 as a starting point. In the `c:\iw6000\labs` folder, delete the `\audioapp` folder. Right-click on your lab6 solution and select copy. Move your mouse to an open spot in the `\labs` folder, right click and choose paste. You will now have a “copy of” the lab6 folder. Rename the folder to audioapp. You now have your lab6 code as a base for beginning this lab.

2. **Reset the DSK and start CCS**

3. **Open audioapp.pjt**
Modify Buffers

We currently have a receive and transmit buffer for the packed left/right data. In order to sort this data into separate buffers of left data and right data, we need to add two new buffers. We will use the current buffers for the left channel, and the two new buffers for the right channel.

4. In main.c, create a new receive buffer

Find the place where we create the two current buffers. Copy and paste the gBufRcv buffer. Make sure to paste it immediately below itself.

5. Rename the buffers

Name the first receive buffer, gBufRcvL, and the second gBufRcvR.

Note: The order in which the buffers are declared is important. The XmtL/XmtR buffers need to be declared together (left, the right) followed by the Rcv buffers (L then R) AND be contiguous.

6. Create and rename the transmit buffers

Repeat the same process for the transmit buffers.

7. Modify the for ( ) loop in main to initialize both transmit buffers

Find the place in main( ) where we initialize the transmit buffer to zero. Modify this loop to initialize both the left and right transmit buffers.

These are all of the changes that we need to make to main.c.

Set Up the EDMA for Channel Sorting

8. In edma.c, change the buffer references

At the beginning of edma.c, there should be two references to the global buffers, gBufRcv and gBufXmt. Change these references to reflect the modifications that we made earlier in main.c.

9. Make sure to changes all instances of the buffer names

We need to make sure to change all instances of gBufRcv and gBufXmt to gBufRcvL and gBufXmtL, respectively. Make this change in edma.c (there should not be any changes anywhere else).
10. Modify the EDMA receive configuration structure

Find the EDMA configuration structure for the receive channel. We need to modify this structure so that it sorts the left and right channels. This list should help you follow the 6-step channel sorting procedure that we discussed:

1. Calculate the values needed to do channel sorting in the lab.
2. Change the destination update mode (DUM) to use an index.
3. We need to change the CNT register. Instead of a single frame transfer with BUFFSIZE number of elements, we need to make BUFFSIZE frame transfers with 2 elements per frame (left and right). In order to make this change and fill in both fields, we will need to use an RMK macro like this:

   ```
   EDMA_CNT_RMK(
               EDMA_CNT_FRMCNT_OF(),
               EDMA_CNT_ELECNT_OF()
   ),
   ```

   This macro will build the correct values and put them in the right place in the register.

   **Hint:** Don't forget that the value that goes in the FRMCNT field is supposed to be NUMFRAMES – 1.

4. We need to change the destination to gBufRcvL.
5. We need to modify the IDX register as well. You will need to use a RMK macro just like you did for the CNT register. The two fields are:
   - FRMIDX – a negative value to move you back to the correct place in the previous buffer
   - ELEIDX – a positive value to take you to the correct place in the next buffer

   **Hint:** Refer back to the discussion material to help you figure out what these values should be. Don't forget that the constant BUFFSIZE represents the number of elements per buffer.

6. The last modification that we need to make is to the RLD register. Since we are doing a synchronized, frame indexed transfer, we need to fill in the element count reload field of the RLD register. You'll need to use an RMK macro again like you did before and here are the fields:
   - ELERLD - The number that you would like reloaded into the element count field after each frame completes.
   - LINK - The set of reload registers to link to. We do this in code later.

11. Apply EDMA configuration changes to the transmit side.

   Does the transmit side get the same changes as the receive side?

   Apply any changes that you feel need to be applied to the transmit side (very few).

12. Build your code and fix any errors. If you get a clean build, move on.
Adding the Sine Wave to Both Channels

Now that we have made the necessary changes to the EDMA code to sort the data, what changes need to be made to how we process that data? What has fundamentally changed?

13. Add a second SINE_Obj to main.c

Now that the data is sorted into two separate channels, we need to change how we are going to add the sine wave to it. To do this, let's create a new instance of the sine generator to add the sine wave to the right channel.

Find the place in main.c where we created the SINE_Obj that we have been using up to this point. Copy this code to create two SINE_Obj's. Name one sineObjL and the other sineObjR.

14. Call SINE_init( ) for both SINE_Obj's

Find the place in main( ) where we call SINE_init( ). You'll need to call this function for both of the SINE_Obj's that you just created.

15. Add external references for both SINE_Obj's to edma.c

We'll be using the two SINE_Obj's that we created earlier in edma.c. So, we need to add external references for them.

16. Change the way we add the sine wave to the buffers

Find the place where we add the sine wave to the audio in edmaHwi( ) in edma.c. The function that we used before to add the sine wave to the audio stream, SINE_addPacked( ), assumed that our data is packed (left/right, left/right, etc.). Since the data is now sorted, we need to change how the sine wave is added so that it is added to each channel's buffers separately. We have provided a function that does this for you called SINE_add( ) and is located in sine.c.

Change the call to SINE_addPacked ( ) to two calls SINE_add( ). The SINE_add( ) function needs to be called twice, once for each buffer (left and right). It takes three arguments, so make sure to add it properly.

17. Change how the data is copied

Now that we have two separate buffers of data, left and right, we also need to change how it gets copied. We'll use copyData( ) for both the left and right channels. Make this change to edmaHwi( ).
Run Audio

18. Run the audio

Make sure that the audio on the computer or whatever source you are using is still playing.

Build and Run

19. Build the project and load it to the DSK

20. Run the code (be prepared for minor disappointment)

Does everything sound OK? Very close. Mute the audio on the PC and listen to the sinewave. It’s not quite right. We have a small problem with our application that we need to fix. Something that we changed in this lab broke our application. What did we do? Well, we basically doubled the amount of data that we need to process. In lab 6, with a buffer size of 32 samples, we needed to generate 16 sine samples per buffer because we basically added the same sine sample to both the left and right channels. The data was packed together.

In the current lab, we are treating left and right as two separate channels. So, with a buffer size of 32 per channel, we are generating a total of 64 sine samples. This is taking too much time. There are three ways that we can fix this problem:

- Decrease the amount of data to process (reduce buffer size)
- Decrease the amount of time needed to process the data (optimize the code)
- Allow more time for processing (add more buffers, next chapter)

Let's try to do the first one with this lab. We know the code worked in the previous lab, so let's make the two equivalent to see if the application still works. What buffer size for the current lab would cause us to generate the same amount of sine values that we did back in chapter 6, 16 sine samples with one per channel?

21. Change the buffer size to 8 (8 samples * 2 channels = 16 samples)

Find the definition of BUFFSIZE in both main.c and edma.c. Change this from 32 to 8 in BOTH files.

22. Rebuild, re-load, and run your code

Your code should now work fine. If it doesn't sound right, go back and debug the code that you added in this lab that does the channelization of the left and right channels. Follow the data from the input/receive side to the output/transmit side. If you get frustrated, ask your instructor for help.

23. Halt the processor
Part A

Note: If you had troubles getting Lab 7 to work, copy the files from `\solutions for c64x\lab7` or `\solutions for c67x\lab7` and begin working on the next step shown below.

Add a switch to turn on/off the sine wave

Some of the functions of the DSK boards are controlled by APIs that are found in the Board Support Library (BSL) for that board. These might control things like dip switches, LEDs, etc. We're going to follow the 3-step procedure that we outlined in the discussion material. We're going to use a very simple API to check the position of a specific dip switch. If it is “on”, the sine wave will be added to the audio. If it is “off”, the audio will be undisturbed. First, we'll add the header files, then the library, then make a call to the proper API.

24. BSL Step 1, include the necessary header files to your code in edma.c:
   ```
   <dsk6416.h>, <dsk6416_dip.h> or <dsk6713.h>, <dsk6713_dip.h>
   ```

25. BSL Step 2, add one of the following libraries to your project:
   ```
   C:\CCStudio_v3.1\c6000\dsk6416\lib\dsk6416bsl.lib
   
   or
   
   C:\CCStudio_v3.1\c6000\dsk6713\lib\dsk6713bsl.lib
   ```

26. BSL Step 3, add the dip switch code to edmaHwi( )
   ```
   if (DSK6416_DIP_get(0) == 0)   or  if (DSK6713_DIP_get(0) == 0)
   SINE_add(…)     SINE_add(…)
   ```

   There are 4 dip switches on the DSK (near the LEDs). _0 is the switch farthest away from the LEDs. DIP_get simply reads the position: up is 1, down is 0. Using BSL is a quick way to add functionality to the DSK board without writing your own routines.

27. Add search path for BSL libraries
   In order for CCS to find the BSL libraries, we need to add a search path. Under Project -> Build Options, click on the Preprocessor category and add the following include search path:
   ```
   c:\ccstudio_v3.1\c6000\dsk6416\include -or- c:\ccstudio_v3.1\c6000\dsk6713\include
   ```

28. Build, Run, Debug

29. Try switching the sine wave on and off...

28. Copy project to preserve your solution.
   Using Windows Explorer, copy the contents of:
   ```
   c:\iw6000\labs\audioapp\*.*  TO  c:\iw6000\labs\lab7
   ```

You're done
Multiple Channels (Optional)

Channel Sorting and the McBSP

- McBSP’s *Multi-Channel* mode
  - E1 example
- Channel sorting multi-channel data from the McBSP

Multi-channel Operation

- Allows multiple channels (words) to be independently selected for transmit and receive
- Combined with the DMA’s flexibility...
EDMA’s flexible (indexed) addressing allows it to sort each channel into separate buffers!
## Discussion Solutions

### Indexed Single Frame Transfer

**Procedure**
- Source & Dest Addr
- Transfer Count
- Element Size
- Increment src/dest
- Frame Sync

8-bit Pixels

<table>
<thead>
<tr>
<th>Source &amp; Dest Addr</th>
<th>Transfer Count</th>
<th>Element Size</th>
<th>Increment src/dest</th>
<th>Frame Sync</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 8 9 10 11 12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13 14 15 16 17 18</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19 20 21 22 23 24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25 26 27 28 29 30</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31 32 33 34 35 36</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Src: mem_8)

**Addr Update Mode (SUM/DUM)**
- Options
  - Source
  - Transfer Count
  - Destination
  - Index

**ESIZE**
- 00: fixed (no modification)
- 01: inc by element size
- 10: dec by element size
- 11: index

**FS**
- 0: Off
- 1: On

**Codec**
- 8 bits

**# Elements**
- 4

**# Frames**
- 0

**ESIZE**
- 00: 32-bits
- 01: 16-bits
- 10: 8-bits
- 11: rsvd

**Addr Update Mode (SUM/DUM)**
- Options
  - Source
  - Transfer Count
  - Destination
  - Index

**ESIZE**
- 00: 32-bits
- 01: 16-bits
- 10: 8-bits
- 11: rsvd

**FS**
- 0: Off
- 1: On

**Codec**
- 16 bits

**# Elements**
- 4

**# Frames**
- 0

---

### Multi-Frame (Block) Transfer

**Procedure**
- Source & Dest Addr
- Transfer Count
- Element Size
- Increment src/dest
- Frame Sync

16-bit Pixels

<table>
<thead>
<tr>
<th>Source &amp; Dest Addr</th>
<th>Transfer Count</th>
<th>Element Size</th>
<th>Increment src/dest</th>
<th>Frame Sync</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 8 9 10 11 12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13 14 15 16 17 18</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19 20 21 22 23 24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25 26 27 28 29 30</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31 32 33 34 35 36</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Addr Update Mode (SUM/DUM)**
- Options
  - Source
  - Transfer Count
  - Destination
  - Index

**ESIZE**
- 00: fixed (no modification)
- 01: inc by element size
- 10: dec by element size
- 11: index

**FS**
- 0: Off
- 1: On

**Codec**
- 16 bits

**# Elements**
- 4

**# Frames**
- 0

---

**Codec**
- 16 bits

**# Elements**
- 4

**# Frames**
- 0

---

**Source & Dest Addr**
- Transfer Count
- Element Size
- Increment src/dest
- Frame Sync
Exercise: Step 1

Modify the configuration from our previous lab exercise:

```c
EDMA_Config gEdmaConfig = {
    EDMA_OPT_RMK(
        EDMA_OPT_PRI_LOW,       // Priority?
        EDMA_OPT_ESIZE_16BIT,   // Element size?
        EDMA_OPT_2DS_NO,        // 2 dimensional source?
        EDMA_OPT_SUM_NONE,      // Src update mode?
        EDMA_OPT_2DD_NO,        // 2 dimensional dest?
        EDMA_OPT_DUM_INC,       // Dest update mode?
        EDMA_OPT_TCINT_YES,     // Cause EDMA interrupt?
        EDMA_OPT_TCC_OF(0),     // Transfer complete code?
        EDMA_OPT_LINK_YES,      // Enable link parameters?
        EDMA_OPT_FS_NO          // Use frame sync?
    ),
    ...,
};
```

Exercise: Steps 2-3

- Using the declarations and variables from the previous slide, fill in the correct values. Use the symbol `BUFFSIZE` rather than just the value, in case we change the buffer size later.
- Refer back to page 7-17 for a hints on how to fill in the blanks.

2. Set Transfer Counter:

   ```c
   EDMA_CNT_RMK(
       EDMA_CNT_FRMCNT_OF( BUFFSIZE - 1 ),
       EDMA_CNT_ELECNT_OF( 2 )
   ),
   ```

3. Set Destination to first buffer’s address

   ```c
   EDMA_DST_OF( gBufRcvL ),
   ```
Exercise: Step 4

- Using the declarations and variables from the previous slide, fill in the correct values. Use the symbol `BUFFSIZE` rather than just the value, in case we change the buffer size later.
- Refer back to page 7-17 for hints on how to fill in the blanks.

Set Index register:

4. `EDMA_IDX_RMK`

    // Negative Frame Index to move us back to the previous channel
    `EDMA_IDX_FRMIDX_OF( -(BUFFSIZE*2) + 2 ),`

    // Positive Element Index to move us to the next channel
    `EDMA_IDX_ELEIDX_OF( BUFFSIZE * 2 ),`

Exercise: Step 5

5. Element Reload:

    `EDMA_RLD_RMK`

    // Number of elements, should be the same as Element Count
    `EDMA_RLD_ELERLD_OF( 2 ),`

    // We'll replace “0” later using EDMA_link()
    `EDMA_RLD_LINK_OF(0)`,

Exercise: Step 6

Complete the “if” condition below using BSL:
If DIP switch 0 is on (down), then add sine-wave values to the Left and Right receive buffers

```c
if ( DSK6713_DIP_get(0) == 0 )
{
    SINE_add(&sineObjL, gBufRcvL, BUFFSIZE);
    SINE_add(&sineObjR, gBufRcvR, BUFFSIZE);
}
```

* Replace DSK6713 with DSK6416 for the C64x DSK