Introduction

In this module, we will learn how to program the C6000 McBSP using the CSL. First, we’ll learn how the McBSP operates and the choices we can make, and then how to use the CSL to program the selected options. In the lab, you will finally use the DSK to make some “noise”. If it sounds like a song, you got it right. If it really is just noise…then you’ll have some debugging to do…

Learning Objectives

Goals for Module 6…

We will learn how to:
♦ Use the McBSP to communicate with an external codec
♦ Synchronize EDMA transfers with an event
♦ Read the position of DIP switch on the DSK
Chapter Topics

McBSP Overview

- McBSP Overview ................................................................. 6-3
- Block Diagram ........................................................................ 6-3
- Basic McBSP Definitions ............................................................. 6-4
- Clocks and Frame Syncs ................................................................. 6-5
- Serial Port Events ...................................................................... 6-7

EDMA Synchronization Events (Triggering the EDMA) ................. 6-9
- EDMA Event Sources (and their channels) ................................. 6-9
- EDMA Event Register and Enabling .............................................. 6-11

DSK Serial Communications .......................................................... 6-12

McBSP and Codec Initialization ....................................................... 6-14
- McBSP Init ................................................................................ 6-14
- Initializing the AIC23 Codec ......................................................... 6-18

Using the AIC23 Data Channel (EDMA) ............................................ 6-19
- Initialize the McBSPs – Paper Exercise .................................... 6-23
- Initialize the McBSPs – Write the Code .................................... 6-25
- Configure the EDMA to talk to the McBSP .................................. 6-28
- Part A ......................................................................................... 6-38

Optional Topics ........................................................................... 6-39
- DMA vs EDMA: Event Synchronization ...................................... 6-39
- DMA Split Mode ........................................................................ 6-40
- DMA vs EDMA: Updated Summary ............................................ 6-40
McBSP Overview

The Multi-Channel Buffered Serial Port (McBSP) is an extremely flexible serial port. The following graphic is a humorous approach at describing its many standards and capabilities.

Could this be you?
The McBSP is an extremely capable serial port

Block Diagram

The McBSP is a full-duplex, synchronous serial port. Either the CPU or EDMA can read and write to its memory-mapped data registers (DRR, DXR).
Basic McBSP Definitions

The following two slides outline three basic components of the McBSP serial stream: Bit, Word (aka element), and Frame. Both the Word and Frame sizes can be defined in the McBSP’s control registers. In fact, the sizes can even be different between the Receive and Transmit sides of the port. (Note, McBSP frames and EDMA frames are not necessarily equivalent; just coincidental.)
Clocks and Frame Syncs

Being a synchronous serial port, McBSP’s always use a clock. The advantage of synchronous serial ports is speed. The McBSP’s are very fast and can drive rates upwards of 100Mb/sec.

Their receive and transmit bit-clocks (CLKR, CLKX) can each be setup as either an input or output pin.

![CLK & FS Pins: Input or Output](image)

- CLK/FS can be inputs or outputs

<table>
<thead>
<tr>
<th>Serial Port</th>
<th>CL/K FS Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP Ctrl (SPCR)</td>
<td>0: Input</td>
</tr>
<tr>
<td>Recv Ctrl (RCR)</td>
<td>1: Output</td>
</tr>
<tr>
<td>Xmt Ctrl (XCR)</td>
<td></td>
</tr>
<tr>
<td>Rate (SRGR)</td>
<td></td>
</tr>
<tr>
<td>Pin Ctrl (PCR)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>11 10 9 8</th>
<th>FSRM</th>
<th>FSXM</th>
<th>CLKX</th>
<th>CLKR</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSR</td>
<td>FSX</td>
<td>CLKR</td>
<td>CLKX</td>
<td></td>
</tr>
</tbody>
</table>
When used as an output, the McBSP generated (CLKG) clock signal can either be divided down from the C6000’s internal clock or from a separate external clock (CLKS) input.

If You Select CLK as Output …

- **CLKSM**: selects clock src (CLKOUT1 or CLKS)
- **CLKGDV**: divide down (1-255)
- **CLKG** = (input clock) / (1 + CLKGDV)
- Max transfer rate is 100Mb/s (for most ‘C6x devices)

Frame sync signals can also be generated or input into the McBSP. When generated, you can define their period and pulse-width. Optionally, the FSX bit can be generated automatically any time a value is written into the Transmit Serial Register (XSR).

If You Select FS as Output …

- **Frame Sync Gen Mode (FSGM)**:
  - 0 = FSX gen’d on every DXR → XSR copy
  - 1 = FSX and/or FSR gen’d by “Framing”
    - **FPER**: frame sync period (12 bits)
    - **FWID**: frame sync pulse width (8 bits)
Serial Port Events

Interrupts and events are an important part of McBSP usage. It’s great that a serial port can transmit data serially. But, if they cannot signal when data is available (or that they’re ready for more data), they cannot be very effective in embedded systems.

The McBSP’s can generate CPU interrupts for a number of conditions (as shown on the next page). In this workshop, we will only use (and study) one of these conditions: data ready.

When the receive channel has data ready to be written (i.e. data has moved from RBR to DRR), it sets the RRDY bit in the Serial Port Control Register (SPCR). This bit can be used to generate an interrupt to the CPU (RINTx) and/or a trigger event to the EDMA (REVTx).

Similarly, the transmit side of the serial port can set the XRDY bit in the control register and generate the XINTx and XEVTx interrupt and event, respectively.

Interrupts vs. Events

It’s probably worthwhile to define how we use each of these terms:

- **(CPU) Interrupts** are signals sent to the CPU by various sources (most peripherals and the four external interrupt pins).
- **(EDMA) Events** are signals sent to the EDMA to trigger a channel to transfer data. In most cases these are the same signals (and sources) that generate interrupts to the CPU.

It was useful for use to use these two terms in order to differentiate the destination of the various synchronization signals. While the signals may be generated (and thus sent from) a common source, the structures in the CPU and EDMA that deal with them are entirely separate & distinct.
As mentioned on the last page, the McBSP can generate CPU interrupts for various conditions. Shown below are the conditions along with the bit fields used to select which condition will be used to generate an interrupt.

### Triggering the CPU Interrupts (R/XINT)

**“Trigger Event”**

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>RRDY</td>
<td>End of Block (RCV)</td>
</tr>
<tr>
<td></td>
<td>New FSR (frame begin)</td>
</tr>
<tr>
<td></td>
<td>Receive Sync Error</td>
</tr>
<tr>
<td>XRDY</td>
<td>End of Block (XMT)</td>
</tr>
<tr>
<td></td>
<td>New FSX (frame begin)</td>
</tr>
<tr>
<td></td>
<td>Transmit Sync Error</td>
</tr>
</tbody>
</table>

**Serial Port**

- SP Ctrl (SPCR)
- Rcv Ctrl (RCR)
- Xmt Ctrl (XCR)
- Rate (SRGR)
- Pin Ctrl (PCR)

**Serial Port Bit Fields**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>XINTM</td>
</tr>
<tr>
<td>20</td>
<td>RINTM</td>
</tr>
<tr>
<td>17</td>
<td>XRDY</td>
</tr>
<tr>
<td>16</td>
<td>RRDY</td>
</tr>
<tr>
<td>15</td>
<td>RW</td>
</tr>
<tr>
<td>14</td>
<td>R</td>
</tr>
<tr>
<td>13</td>
<td>RW</td>
</tr>
<tr>
<td>12</td>
<td>R</td>
</tr>
</tbody>
</table>

The EDMA, on the other hand, only receives data ready events (REVT, XEVT).

### EDMA Sync Events from McBSP

**Receive Event (REVT)**
- When value reaches DRR, sync event sent to EDMA.
- This can be used to trigger an EDMA transfer.

**Transmit Event (XEVT)**
- Sent to EDMA when DXR is emptied (and ready to receive another value)

The EDMA events and CPU interrupts can work hand-in-hand, though. Normal data ready events can be serviced by the EDMA, while CPU can be interrupted to handle error conditions that might occur.
EDMA Synchronization Events (Triggering the EDMA)

EDMA Event Sources (and their channels)

One of the reasons the EDMA has so many channels is that each one is dedicated to a different interrupt event. You don’t have to remember which event is associated with which channel, though, as the EDMA_open() function manages this for you.

<table>
<thead>
<tr>
<th>EDMA Channel</th>
<th>Event Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 DSPINT</td>
<td>HPI to DSP interrupt</td>
</tr>
<tr>
<td>1 TINT0</td>
<td>Timer 0 interrupt</td>
</tr>
<tr>
<td>2 TINT1</td>
<td>Timer 1 interrupt</td>
</tr>
<tr>
<td>3 SD_INT</td>
<td>EMIF SDRAM timer interrupt</td>
</tr>
<tr>
<td>4 EXT_INT4</td>
<td>External interrupt pin 4</td>
</tr>
<tr>
<td>5 EXT_INT5</td>
<td>External interrupt pin 5</td>
</tr>
<tr>
<td>6 EXT_INT6</td>
<td>External interrupt pin 6</td>
</tr>
<tr>
<td>7 EXT_INT7</td>
<td>External interrupt pin 7</td>
</tr>
<tr>
<td>8 EDMA_TCC8</td>
<td></td>
</tr>
<tr>
<td>9 EDMA_TCC9</td>
<td></td>
</tr>
<tr>
<td>10 EDMA_TCC10</td>
<td></td>
</tr>
<tr>
<td>11 EDMA_TCC11</td>
<td></td>
</tr>
<tr>
<td>12 XEVT0</td>
<td>McBSP0 transmit event</td>
</tr>
<tr>
<td>13 REV0</td>
<td>McBSP0 receive event</td>
</tr>
<tr>
<td>14 XEVT1</td>
<td>McBSP1 transmit event</td>
</tr>
<tr>
<td>15 REV1</td>
<td>McBSP1 receive event</td>
</tr>
</tbody>
</table>

- Each channel is associated with a specific sync event
- When a sync event is unused, that channel may still be programmed for a simple block memory-copy operation

The above channels with shown with their sync events was originally designed for the C6711.

The C6713, though, has many more peripherals and thus additional synchronization events. To allow the 16 channel EDMA to accommodate a much larger number of event sources, you can now configure the EDMA channels with whichever event source you prefer. This is done through the memory-mapped EDMA event selector registers. Please refer to the C6713 data sheet additional information.

The list above is the default values for the C6713 EDMA channels. Since the events we care about in our lab exercises are on the above list, we won’t have to reconfigure the EDMA’s event sources.

The C6416 also has a vast number of EDMA event sources. With 64 channels, though, there are still more channels than there are sources. The next page shows the C6416 events and their associated channels.
Included below is a page from the C6416 datasheet which lists the EDMA channel sync events.

Table 25. TMS320C64x EDMA Channel Synchronization Events

<table>
<thead>
<tr>
<th>EDMA CHANNEL</th>
<th>EVENT NAME</th>
<th>EVENT DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DSP_INT</td>
<td>HP1/PCI to DSP interrupt (PCI peripheral supported on C6416 and C6415 only)†</td>
</tr>
<tr>
<td>1</td>
<td>TINT0</td>
<td>Timer 0 interrupt</td>
</tr>
<tr>
<td>2</td>
<td>TINT1</td>
<td>Timer 1 interrupt</td>
</tr>
<tr>
<td>3</td>
<td>SD_INTA</td>
<td>EMIF SRAM timer interrupt</td>
</tr>
<tr>
<td>4</td>
<td>GPINT4/EXT_INT4</td>
<td>GPIO event 4/External interrupt pin 4</td>
</tr>
<tr>
<td>5</td>
<td>GPINT5/EXT_INT5</td>
<td>GPIO event 5/External interrupt pin 5</td>
</tr>
<tr>
<td>6</td>
<td>GPINT6/EXT_INT6</td>
<td>GPIO event 6/External interrupt pin 6</td>
</tr>
<tr>
<td>7</td>
<td>GPINT7/EXT_INT7</td>
<td>GPIO event 7/External interrupt pin 7</td>
</tr>
<tr>
<td>8</td>
<td>GPINT0</td>
<td>GPIO event 0</td>
</tr>
<tr>
<td>9</td>
<td>GPINT1</td>
<td>GPIO event 1</td>
</tr>
<tr>
<td>10</td>
<td>GPINT2</td>
<td>GPIO event 2</td>
</tr>
<tr>
<td>11</td>
<td>GPINT3</td>
<td>GPIO event 3</td>
</tr>
<tr>
<td>12</td>
<td>XEV10</td>
<td>MdSIP transmit event</td>
</tr>
<tr>
<td>13</td>
<td>XEV11</td>
<td>MdSIP receive event</td>
</tr>
<tr>
<td>14</td>
<td>XEV12</td>
<td>MdSIP1 transmit event</td>
</tr>
<tr>
<td>15</td>
<td>REV11</td>
<td>MdSIP1 receive event</td>
</tr>
<tr>
<td>16</td>
<td>REV12</td>
<td>MdSIP2 receive event</td>
</tr>
<tr>
<td>17</td>
<td>REV13</td>
<td>MdSIP2 transmit event</td>
</tr>
<tr>
<td>18</td>
<td>REV14</td>
<td>MdSIP2 receive event</td>
</tr>
<tr>
<td>19</td>
<td>TINT2</td>
<td>Timer 2 interrupt</td>
</tr>
<tr>
<td>20</td>
<td>SD_INTB</td>
<td>EMIF SRAM timer interrupt</td>
</tr>
<tr>
<td>21</td>
<td>Reserved</td>
<td>Reserved, for future expansion</td>
</tr>
<tr>
<td>22–27</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>28</td>
<td>VCPREVVT</td>
<td>VCP receive event (C6416 only)§</td>
</tr>
<tr>
<td>29</td>
<td>VCPXEVVT</td>
<td>VCP transmit event (C6416 only)§</td>
</tr>
<tr>
<td>30</td>
<td>TCPREVVT</td>
<td>TCP receive event (C6416 only)§</td>
</tr>
<tr>
<td>31</td>
<td>TCPXEVVT</td>
<td>TCP transmit event (C6416 only)§</td>
</tr>
<tr>
<td>32</td>
<td>UREVT</td>
<td>UTOPA receive event (C6414 and C6415 only)‡</td>
</tr>
<tr>
<td>33–36</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>37</td>
<td>UXTVT</td>
<td>UTOPA transmit event (C6415 and C6416 only)‡</td>
</tr>
<tr>
<td>38–47</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>48</td>
<td>GPINT8</td>
<td>GPIO event 8</td>
</tr>
<tr>
<td>49</td>
<td>GPINT9</td>
<td>GPIO event 9</td>
</tr>
<tr>
<td>50</td>
<td>GPINT10</td>
<td>GPIO event 10</td>
</tr>
<tr>
<td>51</td>
<td>GPINT11</td>
<td>GPIO event 11</td>
</tr>
<tr>
<td>52</td>
<td>GPINT12</td>
<td>GPIO event 12</td>
</tr>
<tr>
<td>53</td>
<td>GPINT13</td>
<td>GPIO event 13</td>
</tr>
<tr>
<td>54</td>
<td>GPINT14</td>
<td>GPIO event 14</td>
</tr>
<tr>
<td>55</td>
<td>GPINT15</td>
<td>GPIO event 15</td>
</tr>
<tr>
<td>56–83</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>

† In addition to the events shown in this table, each of the 64 channels can also be synchronized with the transfer completion or alternate transfer completion events. For more detailed information on EDMA event-transfer chaining, see the EDMA Controller chapter of the TMS320C6000 Peripherals Reference Guide (literature number SPRU190).

‡ The PCI and UTOPA peripherals are not supported on the C6414 device; therefore, these EDMA synchronization events are reserved.

§ The VCP/TCP EDMA synchronization events are supported on the C6416 only. For the C6414 and C6415 devices, these events are reserved.
EDMA Event Register and Enabling

The EDMA’s event input mechanism is similar to the CPU’s in that it has both flag and enable registers. In the case of the EDMA they are called:

- Event Register (ER): set to a one when an event is received from its respective source
- Event Enable Register (EER): if enabled (set to 1), it allows a received event to trigger the associated EDMA channel to run

Since an event source is going to send a signal whether you want the EDMA to respond or not, the EER allows you to prevent the associated channel from running.

---

### EDMA Sync Events (ER, EER)

- **Previously, EDMA_setChannel() triggered an EDMA channel to run**
- **XEVT1 & REVT1 set the appropriate bits in the Event Register (ER), rather than our code doing this manually**
- **What if there is a sync event I don’t want the EDMA to respond to? Say, DSPINT?**
  - The Event Enable Register (EER) allows event inputs to be blocked.

---

**Hint:** When you set an ER bit from the CPU (for example, when using the EDMA_setChannel() function as we have been doing in our past two lab exercises), the associated EER bit value is ignored. That is, manually setting a channel to run will occur regardless of the value in EER.
The DSK designers chose to use the inexpensive and flexible 'AIC23 codec. It features:

- 24-bit resolution (90db SNR ADC, 100db SNR DAC)
- Multiple Digital transfer widths (16-bits, 20-bits, 24-bits, 32-bits)
- Programmable frequency (8K, 16K, 24K, 32K, 44.1K, 48K, 96K)
- AIC23 has two serial data pins:
  - Input for control – reads/writes AIC23’s control registers
  - Bidirectional pin to transfer data to A/D and D/A converters

The DSK utilizes two McBSP’s to handle AIC23 setup and data transfers, respectively. While one McBSP could be used to handle a single AIC23, it was easier (and saved a small amount of ‘glue’ logic) to use two McBSP’s. Besides, the DSK has only one codec and the DSP’s have 2 or 3 McBSP’s.

- McBSP1 connected to program AIC23’s control registers
- McBSP2 is used to transfer data to A/D and D/A converters
- One McBSP could be made to handle the AIC23, but since multiple McBSP’s were available, using two made the design easier
The C6416 DSK was designed first. It utilized McBSP1 and McBSP2 for the codec interface. When the C6713 DSK was designed, though, McBSP0 & McBSP1 had to be used since the C6713 doesn’t have a McBSP2.

**C6713 DSK: McBSP ↔ Codec Interface**

- McBSP0 connected to program AIC23’s control registers
- McBSP1 is used to transfer data to A/D and D/A converters
- One McBSP could be made to handle the AIC23, but since multiple McBSP’s were available, using two made the design easier
McBSP and Codec Initialization

The difficult part of using a McBSP or codec is in setting them up. Once that’s done, you only need to read or write data to make them work.

To initialize our data stream, we first initialize the McBSP, then use it to setup the codec.

General Procedure to Initialize Codec

1. Setup McBSP
2. Setup Codec via McBSP

舍得 Since the AIC23 is connected to the McBSP, you must first initialize the McBSP, then the codec.

C6416 DSK
- McBSP1 used for control channel

C6713 DSK
- McBSP0 used for control channel

McBSP Init

The McBSP’s can be initialized use CSL functions, definitions, and macros. The process is similar to that of setting up the EDMA. Though, you’ll find the McBSP has more choices and registers. (Good in that all these options belie its flexibility; less so in that you have to figure them all out.)

One thing that differs between EDMA configuration and McBSP configuration is that the McBSP configuration choices are directly related to what the port is connected to. On the DSK, this is the AIC23 codec.

With the great flexibility of the McBSP, you can connect to a great many types of serial devices. In each case, though, you will need to read and understand the data sheet of the device you are connecting to and configure the McBSP accordingly. (This isn’t unlike the old days of using computer modems. To connect to your bank, for example, you usually needed to know the proper settings: bit size, parity, etc.)

The process of reading and deciphering a codec datasheet can be time consuming (and sometimes difficult). Based on this, and the fact that all serial devices seem to work differently, we have chosen not to spend the hours required for this process. Rather, we have provided the McBSP settings provided by the DSK board manufacturer.
The McBSP settings provided by the DSK designers are used in the provided MCBSP_Config structures. Still, you will get to write the remaining McBSP initialization code. Shown below are the same six CSL steps we have been using to configure other peripherals.

### 1. McBSP Setup

```c
#include <csl.h>
#include <csl_mcbsp.h>

MCBSP_Handle hMcbsp0;

MCBSP_Config mcbspCfgControl = {
  0x00001000, // Serial Port Control Reg. (SPCR)
  0x00000000, // Receiver Control Reg. (RCR)
  0x00000040, // Transmitter Control Reg. (XCR)
  0x20001363, // Sample-Rate Generator Reg. (SRGR)
  0x00000000, // Multichannel Control Reg. (MCR)
  0x00000000, // Receiver Channel Enable (RCER)
  0x00000000, // Transmitter Channel Enable (XCER)
  0x00000A0A  // Pin Control Reg. (PCR)
};

void initMcBSP()
{
  hMcbsp0 = MCBSP_open(MCBSP_DEV0, MCBSP_OPEN_RESET);
  MCBSP_config(hMcbsp0, &mcbspCfgControl);
  MCBSP_start(hMcbsp0, MCBSP_XMIT_START | MCBSP_SRGR_START | MCBSP_SRGR_FRAMESYNC, 100);
}
```

Let's look more closely the McBSP configuration ...

### 1. McBSP Config (a)

```c
MCBSP_Config mcbspCfgControl = {
  MCBSP_SPCR_RMK(
    MCBSP_SPCR_FREE_NO,  // Previous slide shows config as 32-bit hex values (because it fit on 1 slide).
    MCBSP_SPCR_SOFT_NO,  // A better method uses _RMK macros. Improves:
    MCBSP_SPCR_FRST_YES, //  - Readability
    MCBSP_SPCR_GRST_YES, //  - Maintainability
    MCBSP_SPCR_XINTM_XRDY,
    MCBSP_SPCR_XSYNCERR_NO,
    MCBSP_SPCR_XRST_YES,
    MCBSP_SPCR_DLB_OFF,
    MCBSP_SPCR_RJUST_RZF,
    MCBSP_SPCR_CLKSTP_NODELAY,
    MCBSP_SPCR_DXENA_OFF,
    MCBSP_SPCR_RINTM_RDY,
    MCBSP_SPCR_RSYNCERR_NO,
    MCBSP_SPCR_RRST_YES
  ),
};
```

### Technical Training

**Organization**

**TT O**

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*

---

*McBSP and Codec Initialization*
While your instructor won’t show the remaining three slides of the McBSP configuration, they are provided for completeness.

1. McBSP Config (c)

```c
MCBSP_SRGR_RMK(
    MCBSP_SRGR_GSYNC_FREE,
    MCBSP_SRGR_CLKSP_RISING,
    MCBSP_SRGR_CLKSM_INTERNAL,
    MCBSP_SRGR_FSGM_DXR2XSR,
    MCBSP_SRGR_FPER_OF(0),
    MCBSP_SRGR_FWID_OF(19),
    MCBSP_SRGR_CLKGDV_OF(99)
),
```
1. McBSP Config (d)

- McBSP_MCR_DEFAULT,
- McBSP_RCERE0_DEFAULT,
- McBSP_RCERE1_DEFAULT,
- McBSP_RCERE2_DEFAULT,
- McBSP_RCERE3_DEFAULT,
- McBSP_XCERE0_DEFAULT,
- McBSP_XCERE1_DEFAULT,
- McBSP_XCERE2_DEFAULT,
- McBSP_XCERE3_DEFAULT,

- These registers control the multi-channel capabilities of the McBSP.
- We aren’t using these features in our lab exercises.

1. McBSP Config (e)

```c
MCBSP_PCR_RMK(
    MCBSP_PCR_XIOEN_SP,
    MCBSP_PCR_RIOEN_SP,
    MCBSP_PCR_FSXM_INTERNAL,
    MCBSP_PCR_FSRM_EXTERNAL,
    MCBSP_PCR_CLKXM_OUTPUT,
    MCBSP_PCR_CLKRM_INPUT,
    MCBSP_PCR_CLKSSTAT_DEFAULT,
    MCBSP_PCR_DXSTAT_DEFAULT,
    MCBSP_PCR_FSRM_ACTIVELOW,
    MCBSP_PCR_FSRM_DEFAULT,
    MCBSP_PCR_CLKXP_FALLING,
    MCBSP_PCR_CLKRP_DEFAULT
)
```


Initializing the AIC23 Codec

The second part of our serial codec initialization is to setup the AIC23 codec, itself.

```c
void initCodec(MCBSP_Handle hMcbsp) {
    short codecConfig[10] = {
        0x0017, // 0 Left line input channel volume
        0x0017, // 1 Right line input channel volume
        0x01f9, // 2 Left channel headphone volume
        ... 
    };
    for (i = 0; i < 10; i++) {
        MCBSP_write(hMcbsp,(i << 9)|codecConfig[i]);
    }
}
```

The codec contains a number of control registers that need to be programmed. These registers specify options for: input and output gain, codec loopback mode, sample frequency, bit-resolution, etc.

Again, since a codec init routine is specific to a given codec, we have provided this routine for you. From the diagram above, you can see the codec routine includes an initialization structure, and a routine that sends the values via the McBSP to the codec control registers. You will find this code in the `codec.c` file.
Using the AIC23 Data Channel (EDMA)

As noted earlier, once configured, using a serial codec is easy. You simply need to read and write to the appropriate McBSP data registers.

Of course, the upcoming lab uses the EDMA to perform the codec reads and writes. This is common for most systems, and a good suggestion, since the EDMA can easily off-load this task from the CPU.

**Note:** Not only do you save the CPU MIPs required to do the reads/writes, but you also minimize the cycles required by the CPU interrupt overhead.

When using the EDMA for McBSP reads/writes, there are a few changes that need to be made to our previous EDMA initialization code. Here’s an example of using the EDMA channel for McBSP transmit:

```
Using the Codec (via EDMA)

Using the Codec (via EDMA)

```

![Diagram of McBSP and EDMA interaction](image)

```c
(... EDMA_OPT_SUM_INC, // Src update mode?
EDMA_OPT_DUM_NONE, // Dest update mode?
EDMA_OPT_TCINT_YES, // Cause EDMA interrupt?
EDMA_OPT_TCC_OF(0), // Transfer complete code?
EDMA_OPT_FS_NO), // Use frame sync?
...
EDMA_SRC_OF(gBufXmt), // src address?
EDMA_DST_OF(0), ... // dest address?

hEdmaXmt = EDMA_open(EDMA_CHA_XEVT2, EDMA_OPEN_RESET);
gEdmaConfigXmt.dst = MCBSP_getXmtAddr(hMcbsp2);
EDMA_intEnable(gTcc);
```

**Note:** McBSP1 and XEVT1 for C6713
*** this page was unintentionally left blank ***
Lab 6 – Audio Pass Thru

Goals:
1. EDMA (RCV) copies values from DRR to gBufRcv
2. CPU copies gBufRcv to gBufXmt
3. EDMA (XMT) copies gBufXmt to DXR
4. Opt: add sine to gBufRcv based on DIP switch

In order to successfully complete this lab, we will need to make the following changes to our code:

1. Change the buffer names so that they make more sense for an audio pass-through. We used names that imply whether the buffer is being used for receive or transmit of the audio.
2. Write the code to initialize two McBSP's (one for control, and one for data).
3. Call a provided routine to initialize the AIC23 codec.
4. Change the transmit EDMA's setup to talk to the McBSP.
5. Add a receive EDMA channel to talk to the McBSP.
6. Modify the EDMA HWI that we have been using to respond to both transmit and receive interrupts, and copy the data.

We have provided some paper exercises to help you along the way. Please use the exercises to test your understanding of what you are doing in this lab. If you have any questions, please feel free to ask your instructor.
**Open the Project**

1. Reset the DSK and start CCS
2. Open audioapp.pjt.

**Make the code more readable**

We need to change some variable names to line up better with the audio application we are building. So, take your time and be careful as you change these names. One small slip and you’ll be debugging it for an hour. No pressure…eh? Use the Edit/Find-Replace feature in CCS.

3. Change all occurrences of:
   - gBuf0 to gBufXmt in both main.c and edma.c
   - gBuf1 to gBufRcv in both main.c and edma.c
   - hEdmaReload to hEdmaReloadXmt in edma.c
   - hEdma to hEdmaXmt (be careful to only choose hEdma, not hEdmaReloadXmt, etc. in main.c, edma.c and edma.h)
   - gEdmaConfig to gEdmaConfigXmt in edma.c
4. Build your project and fix any errors that occur.
Initialize the McBSPs – Paper Exercise

The next task that we need to do is to initialize two McBSP’s to be used to communicate with the AIC23 codec on the DSKs.

As we discussed, the DSK uses two McBSP's to interface with the codec:

- **One serial port to setup and configure the codec.**
  A global variable, called `mcbspCfgControl`, was created and initialized with the appropriate bitfield values to send control register values to the AIC23.

- **A second serial port to send and receive data to/from the codec.**
  The global variable `mcbspCfgData` contains the configuration values to setup the serial port which reads/writes data to the AIC23.

**Why did we write the McBSP configuration structures for you?**

The configuration choices for a McBSP configuration are entirely dependent upon what it is connected to. As an analogy:

When you want to use your modem to connect to your bank, first you must get the configuration choices from them (e.g. 9600 baud, 8-bits, no-parity, etc.). Once you have this information, you can configure the modem.

In the same fashion, once you know the configuration options required by the serial device you are connecting the McBSP to, you can easily plug them in. Unfortunately, extracting the required information from an analogue data converter datasheet is often not trivial. Ideally, we would have enjoyed taking you through this process for the AIC23, but given the time constraints in the workshop plus the fact that you most likely are using another converter (or if using the AIC23, you can just use our code) we decided to provide these Config’s for you.

**What else is there left to write?**

Here is a summary of the things that you need to add to mcbsp.c that we have provided for you in order to use the codec (and the McBSP’s):

- Add the code to open and configure both McBSP’s.
- Start the control McBSP
- Call the provided initCodec( ) routine and pass it the handle for the control McBSP
- Start the data McBSP

To make this all a little easier, we have provided a space for you to write your answers on paper, before you try to write the code. You will need to refer back to the lecture material to figure out exactly what to write. We have provided some hints to help you. These hints are the actual lab steps that you will do to write the code inside CCS. Please write this code in the space provided on the next page …
```c
#include <__________.h>
#include <__________.h>
#include "__________.h"

void initMcBSP(void);

MCBSP_Config mcbspCfgControl = {  
  Provided for you. See file for details.
};
MCBSP_Config mcbspCfgData = {  
  Provided for you. See file for details.
};
MCBSP_            hMcbspControl;
MCBSP_            hMcbspData;

void initMcBSP(void) {  
  /* Open McBSP port for codec control */
  ________________________________________

  /* Open McBSP port for data read/write */
  ________________________________________

  /* Configure McBSP for codec control */
  ________________________________________

  /* Configure McBSP for codec data */
  ________________________________________

  /* Start McBSP for the codec control channel */
  ________________________________________

  /* Call the codec initialization routine */
  ________________________________________

  /* Clear any garbage from the codec data port */
  ________________________________________

  /* Start McBSP used for the codec data channel */
  ________________________________________
  
  MCBSP_write(hMcbspData,0);
}
```

Why do we need this MCBSP_write? ________________________________
Initialize the McBSPs – Write the Code

Now that you have a handle (get it?) on what you need to do in order to initialize the McBSP’s, return back to CCS to write the code. Use your answers from the paper exercise to complete the steps below.

5. **Add mcbsp.c to your project**
   
The mcbsp.c file is like the edma.c file, it has some simple starter code to help you out but you will write most of it.

6. **Open mcbsp.c and inspect it**
   
   You should see the two configurations that we provided for you near the top of the file. The rest of the file should look very similar to the paper exercise and it should be easy to figure out where to put your code. We have provided the steps below to help you through the process.

7. **Delete comments on the code for your processor**
   
   Inside the two configuration structures for the control and data McBSP’s, there are a few lines of code that are specific to the C67x and the C64x. The serial ports on the two devices are just a little different, so we need to account for this. Find the comments in each of the two structures and remove the comments for the processor that you are using. Removing the comments will put this code into the structure.

   Please be very careful making these changes. Those using the C67x will need to remove the comments from 2 lines of code per structure. Those using the C64x will need to remove the comments from 8 lines of code per structure.

8. **Add the header files necessary to use the CSL’s MCBSP module**
   
   Each CSL module requires two header files. Add the two that are needed to use the McBSP module in mcbsp.c. `<csl.h>` and `<csl_mcbsp.h>`

9. **Create a McBSP_Handle for the control and data McBSP’s**
   
   Just under the provided configuration structures, create two McBSP handles for the control and data McBSP’s. Name them hMcbspControl and hMcbspData respectively. Make sure that they are global.

10. **Modify the initMcBSP( ) function to open the control serial port**

    Add the function call necessary to open the control serial port for your DSK. Use the table below to figure out which one to use, or use the online help for your DSK.

    Make sure to open the correct serial port and reset it when you open it. Make sure to set the return value to the correct handle.

    | Control   | Data   |
    |-----------|--------|
    | 6713 DSK  | McBSP0 | McBSP1 |
    | 6416 DSK  | McBSP1 | McBSP2 |

    Note: Symbol name is MCBSP_DEVx where x is the McBSP number.
11. Open the data serial port
   Use code that is similar to that used in the previous step to open the data serial port for your DSK.

12. Configure the control serial port
   Use the appropriate CSL API to configure the control serial port. Pass in the correct configuration structure. Don't forget to use the correct handle.

13. Configure the data serial port
   Both serial ports need to be configured correctly for everything to work.

14. Start the control serial port
   Due to the way we set up the configuration structure for both serial ports, the ports themselves will not actually start until we tell them to. There are individual APIs that can start each independent piece, or we can start each piece all at once with a call like this:

   ```c
   MCBSP_start(hMcbspControl, MCBSP_XMIT_START | MCBSP_SRGR_STARTED | MCBSP_SRGR_FRAMESYNC, 100);
   ```

   **Note:** This is all one line of code. Since it is so long we broke it up for you. The value 100 is the *sample rate generator delay*. McBSP logic requires 2 SRGR clock cycles after enabling the sample rate generator for its logic to stabilize. This parameter is used to provide the appropriate delay.

15. Use the control McBSP to initialize the codec
   Now that the control McBSP is up and running, we can use it to program the AIC23. We have written this code for you and put it in codec.c. All you need to do is call initCodec( ) and pass it a handle to the control McBSP. Add this function call to initMcBSP( ) here.

16. Clean up the data receive register on the data McBSP
   Just to make sure that the data receive register doesn't have any garbage (bad data) sitting in the receive register, add this code:

   ```c
   if (MCBSP_rrdy(hMcbspData))
       MCBSP_read(hMcbspData);
   ```

   This code checks to see if there is anything in the register. If there is, it reads it and throws it away.

17. Start the data serial port
   We are using different pieces of the data serial port, so the code to start it is a little different:

   ```c
   MCBSP_start(hMcbspData, MCBSP_XMIT_START | MCBSP_RCV_START | MCBSP_SRGR_STARTED | MCBSP_SRGR_FRAMESYNC, 220);
   ```

18. Add codec.h to mcbsp.c
   Before we leave the mcbsp.c file, we need to include the header file for the codec, codec.h. It simply has the prototype of the initCodec( ) function.
19. **Call initMcBSP() from main()**
   Add a call to initMcBSP() in main() just before the call to initEdma().

20. **Include mcbsp.h in main.c**
    mcbsp.h has the prototype for the initMcBSP() function as well as the externs for the handles that we will need later.

**Inspect the codec initialization code**

21. **Take a look at codec.c**
    Open codec.c and look at the code inside. This code is simple a data structure of initial values for the codec and the code to write these values through the McBSP who's handle is passed into initCodec(). If you wanted to change how the AIC23 is setup, you could simply change the values in the configuration structure.

22. **Add codec.c to your project**
    Don't forget to add this file to your project since it contains the initCodec() function.
Configure the EDMA to talk to the McBSP

Now that we have the McBSP and the Codec initialized, we need to configure the EDMA to talk to the McBSP (that talks to the Codec). In order to do this, we will need to make the following changes:

1. Change the transmit EDMA configuration to send data to the McBSP.
2. Create a receive EDMA configuration to receive data from the McBSP.
3. Modify the `initEdma()` routine to configure both EDMA channels.
4. Modify the `edmaHwi()` to respond to both channels and copy the data.

Modify the EDMA Config Structures – Paper Exercise

Let's take a moment to see how the configuration structures for the EDMA will need to change in order to talk with the McBSP. Since the McBSP is full-duplex (both receive and transmit), we will need two half-duplex (uni-directional) EDMA channels to exchange data with it. We will use the configuration structure that we already have for the transmit channel to start with.

To make sure that we understand the changes that we are making, let's do another paper exercise before we write the code. Take a look at the following sheet and try to figure out what changes will need to be made in order to configure the EDMA to exchange data with the McBSP, for both receive and transmit.
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_INC, // 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_TCCM_DEFAULT, // TCC Upper Bits (c64x only)
    EDMA_OPT_ATCINT_DEFAULT, // Alternate TCC Interrupt (c64x only)
    EDMA_OPT_ATCC_DEFAULT, // Alternate TCC (c64x only)
    EDMA_OPT_PTS_DEFAULT, // PDT Source (c64x only)
    EDMA_OPT_PTD_DEFAULT, // PDT Dest (c64x only)
    EDMA_OPT_LINK_NO, // Enable link parameters
    EDMA_OPT_FS_YES // Use frame sync
  ),
  EDMA_SRC_OF(gBuf0), // src address
  EDMA_CNT_OF(BUFFSIZE), // Count = buffer size
  EDMA_DST_OF(gBuf1), // dest address
  EDMA_IDX_OF(0), // frame/element index value
  EDMA_RLD_OF(0) // reload
};

EDMA_Config gEdmaConfigXmt = {
  EDMA_OPT_RMK(
    EDMA_OPT_PRI_LOW, // Priority
    EDMA_OPT_ESIZE_16BIT, // Element size
    EDMA_OPT_2DS_NO, // 2 dimensional source
    EDMA_OPT_SUM_INC, // 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_TCCM_DEFAULT, // TCC Upper Bits (c64x only)
    EDMA_OPT_ATCINT_DEFAULT, // Alternate TCC Interrupt (c64x only)
    EDMA_OPT_ATCC_DEFAULT, // Alternate TCC (c64x only)
    EDMA_OPT_PTS_DEFAULT, // PDT Source (c64x only)
    EDMA_OPT_PTD_DEFAULT, // PDT Dest (c64x only)
    EDMA_OPT_LINK_NO, // Enable link parameters
    EDMA_OPT_FS_YES // Use frame sync
  ),
  EDMA_SRC_OF(gBuf0), // src address
  EDMA_CNT_OF(BUFFSIZE), // Count = buffer size
  EDMA_DST_OF(gBuf1), // dest address
  EDMA_IDX_OF(0), // frame/element index value
  EDMA_RLD_OF(0) // reload
};

Hint: Step 24

Hint: Step 23
Modify the EDMA Config Structures – Write the Code

Now that you've figured out all of the changes that need to be made to the code, use the steps below to change edma.c inside CCS.

23. Edit the EDMA Config Structure – gEdmaConfigXmt in edma.c

We will use the current config structure from the previous lab to set up the EDMA channel for the transmit side of the data McBSP. The purpose of this channel will be to transfer values FROM the transmit buffer (gBufXmt) to the transmit register of the data McBSP (fixed address). Check the current settings in the Xmt config structure with this goal in mind and make the necessary modifications. What needs to change? If you need a hint…read on:

- Destination Update Mode (DUM) to NONE
- Frame Sync (FS) to NO (we are now using element synchronization)
- The destination address must be calculated after the data McBSP resource is open and we have a handle. So, initialize the destination address to zero for now.

Note: Refer to the lab diagram and draw notes on that diagram to help you gain a mental image of what is going on in the lab. This will help drive a better understanding of the necessary steps to get the lab working.

24. Create a Receive config structure

Copy the entire gEdmaConfigXmt structure and paste a copy of it right above the existing structure. Rename the new structure, the one that comes first in the code, to gEdmaConfigRcv. The goal of this EDMA channel is to read values from the serial port and place them into a buffer. Modify the receive structure with this goal in mind. What needs to change? If you need a hint, read the following:

- SUM to NONE, DUM to INC
- Source addr = 0, Dest addr = gBufRcv

25. Build your code and fix any errors.
Modify initEdma() – Paper Exercise

We now need to modify the initEdma() function in edma.c to:

- Specify explicitly the sync events used for transmit and receive (vs. ANY channel)
- Initialize the source address of the receive side (data McBSP’s rcv register)
- Allocate a TCC bit for the receive side and put it in the receive structure
- Initialize the destination address of the transmit side (data McBSP’s xmt register)
- Add code to enable both of the channels

Instructions

Here is another exercise to help you understand the changes that you need to make to your code. The opposite page is basically a picture of what your initEdma() function will look like if you take the code that we have already written for Lab 5 and modify it to create a Receive channel and communicate with the McBSP. We’ve already copied the code to create the Receive EDMA channel for you, like we did with the structures earlier. But, we haven’t made all of the changes the you will need to make. We did change the comments for you if you need some help.

So, take a few minutes and try to make all of the necessary changes to the code. We've already made a few of them for you so that you have an idea of what we are looking for. If you need some help, use the hints provided to refer to the actual lab steps that will help you write the code in CCS.
void initEdma (void) {
    // get hEdma handle and reset channel
    hEdmaXmt = EDMA_open(EDMA_CHA_XMT, EDMA_OPEN_RESET);
    hEdmaReloadXmt = EDMA_allocTable(-1);
    EDMA_config(hEdmaReloadXmt, &gEdmaConfigXmt);
    hEdmaXmt = EDMA_open(EDMA_CHA_XMT, EDMA_OPEN_RESET);
    hEdmaReloadXmt = EDMA_allocTable(-1);
    EDMA_config(hEdmaReloadXmt, &gEdmaConfigXmt);
    hEdmaXmt = EDMA_open(EDMA_CHA_XMT, EDMA_OPEN_RESET);
    hEdmaReloadXmt = EDMA_allocTable(-1);
    EDMA_config(hEdmaReloadXmt, &gEdmaConfigXmt);
    hEdmaXmt = EDMA_open(EDMA_CHA_XMT, EDMA_OPEN_RESET);
    hEdmaReloadXmt = EDMA_allocTable(-1);
    EDMA_config(hEdmaReloadXmt, &gEdmaConfigXmt);
    hEdmaXmt = EDMA_open(EDMA_CHA_XMT, EDMA_OPEN_RESET);
    hEdmaReloadXmt = EDMA_allocTable(-1);
    EDMA_config(hEdmaReloadXmt, &gEdmaConfigXmt);
    // get an open TCC and put it in the transmit configuration struct
    gXmtTCC = EDMA_intAlloc(-1);
    gEdmaConfigXmt.opt |= EDMA_FMK(OPT, TCC, gXmtTCC);
    gXmtTCC = EDMA_intAlloc(-1);
    gEdmaConfigXmt.opt |= EDMA_FMK(OPT, TCC, gXmtTCC);
    // set the receive's source to the Data Serial Port
    // configure the receive channel with the correct structure
    EDMA_config(hEdmaXmt, &gEdmaConfigXmt);
    // get hEdmaReloadRcv handle and configure it
    hEdmaReloadRcv = EDMA_allocTable(-1);
    EDMA_config(hEdmaReloadRcv, &gEdmaConfigRcv);
    // set up the reload addresses for both hEdmaRcv and hEdmaReloadRcv
    EDMA_link(hEdmaXmt, hEdmaReloadRcv);
    EDMA_link(hEdmaReloadRcv, hEdmaReloadRcv);
    // get hEdmaXmt handle and reset channel
    hEdmaXmt = EDMA_open(EDMA_CHA_XMT, EDMA_OPEN_RESET);
    hEdmaReloadRcv = EDMA_allocTable(-1);
    EDMA_config(hEdmaReloadRcv, &gEdmaConfigRcv);
    // set up the reload addresses for both hEdmaRcv and hEdmaReloadRcv
    EDMA_link(hEdmaXmt, hEdmaReloadRcv);
    EDMA_link(hEdmaReloadRcv, hEdmaReloadRcv);
    // clear any possible spurious interrupts
    EDMA_intClear(gXmtTCC);
    EDMA_intClear(gXmtTCC);
    // enable EDMA interrupts (CIER)
    EDMA_intEnable(gXmtTCC);
    EDMA_intEnable(gXmtTCC);
    // enable channels ...
}
Modify initEdma() – Write the Code

Now that you have an idea of what you need to do, you can either try to write the code yourself or go through the following steps.

26. Specify transmit side’s sync event

Find the function initEdma() in your code. In the previous lab, we used EDMA_CHA_ANY to pick “any” channel for the transfer from source to destination. Instead of “any” channel, we need to specify the sync event we want for the transmit side. So, in the transmit side’s EDMA_open() API, change EDMA_CHA_ANY to EDMA_CHA_XEVTx where x is equal to the number for your data serial port (1 for C6713, 2 for 6416). You can actually pick one of many sync events supported by the EDMA. If you desire, open the CSL Reference Guide and search for “XEVT”. This will take you to the list of options for EDMA_open.

27. Initialize the transmit side destination address in initEdma()

Let’s work on the transmit side first. The only item that we have left to do is to determine the destination address, i.e. the transmit register of the data McBSP. In the function initEdma(), just after the stmt:

```c
    gEdmaConfigXmt.opt |= …
```

Add the following line of code to initialize the destination address:

```c
    gEdmaConfigXmt.dst = MCBSP_getXmtAddr(hMcbspData);
```

28. Include mcbsp.h in edma.c

Since hMcbsData is declared in mcbsp.c, we need to reference it in this file (edma.c). The mcbsp.h file has the reference that we need, so why not just include it.

29. Create Receive Side EDMA channel initialization

In initEdma(), copy the lines of code that configure the transmit side and paste them just above the call to the EDMA_open( ) for the transmit side. To double-check…it’s 9 lines of code (_open, _intAlloc, edmaConfigXmt.opt = , Xmt.dst =, _config, ReloadXmt =, _config, _link * 2).

Use this code as a starting place to configure the receive side by replacing Xmt with Rcv (don’t use search/replace – just do it manually).

Change the _open to use the appropriate REVTX instead of XEVTx and use the following function (between _open and _config) to set the source address of the transfer rather than the destination:

```c
    gEdmaConfigRcv.src = MCBSP_getRcvAddr(hMcbspData);
```

Also, don’t forget to declare the two new handles for the Rcv side in the global variables area (hEdmaRcv, hEdmaReloadRcv).
30. Add control code to initEdma()

We need to add a few lines of control code to the EDMA initialization. Refer to the diagrams in the discussion material (previous module) to review the CIPR and CIER registers. Use the following API’s just after the transmit side _link statements. Some of this code may already be present. Just make sure all 8 lines are there:

- Clear pending flags in the EDMA’s CIPR register:
  ```c
  EDMA_intClear(gRcvTCC);
  EDMA_intClear(gXmtTCC);
  ```
  *Make sure gRcvTCC is declared as a global (just like gXmtTCC)*

- Enable the interrupts from the EDMA channels (CIER register) to the CPU:
  ```c
  EDMA_intEnable(gRcvTCC);
  EDMA_intEnable(gXmtTCC);
  ```

- Hook the ISR function into the EDMA Dispatcher:
  ```c
  EDMA_intHook(gRcvTCC, edmaHwi);
  EDMA_intHook(gXmtTCC, edmaHwi);
  ```

- Enable the EDMA channels themselves (EER):
  ```c
  EDMA_enableChannel(hEdmaRcv);
  EDMA_enableChannel(hEdmaXmt);
  ```

**Note:** The EDMA_enableChannel() API enables the specified channel using the channel's handle obtained through the _open API. It does not tell the channel to start transferring. In this lab, we accomplish that by using a sync event.

Modify the EDMA’s ISR

Ok. Let’s review:

One EDMA Channel is set up to transfer elements from the receive register of the data McBSP to the Receive Buffer, gBufRcv. The sync event for this transfer is REVTx (x = 0, 1, or 2) – when the receive register is ready to read (RRDY=1). A second EDMA Channel will transfer data from the Transmit Buffer, gBufXmt when XEVTx event occurs (i.e. transmit register in the data McBSP is empty). We also wrote the code to initialize the codec. So, what else do we need to do? Modify the EDMA ISR to accomplish the following:

- When both EDMA interrupts occur (i.e. the receive buffer is full and the transmit buffer is empty), we need to copy the receive buffer (gBufRcv) contents to the transmit buffer (gBufXmt).

- Return to the while loop in main() and wait for the next interrupt

31. Remove instructions from edmaHwi() in edma.c

Locate the edmaHwi() routine. Remove the 2 instructions (SINE_blockFill, EDMA_setChannel) from the function.
Starting From a Clean Slate

We will start with a clean slate. We need to write the code that:

A. checks to see if the receive EDMA interrupt has occurred
B. checks to see if the transmit EDMA interrupt has occurred
C. when we have both receive and transmit, copies the receive buffer to the transmit buffer

32. Add two local variable to track the receive and transmit interrupts

Inside the edmaHwi(), add two static local variables to keep track of which interrupts have occurred.

```c
static int rcvDone = 0;
static int xmtDone = 0;
```

33. Write the interrupt control logic

We will use three if statements to handle the interrupt control logic. What we basically want to do is to check to see if either of the receive or transmit interrupts have occurred by testing the value of the argument passed to edmaHwi() by the EDMA dispatcher. The EDMA passes the CIPR bit of the interrupt that occurred. If either the receive or transmit interrupts have occurred, we'll set the appropriate flag that we created earlier. When both flags are set, we want to copy the data and reset both flags. The routine, copyData(), copies the data from the receive buffer to the transmit buffer. Make sure your edmaHwi code looks like the following:

```c
static int rcvDone = 0;
static int xmtDone = 0;
if (tcc == gRcvTCC) {  //tcc is passed by the dispatcher
    rcvDone = 1;
}
if (tcc == gXmtTCC) {
    xmtDone = 1;
}
if (rcvDone && xmtDone) {
    // do any processing of the data
    copyData(gBufRcv, gBufXmt, BUFFSIZE);
    rcvDone = 0;
    xmtDone = 0;
}
```

Note: We need to make sure BOTH interrupts occur. If only one has triggered, the ISR does nothing but return to the while loop and wait for the 2nd one to trigger.
**Prime the Pump**

In the previous lab, we used `EDMA_setChannel()` or `IRQ_set()` in `main()` to get the first transfer started. In this lab, how does the first transfer happen? Well, any WRITE to the codec uses the transmit side of the data McBSP. When the write to the codec completes, the transmit sync event triggers the EDMA to transfer again. So, all we need to do is write a value to the codec to get things started.

**34. Remove EDMA_setChannel() or IRQ_set() from main()**

In the previous lab, one of these two functions was used to enable the channel and tell it to start transferring. Again, due to the use of sync events, we no longer need this API. Remove or comment out the one that you used in lab 5 from `main()`.

**35. Initialize the transmit buffer to zero in main.c**

We don't want to send any garbage to the codec, so add a small `for` loop to the beginning of `main()` that zeros out the transmit buffer:

```c
for ( i = 0; i < BUFFSIZE; i++)
    gBufXmt[i] = 0;
```

You'll also need to declare a local variable `i` for the loop counter.

**36. Remove SINE_blockFill() from main()**

Since we are going to be receiving audio data directly from the codec, we no longer need to call `SINE_blockFill()` to fill the initial buffers. Comment out or remove this call from `main()`.

**37. Start the transfers**

Add the following line of code to `main()` just before the while ( ) loop to kick everything off:

```c
MCBSP_write(hMcbspData, 0);
```
**Hook up DSK/audio source, Run Audio**

38. Run the audio

Locate the audio source file on your computer and run it – making sure you have the “play forever” option selected. To prove that the audio is truly running, you may want to hook up the headphone jack of your computer directly to your speakers for a moment.

39. Hook up the DSK

Once you are sure the audio is running, hook the headphone output of your computer to the DSK line input and hook the DSK output to the speakers or headphones.

**Build and Run**

40. Header File sanity check

Before you build, you might want to check to make sure that you've added all of the appropriate header files to the appropriate source files. Here is a short list to remind you which source files should have which header files at this point. If you don't have the right header files in the right place, you can get a bunch of build errors.

<table>
<thead>
<tr>
<th>Source Files</th>
<th>Header Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>main.c</td>
<td>&quot;mcbsp.h&quot;</td>
</tr>
<tr>
<td>edma.c</td>
<td></td>
</tr>
<tr>
<td>mcbsp.c</td>
<td>&quot;sine.h&quot;</td>
</tr>
<tr>
<td>&lt;csl.h&gt;</td>
<td>&quot;csl.h&quot;</td>
</tr>
<tr>
<td>&lt;csl_edma.h&gt;</td>
<td>&quot;csl_edma.h&quot;</td>
</tr>
<tr>
<td>&lt;csl_mcbsp.h&gt;</td>
<td>&quot;csl_mcbsp.h&quot;</td>
</tr>
<tr>
<td>&quot;sine.h&quot;</td>
<td>&quot;codec.h&quot;</td>
</tr>
<tr>
<td>&quot;mcbsp.h&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;edma.h&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;mcbsp.h&quot;</td>
<td></td>
</tr>
</tbody>
</table>

Don't forget that ordering is also important with header files. For example, csl.h needs to be included before any files that are dependent on it, csl_edma.h, csl_mcbsp.h, or anything else starting with csl_*\_h.

41. Build the project and load it to the DSK

42. Run the code

You should hear audio playing from your speakers or headphones. If there is any distortion, adjust the volume level on your PC. If you get noise, go back and debug your code. Follow the data from the input/receive side to the output/transmit side. If your audio doesn’t sound good, try to find the error. If, after 5 minutes, you’re stuck, compare the solution to your code and fix the error. Sometimes copying in the mcbsp configuration from the solution helps.

If you get frustrated, ask your instructor for help.

43. Halt the processor
Part A

Note: If you had troubles getting Lab 6 to work, copy the files from \solutions\lab6 and begin working on the next step shown below.

Add the Sine Wave “noise” to the Audio Stream

44. Add the sine wave values to the incoming audio stream

In lab3, we developed the code to create a sine wave. We will now add this “noise” to the incoming (receive side) audio stream before it is copied to the transmit buffer. When played, you should hear the audio along with an annoying sine wave tone.

Inside the edmaHwi() routine in edma.c, just above the call to copyData(), add a call to SINE_addPacked() to add the sine values to gBufRcv. You can find this routine in the sine.c file. Here’s what the call looks like:

SINE_addPacked(&sineObj, gBufRcv, BUFFSIZE);

45. Change SINE_init() values

The codec uses a 48KHz sampling rate to sample the signal that we are going to send it. Right now, we are configuring the sine generator for a 8KHz sample rate. So, for the signal to sound right, we need to change its sample rate.

Find the call to SINE_init() in main.c. Change it so that the sine wave is 200Hz (NOT 256Hz) and it is sampled at 48KHz (NOT 8KHz).

Note: We used a lower sampling rate in the earlier labs so that the graphs would look better. Otherwise, you would not see a full cycle of the sine wave.

46. Rebuild, run, listen

Build and run your code and listen to the audio. Do you hear the sine wave? If not, debug and try again. In a following lab, we’ll add a switch to turn the sine wave on and off.

47. Halt the processor.

48. Copy project to preserve your solution.

Using Windows Explorer, copy the contents of:
c:\iw6000\labs\audioapp\*.* TO c:\iw6000\labs\lab6

You’re done
Optional Topics

DMA vs EDMA: Event Synchronization

- Is the DAC as fast as the EDMA?
  No, the EDMA needs to be sync'd up to the DAC.
- Unlike the EDMA, any DMA channel can be sync'd to and EDMA event.

Frame Synchronization

- Similar to FS on the EDMA
- Unlike the EDMA, though, there is not block-level (2D) synchronization
Optional Topics

DMA Split Mode

- Split mode allows one DMA Channel to handle both rcv/xmt

4 addresses are needed when handling receive & transmit parts of a serial port, unfortunately the DMA only has two address registers. This is solved by:
1. Select SPLIT mode in Primary Control Register
2. Source/Destination registers contain the From/To memory addresses
3. Use global reg (A, B, or C) for address of McBSP's DRR register.
  DMA split mode knows to find the DXR address in the next word location.

**DMA vs EDMA: Updated Summary**

**DMA / EDMA Comparison**

<table>
<thead>
<tr>
<th>Features</th>
<th>DMA</th>
<th>C67x EDMA</th>
<th>C64x EDMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channels</td>
<td>4 channels + 1 for HPI</td>
<td>16 channels + 1 for HPI + Q-DMA</td>
<td>64 channels + 1 for HPI + Q-DMA</td>
</tr>
<tr>
<td>Sync</td>
<td>element, frame</td>
<td>element, frame, 2D (block)</td>
<td></td>
</tr>
<tr>
<td>Sync Events</td>
<td>Any channel can use any event</td>
<td>Each channel has specific event</td>
<td></td>
</tr>
<tr>
<td>CPU Interrupts</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Interrupt Conditions</td>
<td>six: 3 for Count, 3 for errors</td>
<td>Count = 0</td>
<td></td>
</tr>
<tr>
<td>Reload (Auto-Init)</td>
<td>~2</td>
<td>69</td>
<td>21</td>
</tr>
<tr>
<td>Chain Channels</td>
<td>None</td>
<td>4 channels (8-11)</td>
<td>64 channels</td>
</tr>
<tr>
<td>Priority</td>
<td>4 fixed levels</td>
<td>2 prog levels</td>
<td>4 prog levels</td>
</tr>
<tr>
<td>McBSP Operation</td>
<td>Split Mode</td>
<td>Uses two EDMA channels</td>
<td></td>
</tr>
</tbody>
</table>