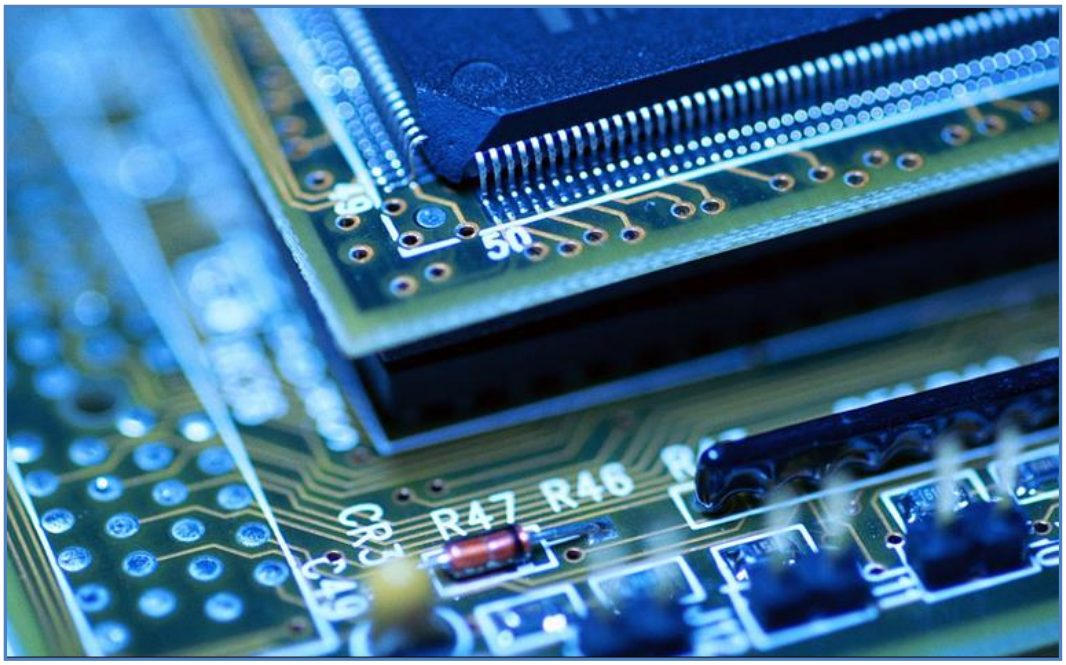
EMBEDDED ADVANTAGE

IN ASSOCIATION WITH TEXAS INSTRUMENTS

LAB 05W: EDMA Audio TUTORIAL



Version 2.0
2022.02.11
www.EmbeddedAdvantage.com

Lab Project Name: `lab_05W_edmaAUDIOFinalSolution`

Introduction

In this lab, you will learn how to use the TMDSLCDK6748 board for an audio application. By the end of this lab you will be able to receive an audio signal (through the input audio jack), apply a FIR (finite impulse response) filter to the audio signal, and output the audio signal (through the output audio jack). Throughout this lab you will use the I2C and McASP peripherals of the C6748 processor to communicate with audio CODEC available on the TMDSLCDK6748 board to receive and transmit audio packets. Finally, the importance of the EDMA3 peripheral in transferring the audio packets to and from the McASP peripheral.

Prerequisites

It is very important that you understand the concepts covered in the videos for the EDMA3 peripheral before starting this lab.

Learning Objectives

- Configure and use the I2C peripheral
- Configure and use the AIC3106 low-power stereo CODEC through the I2C peripheral
- Configure and use the McASP peripheral to send and receive audio packets
- Configure and use the EDMA3 peripheral to transfer data to and from the McASP peripheral
- Implement and use a FIR filter on the audio signal

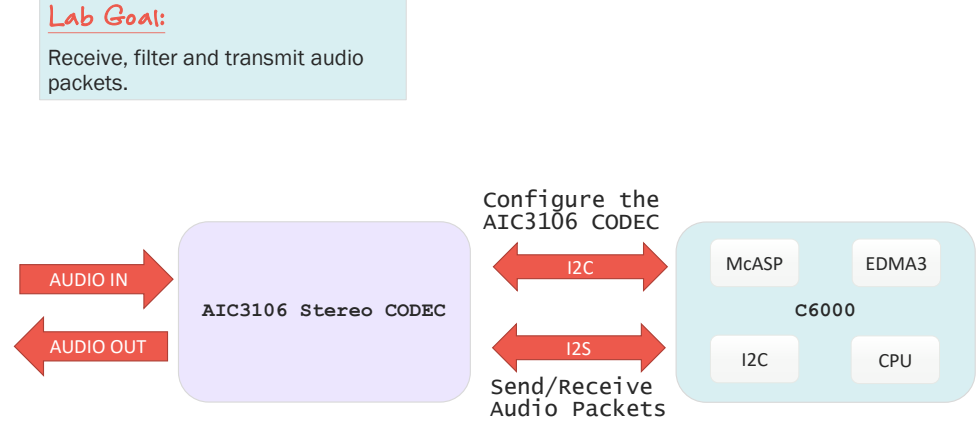


Table of Contents

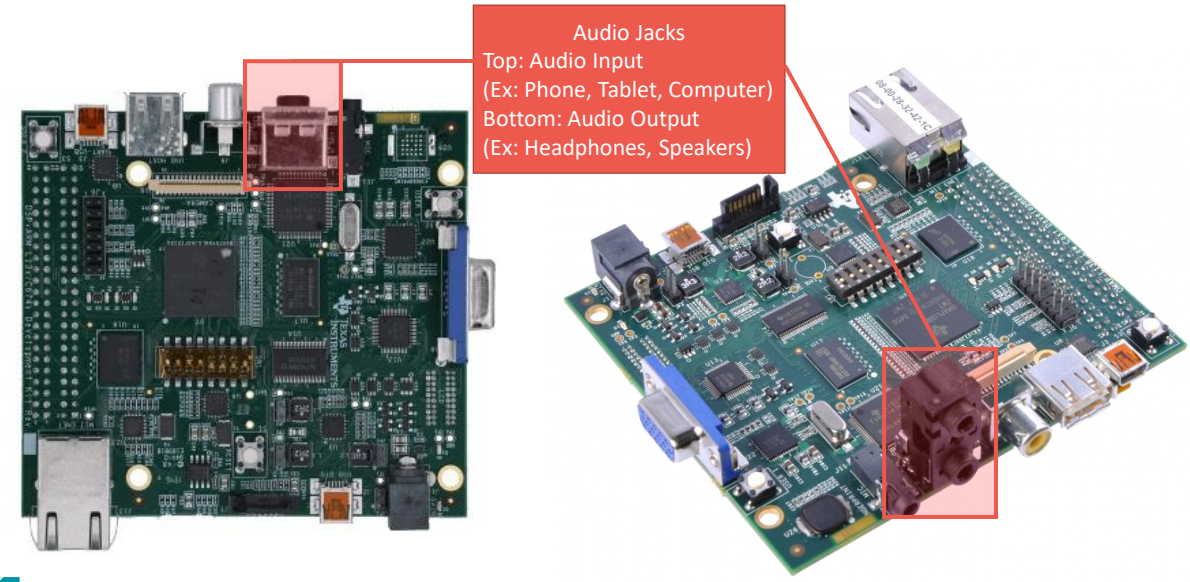
Project Overview

The goal of this project is to show how the show how the EDMA3 peripheral can be used to transfer data to and from the McASP peripheral in a stereo audio application. Along the way we will use other peripherals such as I2C, which is used to configure AIC3106 (external stereo audio CODEC device).

Hardware Overview

On the TMDSLCDK6748, you can find all the hardware required for an audio filtering application. There are two audio jacks, one for audio input and one for audio output. The stacked audio jacks are shown below:

Hardware Overview Audio Jacks

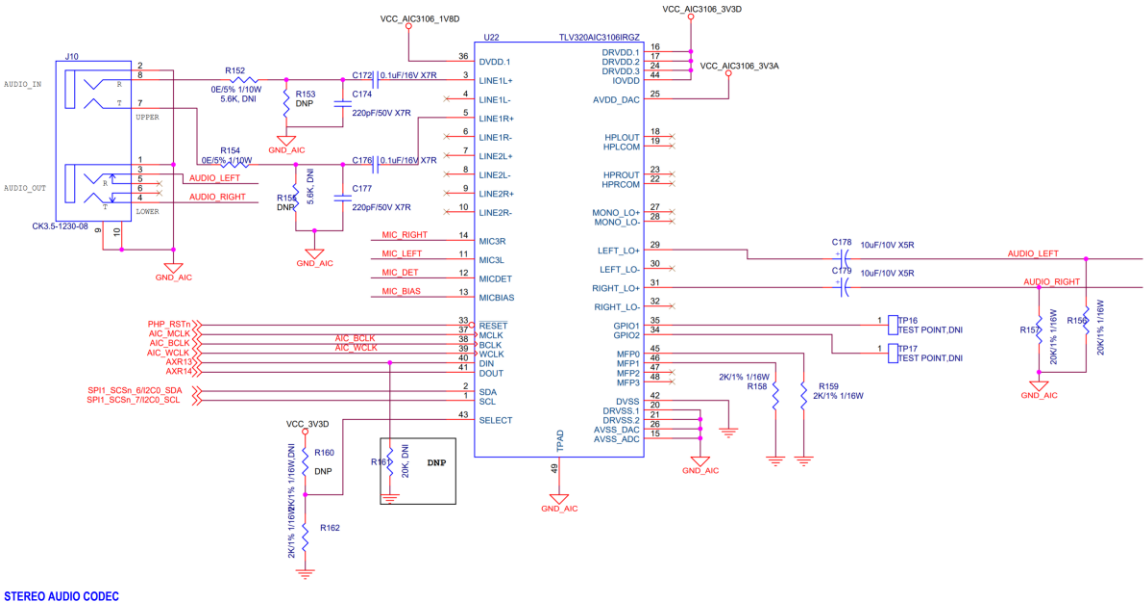


The schematic of the board can be found at:

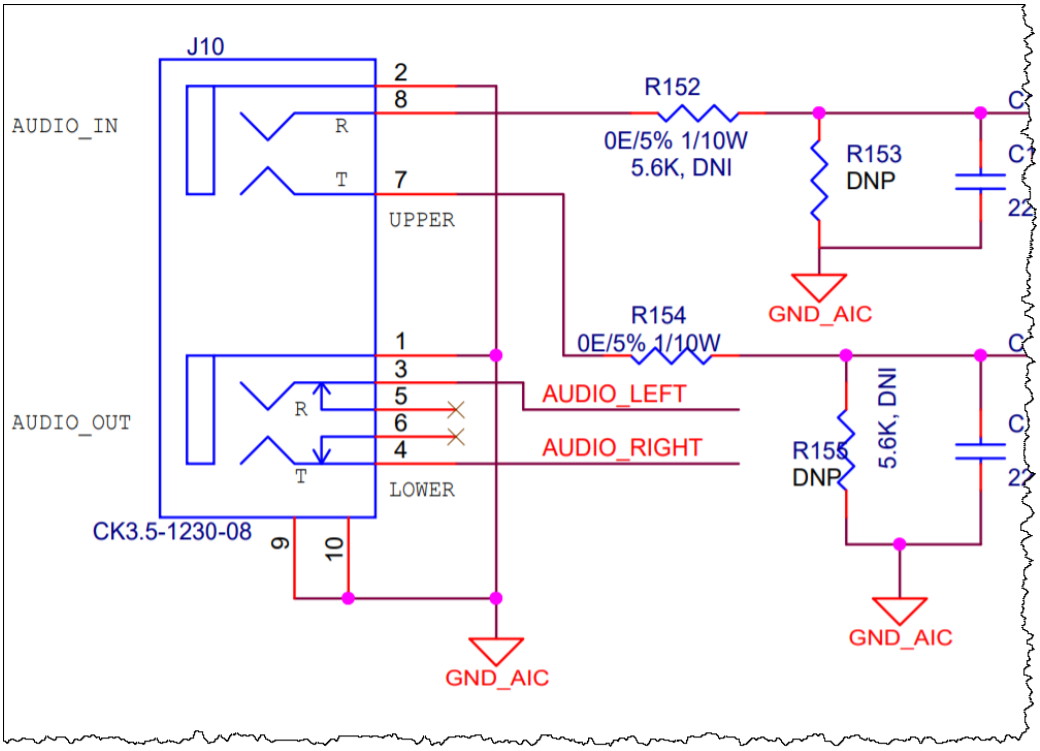
http://www.ti.com/lit/zip/sprcaf4

The pdf file named `C6748 LC DEV KIT VER  A7E.pdf` contains the schematic of the board.

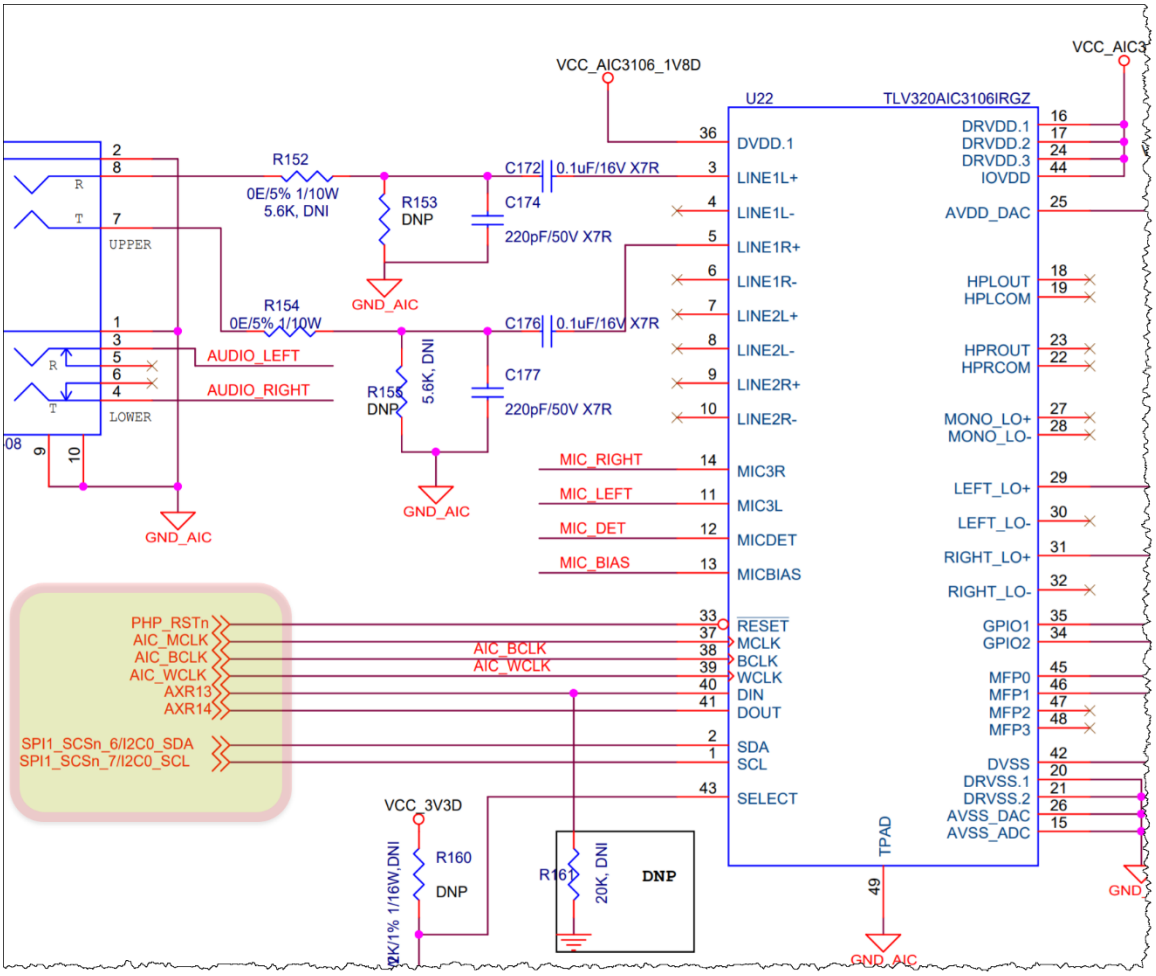
The audio section of the schematic shows how the audio jacks are connected to A3106 CODEC.



At the top left of the **STERIO AUDIO CODEC** schematic, you can see that the top audio jack is for audio input and the bottom audio jack is for audio output.

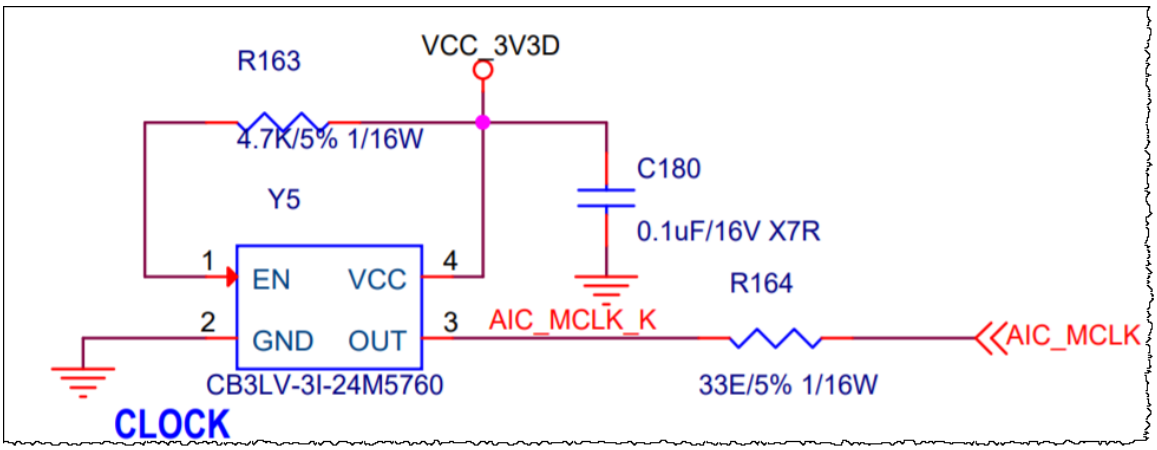


The AIC3106 CODEC and its connections are important when it comes to writing the code which interfaces with this device. One the most important part of the schematic is the signal at the bottom left corner of the AIC3106 CODEC.



PIN1 and PIN2 of the AIC3106 CODEC are connected to I2C0 peripheral of the C6748 DSP. The I2C0 peripheral is used to configure the settings of the AIC3106 CODEC. Pin 40 and 41 are the I2S data input (DIN) and data output (DOUT) of the AIC3106 CODEC and they are connected to AXR13 and AXR14 of the McASP peripheral of the C6748 DSP. AXR13 and AXR14 correspond to serializers 13 and 14 of the McASP peripheral of the C6748 DSP.

Right below the **STEREO AUDIO CODEC** section of the schematic, you can view a section named **CLOCK.** This section shows how the clock for the I2S communication is generated.



The AIC3106 CODEC documentation can be found at:

https://www.ti.com/product/TLV320AIC3106

Import the Starter Project

You must now import the author's starter project which will act as a base for the rest of the lab.

1. **Open CCS and connect your hardware target board to your computer.**
2. **Import the starter lab.**

   If you forgot how to do this, please refer back to a previous lab's PDF.

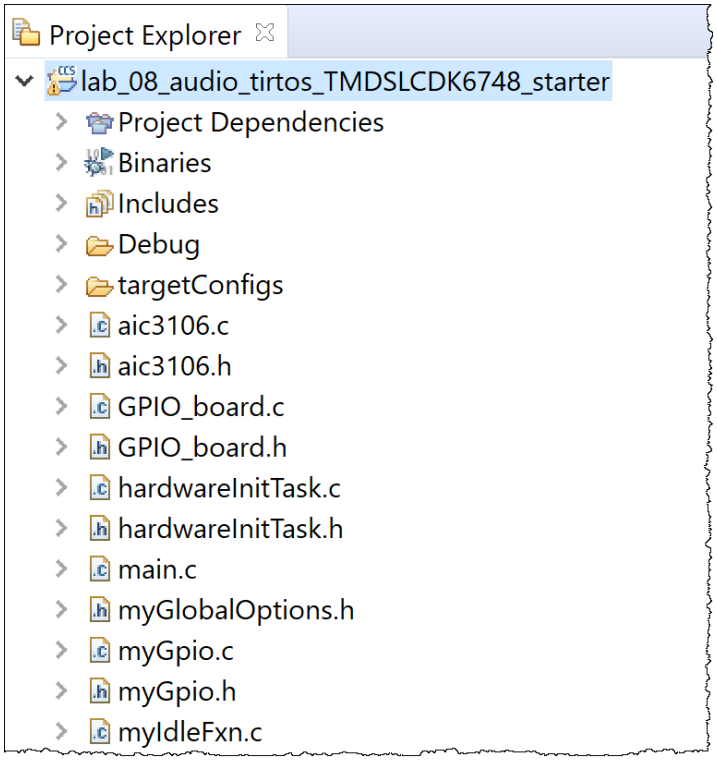
   - Import the `lab_08_audio_tirtos_starter`

Inspect the Starter Code

The starter project contains the base code we will build on to create our audio application. Let's review the files that exist in the starter project and their content.

3. **Inspect the files in the Project Explorer.**

   - Open the project in the starter project in the Project Explorer. It should have the following content:



The project already contains some of the necessary files which we will need to modify and our content to.

The following files are present in the project:

- `aic3106.c`: This file will contain all of our I2C and AIC3106 CODEC setting configuration code.
- `aic3106.h`: This file will contain all of the outward facing content for configuring the AIC3106 CODEC.
- `myGpio.c, myGpio.h, GPIO_board.c` and `GPIO_board.h`: We will not modify these files. These file contain the GPIO driver code.
- `hardwareInitTask.c`: In this file we will implement a ***Task*** which will initialize the peripherals used for our audio application.
- `hardwareInitTask.h`: This file contains the outward facing function prototype for our peripheral initialization ***Task***.
- `main.c`: In this file we will create all the ***Tasks, HWIs*** and ***Semaphores*** and start BIOS.
- `myIdleFxn.c`: This file contains an empty Idle ***Task*** function.

We will implement our audio application in multiple step, adding one feature per step.

- **Lab 08a**: Add I2C support code.
- **Lab 08b**: Add code to configure the AIC3106 CODEC through I2C.
- **Lab 08c**: Add code to configure the McASP peripheral and communicate with the AIC3106 CODEC to send/receive audio packets to ISR and CPU intervention.
- **Lab 08d**: Add a ***Task*** which will toggle and LED. The audio loopback will fail after a short time due to McASP underflow/overflow because the CPU cannot service the McASP buffers in time.
- **Lab 08e**: Add EDMA3 support to transfer the data to and from the McASP buffers, reducing the load of the CPU.
- **Lab 08f**: Add FIR filter code to filter the audio packets.

Adding I2C Support (lab_08a_i2cAudio)

In this part of the lab we must do the following:

1. Create a high priority ***Task*** object and assign the function `hardwareInitTask` to it.
2. Create an ***Hwi*** for the I2C ISR.
3. Create I2C initialization and communication code.
4. Create an ISR (interrupt service routine) for the I2C.
5. Initialize the I2C peripheral in the `hardwareInitTask`.

We have a lot to do so let's begin…

1. Create the BIOS Hwi and Task Modules

4. Add the *Task* module for **`hardwareInitTask`**.

- Open `main.c` and scroll down to the `main` function.

```
35 /*************************************************************************
36 ** Global Variables
37 *************************************************************************/
38
39 /*************************************************************************
40 ** main()
41 *************************************************************************/
42 int main(void)
43 {
44
45     // Initialize Peripherals
46     Board_init(BOARD_INIT_PINMUX_CONFIG |
47                BOARD_INIT_MODULE_CLOCK);
48     myGpio_init();
49
50     BIOS_start();
51
52 }
53
```

- Create the required `Handle` and `Param` variables for our new `hardwareInitTask` and the I2C ISR (Don't forget the Error Block 😉).

```
38 /*************************************************
39 ** Global Variables
40 *************************************************/
41 Task_Handle         hardwareInitTask;
42 Task_Params         hardwareInitTaskParams;
43
44 Hwi_Handle          hwiI2CCODEC;
45 Hwi_Params          hwiI2CCODECParams;
46
47 Error_Block         hwiEb;
48
```

- Add the following code in the main function before the BIOS_start function to create the ***Hwi*** and ***Task*** objects.

```
52    // Initialize Peripherals
53    Board_init(BOARD_INIT_PINMUX_CONFIG |
54               BOARD_INIT_MODULE_CLOCK);
55    myGpio_init();
56
57    Error_init(&hwiEb);
58    Hwi_Params_init(&hwiI2CCODECParams);
59    hwiI2CCODECParams.eventId = CSL_INTC_EVENTID_I2CINT0;
60    hwiI2CCODEC = Hwi_create(
61                            C674X_MASK_INT12,
62                            (Hwi_FuncPtr)I2CCodecIsr,
63                            &hwiI2CCODECParams,
64                            &hwiEb
65    );
66    if (hwiI2CCODEC == NULL) {
67        System_abort("I2C HWI create failed");
68    }
69
70    // Create Task(s)
71
72    Task_Params_init(&hardwareInitTaskParams);
73    hardwareInitTaskParams.priority = 8;
74    hardwareInitTaskParams.stackSize = 2048;
75
76    hardwareInitTask = Task_create(
77            (Task_FuncPtr)hardwareInitTaskFxn,
78            &hardwareInitTaskParams, Error_IGNORE);
79    if (hardwareInitTask == NULL) {
80        System_abort("HW INIT Task create failed");
81    }
82
83    BIOS_start();
```

For the I2C ***Hwi*** we set the `eventId` to be the I2C0 INT event ID, The file `cslr_interrupt.h` can shows you the listing of interrupt symbols and their Event Id. We use **INT12** of the C6748 DSP for the I2C interrupt and assign a function name `I2CCodecIsr` as the ISR handler. We will later define this function.

For the `hardwareInitTask`, we create a high priority (8, make sure all other task we create are lower priority than 8 because we need this task to run and finish first, before yielding to the other tasks). The `stackSize` of the task is set to 2048 and the function for the task is set to `hardwareInitTaskFxn`. This function is defined in `harwareInitTask.c`.

- Add the required #include files which have the prototypes for the ***Hwi*** and ***Task*** functions.

```
31 // Application Header Files
32 #include "myGlobalOptions.h"
33 #include "myGpio.h"
34
35 #include "hardwareInitTask.h"
36 #include "aic3106.h"
37
38 /*************************************
39 ** Global Variables
40 *************************************
```

We are now done with the `main.c` file and can move on…

2. The I2C Configuration and Communication Code

Now we have to add our I2C code into the aic3106.c and aic3106.h files. In this step of the lab we will only create the I2C interface functions to send and receive data, along with the ISR handler function.

Here is a list of the functions that we need to create:

- I2CCodecIfInit: Initialize the I2C peripheral for communicate with the A3106 CODEC.
- I2CCodecSendBlocking: Send data through I2C and wait for completion.
- I2CCodecRcvBlocking: Receive data through I2C and wait for completion.
- I2CDelay: Delay in blocking mode through a for loop and not signal BIOS, causing a HALT.
- I2CCodecIsr: ISR Handler for the I2C peripheral.

To use the I2C peripheral to read a slave device's register, in our case the AIC3106 CODEC, we need implement the following functions:

- CodecRegWrite: Write to a specific register of a slave device (AIC3106 CODEC) through I2C.
- CodecRegRead: Reads a specific register of a slave device (AIC3106 CODEC) through I2C
- CodecRegBitSet: Sets a specific bit of a register for a slave device (AIC3106 CODEC).
- CodecRegBitClr: Clears a specific bit of a register for a slave device (AIC3106 CODEC).

We will start with the simplest function to implement and move on from there.

5. **Implement the I2C functions.**

   ▶ Open the aic3106.c file.

   ▶ At the top of the file, after your `#include` section, add the following:

```
12 #include <ti/csl/src/ip/i2c/V0/i2c.h>
13
14 #include "myGlobalOptions.h"
15 #include "aic3106.h"
16
17 /*****************************************************************************
18 **                      INTERNAL FUNCTION PROTOTYPES
19 *****************************************************************************/
20 static void I2CCodecSendBlocking(unsigned int baseAddr, unsigned int dataCnt);
21 static void I2CCodecRcvBlocking(unsigned int baseAddr, unsigned int dataCnt);
22
23 /*
24 ** Function to be used by codecs
25 */
26 void CodecRegWrite(unsigned int baseAddr, unsigned char regAddr,
27                            unsigned char regData);
28 void CodecRegBitSet(unsigned int baseAddr, unsigned char regAddr,
29                             unsigned char bitMask);
30 void CodecRegBitClr(unsigned int baseAddr, unsigned char regAddr,
31                             unsigned char bitMask);
32 unsigned char CodecRegRead(unsigned int baseAddr,
33                                    unsigned char regAddr);
34
35 /*****************************************************************************
36 **                      INTERNAL VARIABLE DEFINITIONS
37 *****************************************************************************/
38 volatile unsigned int dataIdx = 0;
39 volatile unsigned int txCompFlag = 1;
40 volatile unsigned int slaveData[3];
41 unsigned int savedBase;
42
```

We will implement the some functions, `I2CCodecSendBlocking`, `I2CCodecRcvBlocking` and others in a few steps.

We also created some global variables which we will use to share data between our Send/Receive blocking functions and the ISR handler. This is a common way of handling I2C communication.

- Add the following code for the I2C delay function:

```
55 /*******************************
56 ** Delay
57 *******************************/
58 static void I2CDelay(void)
59 {
60     volatile int i;
61     for(i=0; i<20000; i++);
62 }
```

- Add the code for the I2C initialization:

```
/*****************************************************************
** Initializes the I2C interface for a codec
*****************************************************************/
void I2CCodecIfInit(unsigned int baseAddr, unsigned int intCh,
                    unsigned int slaveAddr)
{
    /* Put i2c in reset/disabled state */
    I2CMasterDisable(baseAddr);

    /* Configure i2c bus speed to 100khz */
    I2CMasterInitExpClk(baseAddr, 24000000, 8000000, 100000);

    /* Set i2c slave address */
    I2CMasterSlaveAddrSet(baseAddr, slaveAddr);

    I2CMasterEnable(baseAddr);
}
```

This function disables the I2C peripheral while the initialization is taking place and enables the I2C peripheral once the initialization is completed.

- Next implement the I2C blocking data transmission/reception code:

```
static void I2CCodecSendBlocking(unsigned int baseAddr, unsigned int dataCnt)
{
    txCompFlag = 1;
    dataIdx = 0;
    savedBase = baseAddr;

    I2CSetDataCount(baseAddr, dataCnt);

    I2CMasterControl(baseAddr, I2C_CFG_MASK_TX | I2C_CFG_MASK_STOP,
            I2C_CFG_CMD_TX | I2C_CFG_CMD_STOP);

    I2CMasterIntEnableEx(baseAddr, I2C_INT_TRANSMIT_READY | I2C_INT_STOP_CONDITION);

    I2CMasterStart(baseAddr);

    /* Wait till the data is sent */
    //while(txCompFlag);
    I2CDelay();
}


/*****************************************************************************
** Function to receive data from the Codec through I2C bus
*****************************************************************************/
static void I2CCodecRcvBlocking(unsigned int baseAddr, unsigned int dataCnt)
{
    txCompFlag = 1;
    dataIdx = 0;
    savedBase = baseAddr;

    I2CSetDataCount(baseAddr, dataCnt);

    I2CMasterControl(baseAddr, I2C_CFG_MASK_RX | I2C_CFG_MASK_STOP,
            I2C_CFG_CMD_RX | I2C_CFG_CMD_STOP);

    I2CMasterIntEnableEx(baseAddr, I2C_INT_RECV_READY | I2C_INT_STOP_CONDITION);

    I2CMasterStart(baseAddr);

    /* Wait till data is received fully */
    //while(txCompFlag);
    I2CDelay();
}
```

- Add the interrupt service routine for the I2C peripheral:

```
/*****************************************************************************
** ISR to handler i2c interrupts
*****************************************************************************/
void I2CCodecIsr(void)
{
    unsigned int intCode = 0;
    unsigned int sysIntNum = 0;

    /* Get interrupt vector code */
    intCode = I2CIntVectGet(savedBase);
    sysIntNum = BOARD_I2C_INTR_NUM;

    while(intCode!=0)
    {

        /* Clear status of interrupt */
        IntEventClear(sysIntNum);

         if (intCode == I2C_IVR_INTCODE_XRDY)
         {
              I2CMasterDataPut(savedBase, slaveData[dataIdx]);
              dataIdx++;
         }

         if(intCode == I2C_IVR_INTCODE_RRDY)
         {
              slaveData[dataIdx] = I2CMasterDataGet(savedBase);
              dataIdx++;
         }

         if (intCode == I2C_IVR_INTCODE_SCD)
         {
              /* Disable transmit data ready and receive data read interupt */
              I2CMasterIntDisableEx(savedBase, I2C_INT_TRANSMIT_READY
                                               | I2C_INT_RECV_READY);
              txCompFlag = 0;
         }

         intCode = I2CIntVectGet(savedBase);
    }
}
```

Now that we are done with the raw I2C communication code, we will move on to implement another software layer on top our raw I2C code to read/write to the registers of a slave device (in our case AIC3106 CODEC).

6. Implement the slave device register access code.

   - Add the following code to the bottom of the aic3106.c file to implement the full register read/write access for a slave device using I2C.

```
155 /*****************************************************************************
156 ** Writes a codec register with the given data value
157 *****************************************************************************/
158 void CodecRegWrite(unsigned int baseAddr, unsigned char regAddr,
159                    unsigned char regData)
160 {
161     /* Send the register address and data */
162     slaveData[0] = regAddr;
163     slaveData[1] = regData;
164
165     I2CCodecSendBlocking(baseAddr, 2);
166 }
167
168 /*****************************************************************************
169 ** Reads codec register
170 *****************************************************************************/
171 unsigned char CodecRegRead(unsigned int baseAddr, unsigned char regAddr)
172 {
173     /* Send the register address */
174     slaveData[0] = regAddr;
175     I2CCodecSendBlocking(baseAddr, 1);
176
177     /* Receive the register contents in slaveData */
178     I2CCodecRcvBlocking(baseAddr, 1);
179
180     return (slaveData[0]);
181 }
```

To write to a register of a slave device, you must send two bytes of data through I2C. The first byte includes the address of the register to be written to and the second byte is the value you wish to write to the register.

To read from a register of a slave device, you must write one byte, the address of the register you wish to access and then read one byte which will contain the value of the requested register.

- Add the following code to set/clear a specific bit of a register in the slave device:

```
183 /*****************************************************************
184 ** Sets codec register bit specified in the bit mask
185 *****************************************************************
186 void CodecRegBitSet(unsigned int baseAddr, unsigned char regAddr,
187                     unsigned char bitMask)
188 {
189     /* Send the register address */
190     slaveData[0] = regAddr;
191     I2CCodecSendBlocking(baseAddr, 1);
192
193     /* Receive the register contents in slaveData */
194     I2CCodecRcvBlocking(baseAddr, 1);
195
196     slaveData[1] =  slaveData[0] | bitMask;
197     slaveData[0] = regAddr;
198
199     I2CCodecSendBlocking(baseAddr, 2);
200 }
201
202
203 /*****************************************************************
204 ** Clears codec register bits specified in the bit mask
205 *****************************************************************
206 void CodecRegBitClr(unsigned int baseAddr, unsigned char regAddr,
207                     unsigned char bitMask)
208 {
209     /* Send the register address */
210     slaveData[0] = regAddr;
211     I2CCodecSendBlocking(baseAddr, 1);
212
213     /* Receive the register contents in slaveData */
214     I2CCodecRcvBlocking(baseAddr, 1);
215
216     slaveData[1] =  slaveData[0] & ~bitMask;
217     slaveData[0] = regAddr;
218
219     I2CCodecSendBlocking(baseAddr, 2);
220
221 }
```

These functions are very similar to the full register read/write functions. The only difference being that for each function, first, the value of the slave device's register must be read, then the bits of the received value are manipulated through software, and finally written back to the slave device.

That is it for the `aic3106.c` file, FOR NOW. From the functions we implemented, we must now add the function prototypes for the outward facing functions to the `aic3106.h` file.

7. **Add the I2C function prototypes to the `aic3106.h` file.**

   ▶ Open the `aic3106.h` file and add the following code:

aic3106.c aic3106.h

```
/*****************************************************************************
**                              CODEC
*****************************************************************************/

#ifndef _AIC3106_H_
#define _AIC3106_H_

/*****************************************************************************
**                         Macro Definitions
*****************************************************************************/

/*
** Macros for configuring the interface Type
*/
#define BOARD_I2C_INTR_NUM              (CSL_INTC_EVENTID_I2CINT0)

/*****************************************************************************
**                        API function Prototypes
*****************************************************************************/
extern void I2CCodecIsr(void);
extern void I2CCodecIfInit(unsigned int baseAddr, unsigned int intCh,
                           unsigned int slaveAddr);

#endif

```

3. Update the hardwareInitTask Function

Now that we have our I2C code in place, we should call the `I2CCodecIfInit` function in our `hardwareInitTaskFxn` function.

8. Call the **`I2CCodecIfInit`** to the **`hardwareInitTaskFxn`** task function.

   - Open the `hardwareInitTask.c` file.
   - Update `hardwareInitTaskFxn` to call the `I2CCodecIfInit` function.

aic3106.c aic3106.h hardwareInitTask.c

```
/*****************************************************************************
** hardwareInitTask.c
*****************************************************************************/

#include "myGlobalOptions.h"

#include "aic3106.h"

#include "hardwareInitTask.h"

/*****************************************************************************
** hardwareInitTask()
*****************************************************************************/
void hardwareInitTaskFxn (void)
{
    /* Initialize the I2C 0 interface for the codec AIC31 */
    I2CCodecIfInit(CSL_I2C_0_DATA_CFG, INT_CHANNEL_I2C, I2C_SLAVE_CODEC_AIC31);
}

```

Build!

9. Build, to make sure everything compiles successfully.
   - Build your application.
   - If there are any build errors, fix them.

Adding AIC3106 Support (lab_08b_aic3106Audio)

In this part of the lab we will add the code for configuring the AIC3106 CODEC on TMDSLCDK6748 board. We will use the I2C code we wrote in the previous section and build on top of that to configure the AIC3106 device.

AIC3106 Configuration Code

Let's begin by updating the `aic3106.h` file and adding a function prototype for the outward facing AIC3106 CODEC function.

10. Add the prototype for `AIC31I2SConfigure` function to the `aic3106.h` file.

- Open `aic3106.h` file.
- Add the following `AIC31I2SConfigure` function prototype to the `aic3106.h` function.

```
17 /*****************************************************************************
18 **                      API function Prototypes
19 *****************************************************************************/
20 extern void I2CCodecIsr(void);
21 extern void I2CCodecIfInit(unsigned int baseAddr, unsigned int intCh,
22                            unsigned int slaveAddr);
23
24 extern void AIC31I2SConfigure(void);
25
26 #endif
27
```

Next we move on to implement the `AIC31I2SConfigure` function and more in the `aic3106.c` file.

11. Add the support code for AIC3106 CODEC to the `aic3106.h` file.

- Open the `aic3106.c` file.
- Add the following #define statements to the top of the `aic3106.h` file.

```
/*****************************************************************************
**                        Macro Definitions
*****************************************************************************/
/*
** Macros for the dataType variable to pass to AIC31DataConfig function
*/
#define AIC31_DATATYPE_I2S              (0u << 6u) /* I2S Mode */
#define AIC31_DATATYPE_DSP              (1u << 6u) /* DSP Mode */
#define AIC31_DATATYPE_RIGHTJ           (2u << 6u) /* Right Aligned Mode */
#define AIC31_DATATYPE_LEFTJ            (3u << 6u) /* Left Aligned Mode */

/*
** Macros for the mode variable for the AIC31SampleRateConfig function
*/
#define AIC31_MODE_ADC                  (0xF0u)
#define AIC31_MODE_DAC                  (0x0Fu)
#define AIC31_MODE_BOTH                 (0xFFu)

/*
** Register Address for AIC31 Codec
*/
#define AIC31_P0_REG0               (0)  /* Page Select  */
#define AIC31_P0_REG1               (1)  /* Software Reset */
#define AIC31_P0_REG2               (2)  /* Codec Sample Rate Select */
#define AIC31_P0_REG3               (3)  /* PLL Programming A */
#define AIC31_P0_REG4               (4)  /* PLL Programming B */
#define AIC31_P0_REG5               (5)  /* PLL Programming C */
#define AIC31_P0_REG6               (6)  /* PLL Programming D */
#define AIC31_P0_REG7               (7)  /* Codec Datapath Setup */
#define AIC31_P0_REG8               (8)  /* Audio Serial Data I/f Control A */
#define AIC31_P0_REG9               (9)  /* Audio Serial Data I/f Control B */
#define AIC31_P0_REG10              (10) /* Audio Serial Data I/f Control C */
#define AIC31_P0_REG11              (11) /* Audio Codec Overflow Flag */
#define AIC31_P0_REG12              (12) /* Audio Codec Digital Filter Ctrl */
#define AIC31_P0_REG13              (13) /* Headset / Button Press Detect A */
#define AIC31_P0_REG14              (14) /* Headset / Button Press Detect B */
#define AIC31_P0_REG15              (15) /* Left ADC PGA Gain Control */
#define AIC31_P0_REG16              (16) /* Right ADC PGA Gain Control */
#define AIC31_P0_REG17              (17) /* MIC3L/R to Left ADC Control */
#define AIC31_P0_REG18              (18) /* MIC3L/R to Right ADC Control */
#define AIC31_P0_REG19              (19) /* LINE1L to Left ADC Control */
#define AIC31_P0_REG20              (20) /* LINE2L to Left ADC Control */
#define AIC31_P0_REG21              (21) /* LINE1R to Left ADC Control */
#define AIC31_P0_REG22              (22) /* LINE1R to Right ADC Control */
#define AIC31_P0_REG23              (23) /* LINE2R to Right ADC Control */
#define AIC31_P0_REG24              (24) /* LINE1L to Right ADC Control */
#define AIC31_P0_REG25              (25) /* MICBIAS Control */
#define AIC31_P0_REG26              (26) /* Left AGC Control A */
#define AIC31_P0_REG27              (27) /* Left AGC Control B */
#define AIC31_P0_REG28              (28) /* Left AGC Control C */
#define AIC31_P0_REG29              (29) /* Right AGC Control A */
#define AIC31_P0_REG30              (30) /* Right AGC Control B */
#define AIC31_P0_REG31              (31) /* Right AGC Control C */
#define AIC31_P0_REG32              (32) /* Left AGC Gain */
#define AIC31_P0_REG33              (33) /* Right AGC Gain */
#define AIC31_P0_REG34              (34) /* Left AGC Noise Gate Debounce */
#define AIC31_P0_REG35              (35) /* Right AGC Noise Gate Debounce */
#define AIC31_P0_REG36              (36) /* ADC Flag */
#define AIC31_P0_REG37              (37) /* DAC Power and Output Driver Control */
#define AIC31_P0_REG38              (38) /* High Power Output Driver Control*/
#define AIC31_P0_REG40              (40) /* High Power Output Stage Control*/
#define AIC31_P0_REG41              (41) /* DAC Output Switching Control */
#define AIC31_P0_REG42              (42) /* Output Driver Pop Reduction */
#define AIC31_P0_REG43              (43) /* Left DAC Digital Volume Control */
#define AIC31_P0_REG44              (44) /* Right DAC Digital Volume Control */
#define AIC31_P0_REG45              (45) /* LINE2L to HPLOUT Volume Control */
#define AIC31_P0_REG46              (46) /* PGA_L to HPLOUT Volume Control */
#define AIC31_P0_REG47              (47) /* DAC_L1 to HPLOUT Volume Control */
#define AIC31_P0_REG48              (48) /* LINE2R to HPLOUT Volume Control */
#define AIC31_P0_REG49              (49) /* PGA_R to HPLOUT Volume Control */
#define AIC31_P0_REG50              (50) /* DAC_R1 to HPLOUT Volume Control */
```

```
#define AIC31_P0_REG51              (51) /* HPLOUT Output Level Control */
#define AIC31_P0_REG52              (52) /* LINE2L to HPLCOM Volume Control */
#define AIC31_P0_REG53              (53) /* PGA_L to HPLCOM Volume Control */
#define AIC31_P0_REG54              (54) /* DAC_L1 to HPLCOM Volume Control */
#define AIC31_P0_REG55              (55) /* LINE2R to HPLCOM Volume Control */
#define AIC31_P0_REG56              (56) /* PGA_R to HPLCOM Volume Control */
#define AIC31_P0_REG57              (57) /* DAC_R1 to HPLCOM Volume Control */
#define AIC31_P0_REG58              (58) /* HPLCOM Output Level Control */
#define AIC31_P0_REG59              (59) /* LINE2L to HPROUT Volume Control */
#define AIC31_P0_REG60              (60) /* PGA_L to HPROUT Volume Control */
#define AIC31_P0_REG61              (61) /* DAC_L1 to HPROUT Volume Control */
#define AIC31_P0_REG62              (62) /* LINE2R to HPROUT Volume Control */
#define AIC31_P0_REG63              (63) /* PGA_R to HPROUT Volume Control  */
#define AIC31_P0_REG64              (64) /* DAC_R1 to HPROUT Volume Control */
#define AIC31_P0_REG65              (65) /* HPROUT Output Level Control */
#define AIC31_P0_REG66              (66) /* LINE2L to HPRCOM Volume Control  */
#define AIC31_P0_REG67              (67) /* PGA_L to HPRCOM Volume Control */
#define AIC31_P0_REG68              (68) /* DAC_L1 to HPRCOM Volume Control */
#define AIC31_P0_REG69              (69) /* LINE2R to HPRCOM Volume Control */
#define AIC31_P0_REG70              (70) /* PGA_R to HPRCOM Volume Control */
#define AIC31_P0_REG71              (71) /* DAC_R1 to HPRCOM Volume Control */
#define AIC31_P0_REG72              (72) /* HPRCOM Output Level Control */
#define AIC31_P0_REG73              (73) /* LINE2L to MONO_LOP/M Volume Control*/
#define AIC31_P0_REG74              (74) /* PGA_L to MONO_LOP/M Volume Control */
#define AIC31_P0_REG75              (75) /* DAC_L1 to MONO_LOP/M Volume Control */
#define AIC31_P0_REG76              (76) /* LINE2R to MONO_LOP/M Volume Control */
#define AIC31_P0_REG77              (77) /* PGA_R to MONO_LOP/M Volume Control */
#define AIC31_P0_REG78              (78) /* DAC_R1 to MONO_LOP/M Volume Control */
#define AIC31_P0_REG79              (79) /* MONO_LOP/M Output Level Control */
#define AIC31_P0_REG80              (80) /* LINE2L to LEFT_LOP/M Volume Control */
#define AIC31_P0_REG81              (81) /* PGA_L to LEFT_LOP/M Volume Control */
#define AIC31_P0_REG82              (82) /* DAC_L1 to LEFT_LOP/M Volume Control */
#define AIC31_P0_REG83              (83) /* LINE2R to LEFT_LOP/M Volume Control */
#define AIC31_P0_REG84              (84) /* PGA_R to LEFT_LOP/M Volume Control */
#define AIC31_P0_REG85              (85) /* DAC_R1 to LEFT_LOP/M Volume Control */
#define AIC31_P0_REG86              (86) /* LEFT_LOP/M Output Level Control */
#define AIC31_P0_REG87              (87) /* LINE2L to RIGHT_LOP/M Volume Control */
#define AIC31_P0_REG88              (88) /* PGA_L to RIGHT_LOP/M Volume Control */
#define AIC31_P0_REG89              (89) /* DAC_L1 to RIGHT_LOP/M Volume Control */
#define AIC31_P0_REG90              (90) /* LINE2R to RIGHT_LOP/M Volume Control */
#define AIC31_P0_REG91              (91) /* PGA_R to RIGHT_LOP/M Volume Control */
#define AIC31_P0_REG92              (92) /* DAC_R1 to RIGHT_LOP/M Volume Control*/
#define AIC31_P0_REG93              (93) /* RIGHT_LOP/M Output Level Control */
#define AIC31_P0_REG94              (94) /* Module Power Status */
#define AIC31_P0_REG95              (95) /**< O/P Driver Short Circuit Detection Status*/
#define AIC31_P0_REG96              (96) /* Sticky Interrupt Flags */
#define AIC31_P0_REG97              (97) /* Real-time Interrupt Flags  */
#define AIC31_P0_REG98              (98) /* GPIO1 Control */
#define AIC31_P0_REG99              (99) /* GPIO2 Control */
#define AIC31_P0_REG100             (100)  /* Additional GPIO Control A */
#define AIC31_P0_REG101             (101)  /* Additional GPIO Control B */
#define AIC31_P0_REG102             (102)  /* Clock Generation Control */

#define AIC31_RESET                 (0x80)

#define AIC31_SLOT_WIDTH_16         (0u << 4u)
#define AIC31_SLOT_WIDTH_20         (1u << 4u)
#define AIC31_SLOT_WIDTH_24         (2u << 4u)
#define AIC31_SLOT_WIDTH_32         (3u << 4u)
```

These macros define registers addresses and values for the AIC3106 CODEC. You can find the description of these register in the AIC3106 datasheet which can be accessed at the link below:

https://www.ti.com/lit/ds/symlink/tlv320aic3106.pdf

Next will use a subset of these registers to configure the AIC3106 CODEC to the setting that our application requires.

- Add the following function prototypes for the AIC3106 CODEC configuration to the top of the `aic3106.h` file.

```
35 extern void AIC31Reset(unsigned int baseAddr);
36 extern void AIC31DataConfig(unsigned int baseAddr, unsigned char dataType,
37                              unsigned char slotWidth, unsigned char dataOff);
38 extern void AIC31SampleRateConfig(unsigned int baseAddr, unsigned int mode,
39                                   unsigned int sampleRate);
40 extern void AIC31ADCInit(unsigned int baseAddr);
41 extern void AIC31DACInit(unsigned int baseAddr);
```

For the AIC3106 CODEC configuration, we need to be able to reset the CODEC, configure the I2S communication settings, configure the ADC (for audio input) and DAC (for audio output) settings inside the AIC3106 CODEC, and finally se the sampling rate of the device.

We will now implement AIC3106 configuration functions.

- Add the definition for the AIC3106 reset function.

```
207 /**************************************************************
208 ** Resets the AIC31 Codec
209 **************************************************************/
210 void AIC31Reset(unsigned int baseAddr)
211 {
212     /* Select Page 0 */
213     CodecRegWrite(baseAddr, AIC31_P0_REG0, 0);
214
215     /* Reset the codec */
216     CodecRegWrite(baseAddr, AIC31_P0_REG1, AIC31_RESET);
217 }
```

- Add the definition for the AIC3106 I2S communication setting function.

```
220 /*****************************************************************************
221 ** Configures the data format and slot width
222 **
223 ** dataType     Data type for the codec operation
224 ** slotWidth    Slot width in bits
225 ** dataOff      The number of clocks from the word clock rising edge
226 **              to capture the actual data
227 **
228 *****************************************************************************/
229 void AIC31DataConfig(unsigned int baseAddr, unsigned char dataType,
230                      unsigned char slotWidth, unsigned char dataOff)
231 {
232     unsigned char slot;
233
234     switch(slotWidth)
235     {
236         case 16:
237             slot = AIC31_SLOT_WIDTH_16;
238         break;
239
240         case 20:
241             slot = AIC31_SLOT_WIDTH_20;
242         break;
243
244         case 24:
245             slot = AIC31_SLOT_WIDTH_24;
246         break;
247
248         case 32:
249             slot = AIC31_SLOT_WIDTH_32;
250         break;
251
252         default:
253             slot = AIC31_SLOT_WIDTH_16;
254         break;
255     }
256
257
258     /* Write the data type and  slot width */
259     CodecRegWrite(baseAddr, AIC31_P0_REG9, (dataType | slot));
260
261     /* valid data after dataOff number of clock cycles */
262     CodecRegWrite(baseAddr, AIC31_P0_REG10, dataOff);
263 }
```

The rest of the AIC3106 configuration functions are similar. Lets continue with adding their definitions.

- Add the definitions for `AIC31SampleRateConfig`, `AIC31ADCInit`, and `AIC31DACInit`.

```
/*****************************************************************************
** Configures the data format and slot width
**
** mode         section of the codec (ADC/DAC) for which the sample
** sampleRate   Sample rate in samples per second
**
*****************************************************************************
/
void AIC31SampleRateConfig(unsigned int baseAddr, unsigned int mode,
                           unsigned int sampleRate)
{
    unsigned char fs;
    unsigned char ref = 0x0Au;
    unsigned char temp;
    unsigned char pllPval = 4u;
    unsigned char pllRval = 1u;
    unsigned char pllJval = 16u;
    unsigned short pllDval = 0u;

    /* Select the configuration for the given sampling rate */
    switch(sampleRate)
    {
        case 8000:
            fs = 0xAAu;
        break;

        case 11025:
            fs = 0x66u;
            ref = 0x8Au;
            pllJval = 14u;
            pllDval = 7000u;
        break;

        case 16000:
            fs = 0x44u;
        break;

        case 22050:
            fs = 0x22u;
            ref = 0x8Au;
            pllJval = 14u;
            pllDval = 7000u;
        break;

        case 24000:
            fs = 0x22u;
        break;

        case 32000:
            fs = 0x11u;
        break;

        case 44100:
            ref = 0x8Au;
            fs = 0x00u;
            pllJval = 14u;
            pllDval = 7000u;
        break;

        case 48000:
            fs = 0x00u;
        break;
```

```
        case 96000:
            ref = 0x6Au;
            fs = 0x00u;
        break;

        default:
            fs = 0x00u;
        break;
    }

    temp = (mode & fs);

    /* Set the sample Rate */
    CodecRegWrite(baseAddr, AIC31_P0_REG2, temp);

    CodecRegWrite(baseAddr, AIC31_P0_REG3, 0x80 | pllPval);

    /* use PLL_CLK_IN as MCLK */
    CodecRegWrite(baseAddr, AIC31_P0_REG102, 0x08);

    /* Use PLL DIV OUT as codec CLK IN */
    CodecRegBitClr(baseAddr, AIC31_P0_REG101, 0x01);

    /* Select GPIO to output the divided PLL IN */
    CodecRegWrite(baseAddr, AIC31_P0_REG98, 0x20);

    temp = (pllJval << 2);
    CodecRegWrite(baseAddr, AIC31_P0_REG4, temp);

    /* Configure the PLL divide registers */
    CodecRegWrite(baseAddr, AIC31_P0_REG5, (pllDval >> 6) & 0xFF);
    CodecRegWrite(baseAddr, AIC31_P0_REG6, (pllDval & 0x3F) << 2);

    temp = pllRval;
    CodecRegWrite(baseAddr, AIC31_P0_REG11, temp);

    /* Enable the codec to be master for fs and bclk */
    CodecRegWrite(baseAddr, AIC31_P0_REG8, 0xD0);

    CodecRegWrite(baseAddr, AIC31_P0_REG7, ref);
}

/*****************************************************************************
** Initializes the ADC section of the AIC31 Codec
*****************************************************************************
/
void AIC31ADCInit(unsigned int baseAddr)
{
    /* enable the programmable PGA for left and right ADC  */
    CodecRegWrite(baseAddr, AIC31_P0_REG15, 0x00);
    CodecRegWrite(baseAddr, AIC31_P0_REG16, 0x00);

    /* Connect MIC3L is  to the left ADC PGA */
    CodecRegWrite(baseAddr, AIC31_P0_REG17, 0x00);

    /* Connect MIC3R is  to the right ADC PGA */
    CodecRegWrite(baseAddr, AIC31_P0_REG18, 0x00);

    /* Power MICBIAS output to 2.5V */
    CodecRegWrite(baseAddr, AIC31_P0_REG25, 0x80);

    /* power on the Line L1R */
    CodecRegWrite(baseAddr, AIC31_P0_REG19, 0x04);
```

```
    /* power on the Line LIL */
    CodecRegWrite(baseAddr, AIC31_P0_REG22, 0x04);
}

/*****************************************************************************
** Initializes the DAC section of the AIC31 Codec
*****************************************************************************
/
void AIC31DACInit(unsigned int baseAddr)
{
    /* power up the left and right DACs */
    CodecRegWrite(baseAddr, AIC31_P0_REG37, 0xE0);

    /* select the DAC L1 R1 Paths */
    CodecRegWrite(baseAddr, AIC31_P0_REG41, 0x02);
    CodecRegWrite(baseAddr, AIC31_P0_REG42, 0x6C);


    /* DAC L to HPLOUT Is connected */
    CodecRegWrite(baseAddr, AIC31_P0_REG47, 0x80);
    CodecRegWrite(baseAddr, AIC31_P0_REG51, 0x09);

    /* DAC R to HPROUT is connected */
    CodecRegWrite(baseAddr, AIC31_P0_REG64, 0x80);
    CodecRegWrite(baseAddr, AIC31_P0_REG65, 0x09);

    /* DACL1 connected to LINE1 LOUT */
    CodecRegWrite(baseAddr, AIC31_P0_REG82, 0x80);
    CodecRegWrite(baseAddr, AIC31_P0_REG86, 0x09);

    /* DACR1 connected to LINE1 ROUT */
    CodecRegWrite(baseAddr, AIC31_P0_REG92, 0x80);
    CodecRegWrite(baseAddr, AIC31_P0_REG93, 0x09);

    /* unmute the DAC */
    CodecRegWrite(baseAddr, AIC31_P0_REG43, 0x00);
    CodecRegWrite(baseAddr, AIC31_P0_REG44, 0x00);
}
```

Inspect the functions we just created and review the AIC3106 datasheet for the register definitions if you want to further investigate the purpose of the code.

The only thing left now is defining the AIC31I2SConfigure function.

- Implement the AIC3106 initialization function.

```
/*****************************************************************************
** Function to configure the codec for I2S mode
*****************************************************************************/
void AIC31I2SConfigure(void)
{
    volatile unsigned int delay = 0xFFF;

    AIC31Reset(CSL_I2C_0_DATA_CFG);
    while(delay--);

    /* Configure the data format and sampling rate */
    AIC31DataConfig(CSL_I2C_0_DATA_CFG, AIC31_DATATYPE_I2S, SLOT_SIZE, 0);
    AIC31SampleRateConfig(CSL_I2C_0_DATA_CFG, AIC31_MODE_BOTH, SAMPLING_RATE);

    /* Initialize both ADC and DAC */
    AIC31ADCInit(CSL_I2C_0_DATA_CFG);
    AIC31DACInit(CSL_I2C_0_DATA_CFG);
}
```

This function resets the AIC3106 CODEC, waits for the reset to take effect, configures the I2S communication settings, sets the device ADC and DAC samples rates, and finally initializes the ADC and the DAC modules.

We are now done with the `aic3106.c` and `aic3106.h` files. We can now call the `AIC31I2SConfigure` function in the `hardwareInitTask`.

12. Add the AIC3106 CODEC initialization function to the **`hardwareInitTask`**.

- Open the `hardwareInitTask.c` file and update the `hardwareInitTaskFxn` function to call the AIC3106 initialization function.

```
/*****************************************************************************
** hardwareInitTask.c
*****************************************************************************/

#include "myGlobalOptions.h"

#include "aic3106.h"

#include "hardwareInitTask.h"

/*****************************************************************************
** hardwareInitTask()
*****************************************************************************/
void hardwareInitTaskFxn (void)
{
    /* Initialize the I2C 0 interface for the codec AIC31 */
    I2CCodecIfInit(CSL_I2C_0_DATA_CFG, INT_CHANNEL_I2C, I2C_SLAVE_CODEC_AIC31);

    /* Configure the Codec for I2S mode */
    AIC31I2SConfigure();
}
```

The initializing of the AIC3106 CODEC device is completed.

Build, Load, Run…OBSERVE!

13. Build, load and run…

- Build your application and load your `.out` file into a new Debug session.
- If there are any build errors, fix them.
- Press “Resume” (Play) and make sure the AIC31I2SConfigure function executed.

If not, go resolve the problem. If it did, move on…

Adding McASP Support (lab_08c_loopbackAudio)

In this part of the lab we will add the code for sending/receiving data to/from the AIC3106 CODEC on TMDSLCDK6748 board, through I2S communication protocol supported by the McASP peripheral.

McASP Configuration Code

In order to configure the McASP peripheral to send/receive audio packets, we must examine the TMDSLCDK6748 board's schematic, so have the file open.

14. Create the header and source file which we will use for our McASP configuration code.

- Create a new file named `myMcasp.h`.
- Create a new file named `myMcsp.c`.

15. Add the McASP function prototypes to the `myMcsp.h` file.

- Open the `myMcasp.h` file and add the following function prototypes to the file.

myMcasp.c myMcasp.h

```
1  /**********************************
2  ** myMcasp.h
3  **********************************
4
5  #ifndef MYMCASP_H_
6  #define MYMCASP_H_
7
8  void McASPIsr(void);
9  void McASPI2SConfigure(void);
10 void I2SDataTxRxActivate(void);
11
12 #endif /* MYMCASP_H_ */
13
```

We will create three functions. The `McASPI2SConfigure` function will be used to setup the peripheral settings for I2S communication with the CODEC. The `I2SDataTxRxActivate` function will be used to enable the serializers used for I2S communication. The `McASPIsr` is the interrupt service routine handler for the McASP. We will add the necessary code to the ISR handler to service the receive and transmit buffers of the McASP peripheral.

16. Add the McASP function definitions to the myMcsp.c file.

- Open the `myMcasp.c` file.
- Add the necessary #includes shown below.

myMcasp.c myMcasp.h myMcasp.c

```
1 /*************************************************
2 ** myMcasp.c
3 *************************************************
4
5 #include <stdint.h>
6 #include <string.h>
7 #include <stdbool.h>
8 #include <ti/csl/hw_types.h>
9 #include <ti/csl/soc.h>
10 #include <ti/csl/arch/c67x/interrupt.h>
11 #include <ti/csl/csl_mcasp.h>
12
13 #include "myGlobalOptions.h"
14 #include "myMcasp.h"
15
```

Next, we have to add the code to configure the McASP I2S communication settings.

- Implement the `McASPI2SConfigure` function by adding the code below.

```
13 #include "myGlobalOptions.h"
14 #include "myMcasp.h"
15
16 /*************************************************************
17 ** Function Definitions
18 *************************************************************
19
20 /*
21 ** Configures the McASP Transmit Section in I2S mode.
22 */
23 void McASPI2SConfigure(void)
24 {
25
26
27 }
28
```

Inside the function we have to add our McASP initialization code.

The first thing needed is to put the McASP peripheral in RESET while we initialize the communication settings.

```
McASPRxReset(CSL_MCASP_0_CFG_REGS);

McASPTxReset(CSL_MCASP_0_CFG_REGS);
```

After placing the McASP transmitter and receiver in reset, we have to disable the McASP read and write FIFOs since for this section of the lab we will be using the CPU to service the McASP buffers. The read/write McASP FIFOs are only supported when the DMA is servicing the McASP buffers.

```
McASPReadFifoDisable(CSL_MCASP_0_FIFO_CFG_REGS);

McASPWriteFifoDisable(CSL_MCASP_0_FIFO_CFG_REGS);
```

The McASP I2S frame format must match the settings in the AIC3106 CODEC. The I2S frame **WORD SIZE** and **SLOT SIZE** are set to match the ones used to when we configured the AIC3106 CODEC in the previous section.

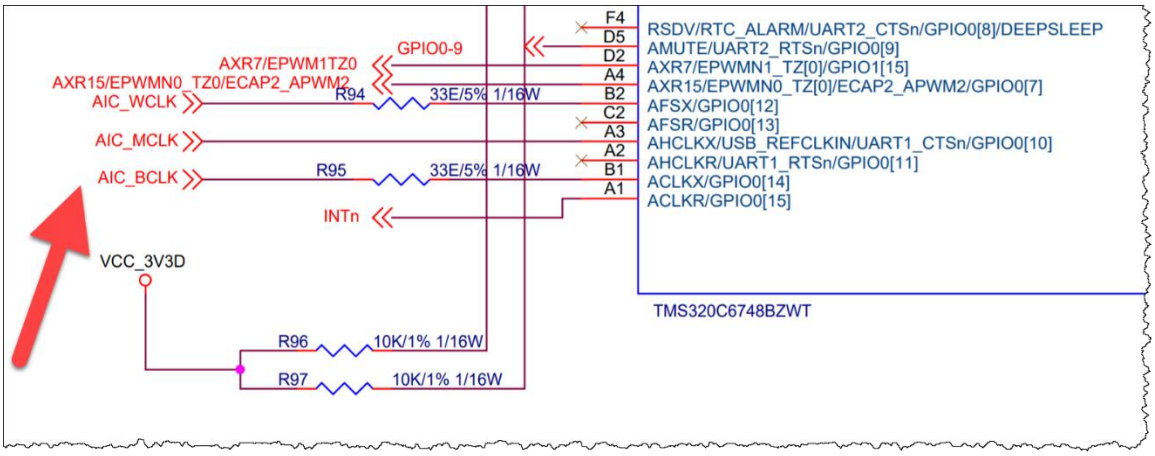
```
McASPRxFmtI2SSet(CSL_MCASP_0_CFG_REGS, WORD_SIZE, SLOT_SIZE,
                 MCASP_TX_MODE_NON_DMA);
McASPTxFmtI2SSet(CSL_MCASP_0_CFG_REGS, WORD_SIZE, SLOT_SIZE,
                 MCASP_TX_MODE_NON_DMA);
```

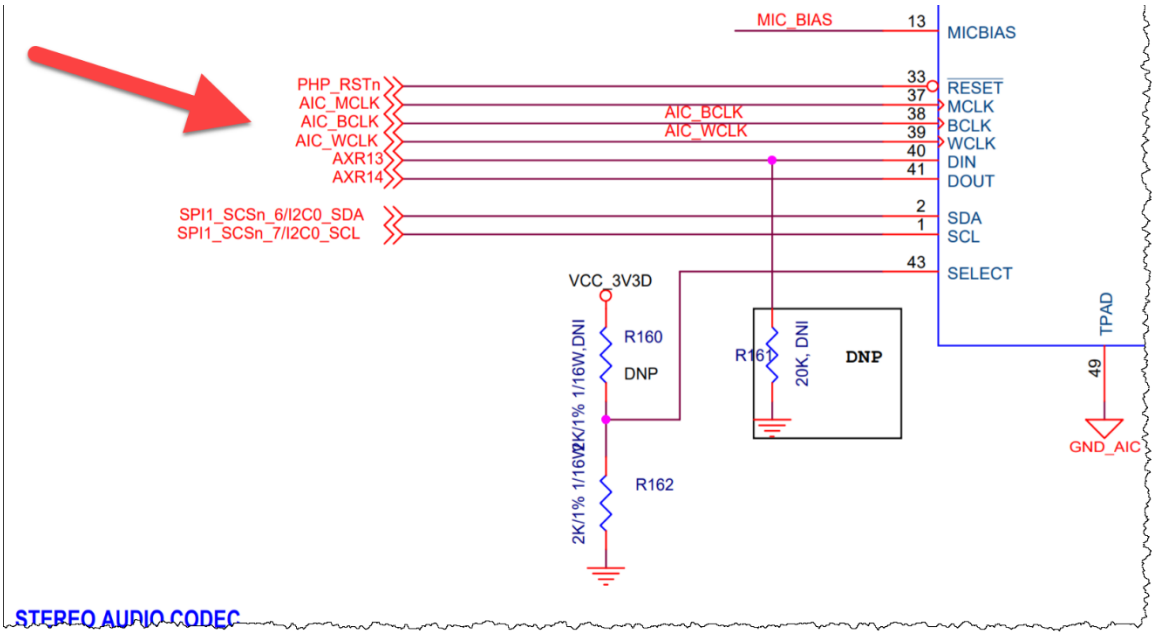
The last parameter in the functions above decide whether the CPU or the DMA is going to service the receive/transmit buffers of the McASP. For this section, we choose the CPU (NON_DMA). We will change in the future sections when we use the EDMA3 to service the McASP buffers.

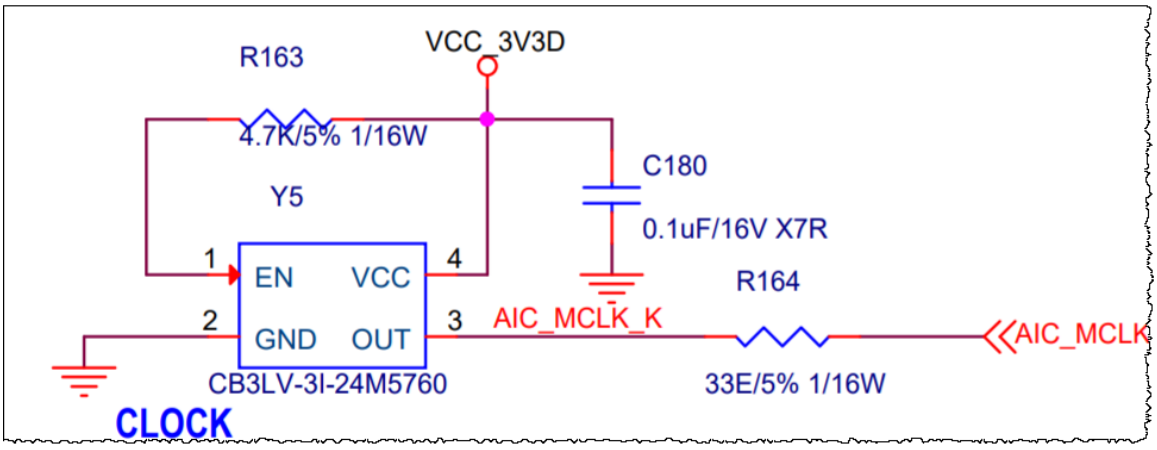
We set the McASP frame sync options to TDM mode with TWO SLOTs. The frame sync is set to external mode and the width of the frame sync signal is set to the width of a WORD. The receive frame sync will begin on the falling edge, while the transmit frame sync signal will begin on the rising edge.

```
McASPRxFrameSyncCfg(CSL_MCASP_0_CFG_REGS, 2,
                    MCASP_RX_FS_WIDTH_WORD,
                    MCASP_RX_FS_EXT_BEGIN_ON_FALL_EDGE);
McASPTxFrameSyncCfg(CSL_MCASP_0_CFG_REGS, 2,
                    MCASP_TX_FS_WIDTH_WORD,
                    MCASP_TX_FS_EXT_BEGIN_ON_RIS_EDGE);
```

We must also configure the McASP clock settings. Checking the board schematic, we see that the McASP clock is generated external and brought into the C6748 McASP peripheral through the **AHCLKX** pin. Look for **AIC_MCLK, AIC_WCLK** and **AIC_BCLK** in the schematic of the board.








```
McASPRxClkCfg(CSL_MCASP_0_CFG_REGS,
                  MCASP_RX_CLK_EXTERNAL, 0, 0);
McASPRxClkPolaritySet(CSL_MCASP_0_CFG_REGS,
                  MCASP_RX_CLK_POL_RIS_EDGE);
McASPRxClkCheckConfig(CSL_MCASP_0_CFG_REGS,
                  MCASP_RX_CLKCHCK_DIV32,
                  0x00, 0xFF);

McASPTxClkCfg(CSL_MCASP_0_CFG_REGS,
                  MCASP_TX_CLK_EXTERNAL, 0, 0);
McASPTxClkPolaritySet(CSL_MCASP_0_CFG_REGS,
                  MCASP_TX_CLK_POL_FALL_EDGE);
McASPTxClkCheckConfig(CSL_MCASP_0_CFG_REGS,
                  MCASP_TX_CLKCHCK_DIV32,
                  0x00, 0xFF);
```

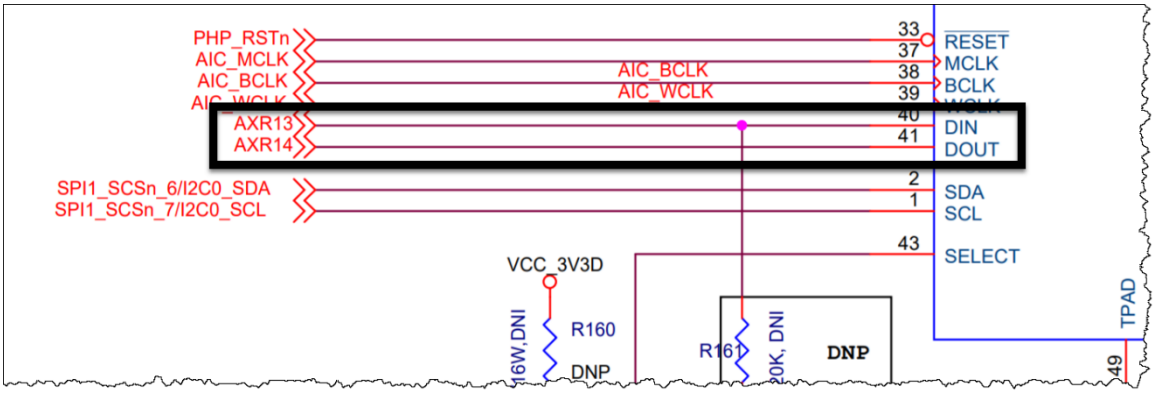
Then we must enable the frame sync for the McASP transmitter and receiver.

```
McASPTxRxClkSyncEnable(CSL_MCASP_0_CFG_REGS);
```

Next, we set the number of slots for both the McASP transmitter and receiver, which in our case is TWO.

```
McASPRxTimeSlotSet(CSL_MCASP_0_CFG_REGS, I2S_SLOTS);
McASPTxTimeSlotSet(CSL_MCASP_0_CFG_REGS, I2S_SLOTS);
```

Now it's time to find the McASP transmitter/receiver serializer numbers used. Open the schematic file again and you will see that the receiver serializer is #14 while the transmitter serializer is #13.




```
McASPSerializerRxSet(CSL_MCASP_0_CFG_REGS, MCASP_XSER_RX);

McASPSerializerTxSet(CSL_MCASP_0_CFG_REGS, MCASP_XSER_TX);
```

Each McASP pin must also be configured. The direction of the frame sync, clock and receiver serializer are set to INPUT. The direction of the transmitter serializer is set to OUTPUT.

```
McASPPinMcASPSet(CSL_MCASP_0_CFG_REGS, 0xFFFFFFFF);

McASPPinDirOutputSet(CSL_MCASP_0_CFG_REGS,
                     MCASP_PIN_AXR(MCASP_XSER_TX));

McASPPinDirInputSet(CSL_MCASP_0_CFG_REGS, MCASP_PIN_AFSX
                                | MCASP_PIN_ACLKX
                                | MCASP_PIN_AFSR
                                | MCASP_PIN_ACLKR
                                | MCASP_PIN_AXR(MCASP_XSER_RX));
```

Finally, we must enable the transmitter and receiver interrupts for the McASP peripheral. In this section of the lab we will enable all McASP EVENTs to generate interrupts, but we will only use the **DATAREADY** events for servicing the buffers. In the later labs we will only use the **ERROR** EVENTs to generate interrupts and use the McASP interrupt as a McASP **ERROR** interrupt.

```
McASPTxIntEnable(CSL_MCASP_0_CFG_REGS, MCASP_TX_DMAERROR
                                     | MCASP_TX_CLKFAIL
                                     | MCASP_TX_SYNCERROR
                                     | MCASP_TX_UNDERRUN
                                     | MCASP_TX_DATAREADY);

McASPRxIntEnable(CSL_MCASP_0_CFG_REGS, MCASP_RX_DMAERROR
                                     | MCASP_RX_CLKFAIL
                                     | MCASP_RX_SYNCERROR
                                     | MCASP_RX_OVERRUN
                                     | MCASP_RX_DATAREADY);
```

This concludes the McASP peripheral setting initialization..

- Ensure the `McASPI2SConfigure` function looks the same as the code below.

```
void McASPI2SConfigure(void)
{
    McASPRxReset(CSL_MCASP_0_CFG_REGS);
    McASPTxReset(CSL_MCASP_0_CFG_REGS);

    /* Disable the FIFOs for Non-DMA transfer */
    McASPReadFifoDisable(CSL_MCASP_0_FIFO_CFG_REGS);
    McASPWriteFifoDisable(CSL_MCASP_0_FIFO_CFG_REGS);

    /* Set I2S format in the transmitter/receiver format units */
    McASPRxFmtI2SSet(CSL_MCASP_0_CFG_REGS, WORD_SIZE, SLOT_SIZE,
                     MCASP_TX_MODE_NON_DMA);
    McASPTxFmtI2SSet(CSL_MCASP_0_CFG_REGS, WORD_SIZE, SLOT_SIZE,
                     MCASP_TX_MODE_NON_DMA);

    /* Configure the frame sync. I2S shall work in TDM format with 2 slots */
    McASPRxFrameSyncCfg(CSL_MCASP_0_CFG_REGS, 2, MCASP_RX_FS_WIDTH_WORD,
                        MCASP_RX_FS_EXT_BEGIN_ON_FALL_EDGE);
    McASPTxFrameSyncCfg(CSL_MCASP_0_CFG_REGS, 2, MCASP_TX_FS_WIDTH_WORD,
                        MCASP_TX_FS_EXT_BEGIN_ON_RIS_EDGE);

    /* configure the clock for receiver */
    McASPRxClkCfg(CSL_MCASP_0_CFG_REGS, MCASP_RX_CLK_EXTERNAL, 0, 0);
    McASPRxClkPolaritySet(CSL_MCASP_0_CFG_REGS, MCASP_RX_CLK_POL_RIS_EDGE);
    McASPRxClkCheckConfig(CSL_MCASP_0_CFG_REGS, MCASP_RX_CLKCHCK_DIV32,
                          0x00, 0xFF);

    /* configure the clock for transmitter */
    McASPTxClkCfg(CSL_MCASP_0_CFG_REGS, MCASP_TX_CLK_EXTERNAL, 0, 0);
    McASPTxClkPolaritySet(CSL_MCASP_0_CFG_REGS, MCASP_TX_CLK_POL_FALL_EDGE);
    McASPTxClkCheckConfig(CSL_MCASP_0_CFG_REGS, MCASP_TX_CLKCHCK_DIV32,
                          0x00, 0xFF);

    /* Enable synchronization of RX and TX sections  */
    McASPTxRxClkSyncEnable(CSL_MCASP_0_CFG_REGS);

    /* Enable the transmitter/receiver slots. I2S uses 2 slots */
    McASPRxTimeSlotSet(CSL_MCASP_0_CFG_REGS, I2S_SLOTS);
    McASPTxTimeSlotSet(CSL_MCASP_0_CFG_REGS, I2S_SLOTS);

    /*
    ** Set the serializers, Currently only one serializer is set as
    ** transmitter and one serializer as receiver.
    */
    McASPSerializerRxSet(CSL_MCASP_0_CFG_REGS, MCASP_XSER_RX);
    McASPSerializerTxSet(CSL_MCASP_0_CFG_REGS, MCASP_XSER_TX);

    /*
    ** Configure the McASP pins
    ** Input - Frame Sync, Clock and Serializer Rx
    ** Output - Serializer Tx is connected to the input of the codec
    */
    McASPPinMcASPSet(CSL_MCASP_0_CFG_REGS, 0xFFFFFFFF);
    McASPPinDirOutputSet(CSL_MCASP_0_CFG_REGS, MCASP_PIN_AXR(MCASP_XSER_TX));
    McASPPinDirInputSet(CSL_MCASP_0_CFG_REGS, MCASP_PIN_AFSX
                                            | MCASP_PIN_ACLKX
                                            | MCASP_PIN_AFSR
                                            | MCASP_PIN_ACLKR
                                            | MCASP_PIN_AXR(MCASP_XSER_RX));

    /* Enable error interrupts for McASP */
    McASPTxIntEnable(CSL_MCASP_0_CFG_REGS, MCASP_TX_DMAERROR
                                         | MCASP_TX_CLKFAIL
```

```
                                                | MCASP_TX_SYNCERROR
                                                | MCASP_TX_UNDERRUN
                                                | MCASP_TX_DATAREADY);

    McASPRxIntEnable(CSL_MCASP_0_CFG_REGS, MCASP_RX_DMAERROR
                                                | MCASP_RX_CLKFAIL
                                                | MCASP_RX_SYNCERROR
                                                | MCASP_RX_OVERRUN
                                                | MCASP_RX_DATAREADY);
```

- Implement the `I2SDataTxRxActivate` function by adding the code below.

```
void I2SDataTxRxActivate(void)
{
    /* Start the clocks */
    McASPRxClkStart(CSL_MCASP_0_CFG_REGS, MCASP_RX_CLK_EXTERNAL);
    McASPTxClkStart(CSL_MCASP_0_CFG_REGS, MCASP_TX_CLK_EXTERNAL);

    /* Activate the  serializers */
    McASPRxSerActivate(CSL_MCASP_0_CFG_REGS);
    McASPTxSerActivate(CSL_MCASP_0_CFG_REGS);

    /* make sure that the XDATA bit is cleared to zero */
    while(McASPTxStatusGet(CSL_MCASP_0_CFG_REGS) & MCASP_TX_STAT_DATAREADY);

    /* Activate the state machines */
    McASPRxEnable(CSL_MCASP_0_CFG_REGS);
    McASPTxEnable(CSL_MCASP_0_CFG_REGS);
}
```

This function will start the serializer clocks, and activate the McASP receiver and transmitter. Before we continue to enable the McASP receiver and transmitter, we put a while loop which will ensure that the transmitter data buffers are serviced before the McASP receiver/transmitter are enabled, avoiding an UNDERFLOW scenario.

The only function left to implement is the McASP ISR handler function.

- Add a global variable named `val` to the top of the `myMcasp.c` file.

```
13 #include "myGlobalOptions.h"
14 #include "myMcasp.h"
15
16
17 volatile uint32_t val = 0;
```

This variable will be used to pass the received value from the receive serializer buffer to the transmit serializer buffer.

- Implement the `McASPIsr` function by adding the code below.

```
void McASPIsr(void)
{
    while (McASPRxStatusGet(CSL_MCASP_0_CFG_REGS) & MCASP_RX_STAT_DATAREADY){
        val = McASPRxBufRead(CSL_MCASP_0_CFG_REGS, MCASP_XSER_RX);
    }
    while (McASPTxStatusGet(CSL_MCASP_0_CFG_REGS) & MCASP_TX_STAT_DATAREADY){
        McASPTxBufWrite(CSL_MCASP_0_CFG_REGS, MCASP_XSER_TX, val);
    }

    IntEventClear(CSL_INTC_EVENTID_MCASP0INT);
}
```

We will service the McASP receive buffer by reading the buffer using the CPU and writing to the `val` variable. Then when the transmitter is requesting new data, we will write `val` to the McASP transmit buffer. Finally the interrupt event is cleared.

This concludes the changes that must be made the `myMcasp.c` file. Now we have to call these newly added functions in the `hardwareInitTaskFxn` function.

17. Add the McASP initalization functions to the `hardwareInitTask.c` file.

- Open the `hardwareInitTask.c` file and add the `myMcsap.h #include` to the file.
- Update the `hardwareInitTaskFxn` function to call the McASP initialization functions.

```
 9 #include "hardwareInitTask.h"
10
11 #include "myMcasp.h"
12
13 /*************************************************************************
14 ** hardwareInitTask()
15 *************************************************************************/
16 void hardwareInitTaskFxn (void)
17 {
18     /* Initialize the I2C 0 interface for the codec AIC31 */
19     I2CCodecIfInit(CSL_I2C_0_DATA_CFG, INT_CHANNEL_I2C, I2C_SLAVE_CODEC_AIC31);
20
21     /* Configure the Codec for I2S mode */
22     AIC31I2SConfigure();
23
24     /* Configure the McASP for I2S */
25     McASPI2SConfigure();
26
27     /* Activate the audio transmission and reception */
28     I2SDataTxRxActivate();
29 }
```

The only thing left at this point is to add the HWI support for the McASP ISR handler. So let's do it!

18. Add the McASP ISR HWI to the `main.c` file.

- Open the `main.c` file and add the `myMcsap.h #include` to the file.
- Add the ***Hwi*** handle and params global variables.

```
35 #include "hardwareInitTask.h"
36 #include "aic3106.h"
37 #include "myMcasp.h"
38
39 /*****************************************************
40 ** Global Variables
41 *****************************************************/
42 Task_Handle         hardwareInitTask;
43 Task_Params         hardwareInitTaskParams;
44
45 Hwi_Handle          hwiMCASP;
46 Hwi_Params          hwiMCASPParams;
47
48 Hwi_Handle          hwiI2CCODEC;
49 Hwi_Params          hwiI2CCODECParams;
50
51 Error_Block         hwiEb;
```

- In the `main()` function, add the ***Hwi*** initialization code.

hardwareInitTask.c | hardwareInitTask.c | main.c | main.c

```
52
53 /*************************************************************************
54 ** main()
55 *************************************************************************/
56 int main(void)
57 {
58
59     // Initialize Peripherals
60     Board_init(BOARD_INIT_PINMUX_CONFIG |
61                BOARD_INIT_MODULE_CLOCK);
62     myGpio_init();
63
64     Error_init(&hwiEb);
65     Hwi_Params_init(&hwiMCASPParams);
66     hwiMCASPParams.maskSetting = Hwi_MaskingOption_NONE;
67     hwiMCASPParams.eventId = CSL_INTC_EVENTID_MCASP0INT;
68     hwiMCASP = Hwi_create(
69                             C674X_MASK_INT4,
70                             (Hwi_FuncPtr)McASPIsr,
71                             &hwiMCASPParams,
72                             &hwiEb
73     );
74     if (hwiMCASP == NULL) {
75         System_abort("MCASP HWI create failed");
76     }
77
78     Error_init(&hwiEb);
79     Hwi_Params_init(&hwiI2CCODECParams);
80     hwiI2CCODECParams.eventId = CSL_INTC_EVENTID_I2CINT0;
81     hwiI2CCODEC = Hwi_create(
82                             C674X_MASK_INT12,
83                             (Hwi_FuncPtr)I2CCodecIsr,
84                             &hwiI2CCODECParams,
85                             &hwiEb
86     );
87     if (hwiI2CCODEC == NULL) {
88         System_abort("I2C HWI create failed");
89     }
```

The McASP ***Hwi*** is set to use `INT4` with `McASPIsr` set as the handler. The `eventid` is set to the **McASP EVENT ID**.

That is it, now the McASP module is setup for audio LOOPBACK mode. You should now be able to build and run the code and hear the Audio Input as the Audio Output.

Build, Load, Run…OBSERVE!

19. Build, load and run…

- Build your application and load your `.out` file into a new Debug session.
- If there are any build errors, fix them.
- Press “Resume” (Play).

- Connect your phone (or any other device with an auxiliary audio output) to the Audio Input Jack connection on the board.
- Connect your headphones (or any device with an auxiliary audio input such as speakers) to the Audio Output Jack connection on the board,
- Play the music on your phone.

Do you hear the music in your headphones? If not, go resolve the problem. If you did, move on…

Audio Buffer Overflow/Underflow (lab_08d_overflowErrorAudio)

In this part of the lab we will add the code to toggle an LED. The McASP will still be serviced by the CPU. Adding a new ***Task*** to toggle an LED will actually cause a McASP buffer underflow/overflow error because the CPU will not be able to service the buffers in time.

LED (GPIO) Configuration Code

The purpose of this lab is not to teach a user how to the setup a GPIO peripheral. Therefore, we have existing GPIO configuration code in our project and will use this GPIO code to toggle an LED.

20. Add the LED toggle *Task* to the `main()` function.

- Open the `main.c` file.
- Add the global variable for the `ledTask`.

```
31 // Application Header Files
32 #include "myGlobalOptions.h"
33 #include "myGpio.h"
34
35 #include "hardwareInitTask.h"
36 #include "aic3106.h"
37 #include "myMcasp.h"
38
39 /*****************************************************
40 ** Global Variables
41 *****************************************************
42 Task_Handle         ledTask;
43 Task_Params         ledTaskParams;
44
45 Task_Handle         hardwareInitTask;
46 Task_Params         hardwareInitTaskParams;
47
48 Hwi_Handle          hwiMCASP;
49 Hwi_Params          hwiMCASPParams;
50
```

- Add the Task implementation code in the `main()` function above the `hardwareInitTask` implementation code.

```
95      // Create Task(s)
96      Task_Params_init(&ledTaskParams);
97      ledTaskParams.priority = 1;
98      ledTaskParams.stackSize = 2048;
99
00      ledTask = Task_create((Task_FuncPtr)ledToggleTaskFxn,
01                            &ledTaskParams, Error_IGNORE);
02      if (ledTask == NULL) {
03          System_abort("LED Task create failed");
04      }
05
```

The function which the task will use will be called `ledToggleTaskFxn` and we will implement it shortly. we will implement this function in a file set `called ledTask.c and ledTask.h`. Let's go ahead and #include the header file in the `main.c` file.

- Add the following `#define` to the top of the `main.c` file.

```
#include "ledTask.h"
```

21. Implement the LED toggle task function.

- Create two new files in your project named `ledTask.c and ledTask.h`.
- Open the `ledTask.h` file and update it to match the content below.

```
1  /*************************************
2  ** ledTask
3  *************************************
4
5  #ifndef LEDTASK_H_
6  #define LEDTASK_H_
7
8  // LED toggle Fxn
9  void ledToggleTaskFxn (void);
10
11 #endif /* LEDTASK_H_ */
12
```

- Open the ledTask.c file and update it to match the content below.

```
/*************************************
** ledTask.c
*************************************

#include <stdint.h>
#include <string.h>
#include <stdbool.h>

#include "myGlobalOptions.h"

#include <ti/sysbios/knl/Task.h>
#include <ti/drv/gpio/GPIO.h>
#include "GPIO_board.h"
#include "myGpio.h"


void ledToggleTaskFxn (void)
{
    while(1)
    {
        Task_sleep(5000);

        GPIO_toggle(USER_LED0);
    }

}
```

The task will toggle the GPIO connected to the board LED every 5s. The faster you request to blink the LED, the faster the audio application will fail.

Build, Load, Run...OBSERVE!

22. Build, load and run...

- Build your application and load your `.out` file into a new Debug session.
- If there are any build errors, fix them.
- Connect your phone (or any other device with an auxiliary audio output) to the Audio Input Jack connection on the board.
- Connect your headphones (or any device with an auxiliary audio input such as speakers) to the Audio Output Jack connection on the board,
- Play the music on your phone.
- Press “Resume” (Play).

Do you hear the music in your headphones? If not, go resolve the problem. If you did, move on...

- The music will eventually stop due to the McASP buffer error.

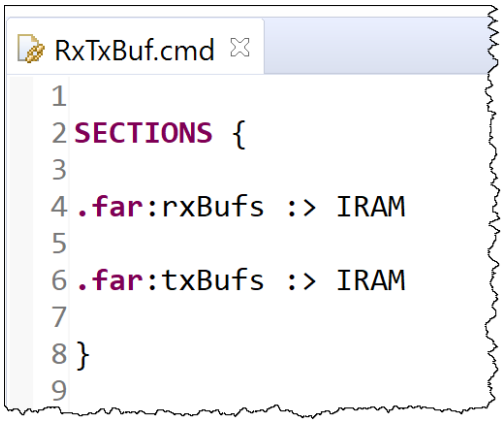
Adding EDMA Support (lab_08e_edmaAudio)

Examining the previous section, it is obvious that having the CPU service the McASP buffers is not feasible. Therefore, we need to ass EDMA3 support to service the McASP receive and transmit buffers.

We will have to use PING-PONG buffers for both McASP receiver and transmitter.

23. Create memory *data sections* for the audio buffers accessed by EDMA3.

- Create a new linker command file (CMD) named `RxTxBuf.cmd`.
- Add the following ***data sections*** to the CMD file:



24. Update McASP configuration code to enable EDMA access.

We no longer wish to use the CPU to service the buffers, therefore the McASP configuration code must change. Also, the McASP ISR will only be used for McASP ERROR events.

- Open the `myMcasp.h` file.
- Rename the `McASPIsr` to `McASPErrorIsr`.
- Open the `myMcasp.c` file.
- Delete the `uint32_t val` global variable.

  This variable is no longer needed since EDMA3 will have the audio data.
- Update the McASP ISR handler function to match the one below:

```
13 #include "myGlobalOptions.h"
14 #include "myMcasp.h"
15
16 /*************************************************
17 ** Function Definitions
18 *************************************************/
19
20 /*
21 ** Error ISR for McASP
22 */
23 void McASPErrorIsr(void)
24 {
25     IntEventClear(CSL_INTC_EVENTID_MCASP0INT);
26 }
```

- Enable the McASP read/write buffer FIFOs.

```
28 /*
29 ** Configures the McASP Transmit Section in I2S mode.
30 */
31 void McASPI2SConfigure(void)
32 {
33     McASPRxReset(CSL_MCASP_0_CFG_REGS);
34     McASPTxReset(CSL_MCASP_0_CFG_REGS);
35
36     /* Enable the FIFOs for DMA transfer */
37     McASPReadFifoEnable(CSL_MCASP_0_FIFO_CFG_REGS, 1, 1);
38     McASPWriteFifoEnable(CSL_MCASP_0_FIFO_CFG_REGS, 1, 1);
```

- Select DMA as the McASP buffers owner.

```
40     /* Set I2S format in the transmitter/receiver format units */
41     McASPRxFmtI2SSet(CSL_MCASP_0_CFG_REGS, WORD_SIZE, SLOT_SIZE,
42                      MCASP_RX_MODE_DMA);
43     McASPTxFmtI2SSet(CSL_MCASP_0_CFG_REGS, WORD_SIZE, SLOT_SIZE,
44                      MCASP_TX_MODE_DMA);
```

- Remove the `DATA_READY` event from the interrupt events.

```
91     /* Enable error interrupts for McASP */
92     McASPTxIntEnable(CSL_MCASP_0_CFG_REGS, MCASP_TX_DMAERROR
93                                          | MCASP_TX_CLKFAIL
94                                          | MCASP_TX_SYNCERROR
95                                          | MCASP_TX_UNDERRUN);
96
97     McASPRxIntEnable(CSL_MCASP_0_CFG_REGS, MCASP_RX_DMAERROR
98                                          | MCASP_RX_CLKFAIL
99                                          | MCASP_RX_SYNCERROR
100                                         | MCASP_RX_OVERRUN);
```

There is one final item we need to take care of in this file, and that is enabling the EDMA transfers in the I2SDataTxRxActivate before enabling the McASP transmit and receive serializers. We will take care of this later in the chapter.

25. Add the audio application code using EDMA3.

We will add two new files named `audioTask.c` and `audioTask.h` to the project. This will include the EDMA configuration code along with any audio application specific code. In this section, we will only copy our received audio buffer to the transmit audio buffer. In the later sections we will add audio filtering code to these files.

- Create a new file in the project named `audioTask.h`.
- Update the `audioTask.h` file to contain the code below:

myMcasp.c | audioTask.h

```
1  /*************************************
2  ** audioTask.h
3  *************************************/
4
5  #ifndef AUDIOTASK_H_
6  #define AUDIOTASK_H_
7
8  void CopyBufRxToTxTaskFxn (void);
9  void I2SDMAParamInit(void);
10 void EDMA3CCComplIsr(void);
11
12 #endif /* AUDIOTASK_H_ */
13
```

We will implement three outward facing functions in the `audioTask.h` file. A ***Task*** function named `CopyBufRxToTxTaskFxn`, an EDMA initialization function named `I2SDMAParamInit`, and an interrupt service routine handler named `EDMA3CCComplIsr` for when the EDMA3 transfers have completed.

- Create a new file in the project named `audioTask.c`.
- Add the necessary `#include` files:

myMcasp.c | audioTask.h | audioTask.c

```
/*****************************************
** audioTask.c
*****************************************/

// XDC
#include <xdc/std.h>
#include <xdc/runtime/Types.h>
#include <xdc/cfg/global.h>
#include <xdc/runtime/Types.h>
#include <xdc/runtime/Log.h>

// TI-RTOS Kernel Header Files
#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Semaphore.h>

// Standard C Header Files
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>

//CSL
#include <ti/csl/soc.h>
#include <ti/csl/hw_types.h>
#include <ti/csl/csl_mcasp.h>
#include <ti/csl/csl_edma.h>
#include <ti/csl/arch/c67x/interrupt.h>
#include <ti/csl/src/ip/edma/V1/edma.h>

#include "myGlobalOptions.h"


```

- Add the function prototypes for the new functions we will be writing.

```
/*****************************************************************************
** Function Prototypes
*****************************************************************************/
void ParamTxLoopJobSet(unsigned short parId);
void BufferTxDMAActivate(unsigned int txBuf, unsigned short numSamples,
                         unsigned short parToUpdate,
                         unsigned short linkAddr);
void BufferRxDMAActivate(unsigned int rxBuf, unsigned short parId,
                         unsigned short parLink);

```

We will write three new functions. `ParamTxLoopJobSet`, `BufferTxDMAActivate`, and `BufferTxDMAActivate`.

The first function `ParamTxLoopJobSet` is used to setup the audio transmit **EDMA PaRAM** for the given **PaRAM ID** and setting the **Link Address** of the **EDMA PaRAM** to itself, so it continues to loop back to use the same settings.

The second and third functions `BufferTxDMAActivate`, and `BufferTxDMAActivate` are used to reactivate the transmit and receive EDMA channels. The transmit EDMA channel is activated when a transmit is required (send data out to the **McASP**) while the receive EDMA channel is activated every time a reception is completed, and we need to start kickstart a new reception (from the **McASP**).

Next, we need to declare the global variables.

- Declare an extern global variable (***Semaphore***) for the receive EDMA interrupt.

```
41 /**************************************
42 ** Global Variables
43 **************************************
44 extern Semaphore_Handle    RxDmaReady;
```

  We will define and initialize this in `main.c` but we will use it in `audioTask.c`.

Now, let's define the arrays we need for receive and transmit buffers. We have to make sure to place them in the memory sections we created in the linker command file `RxTxBuf.cmd`.

- Create a variable named `loopBuf` which will be used by **EDMA** to transmit data to the **McASP** on the VERY FIRST transmission.

```
45
46 #pragma DATA_SECTION (loopBuf, ".far:txBufs");
47 #pragma DATA_ALIGN (loopBuf, 128);
48 static unsigned char loopBuf[NUM_SAMPLES_LOOP_BUF * BYTES_PER_SAMPLE] = {0};
49
```

  The `loopBuf` array will hold dummy data to kick start the audio communication.

- Create the ping-pong buffers for **EDMA** transmit and receive operations.

```
50 /*
51 ** Transmit buffers. If any new buffer is to be added, define it here and
52 ** update the NUM_BUF. These Tx buffers are accessed via EDMA3.
53 ** See RxTxBuf.cmd for placement of these user-defined sections.
54 */
55
56 #pragma DATA_SECTION (txBuf0, ".far:txBufs");
57 #pragma DATA_ALIGN (txBuf0, L2_LINESIZE);
58 static unsigned char txBuf0[AUDIO_BUF_SIZE];
59
60 #pragma DATA_SECTION (txBuf1, ".far:txBufs");
61 #pragma DATA_ALIGN (txBuf1, L2_LINESIZE);
62 static unsigned char txBuf1[AUDIO_BUF_SIZE];
63
64 #pragma DATA_SECTION (txBuf2, ".far:txBufs");
65 #pragma DATA_ALIGN (txBuf2, L2_LINESIZE);
66 static unsigned char txBuf2[AUDIO_BUF_SIZE];
67
68 /*
69 ** Receive buffers. If any new buffer is to be added, define it here and
70 ** update the NUM_BUF. These Rx buffers are accessed via EDMA3
71 */
72 #pragma DATA_SECTION (rxBuf0, ".far:rxBufs");
73 #pragma DATA_ALIGN (rxBuf0, L2_LINESIZE);
74 static unsigned char rxBuf0[AUDIO_BUF_SIZE];
75
76 #pragma DATA_SECTION (rxBuf1, ".far:rxBufs");
77 #pragma DATA_ALIGN (rxBuf1, L2_LINESIZE);
78 static unsigned char rxBuf1[AUDIO_BUF_SIZE];
79
80 #pragma DATA_SECTION (rxBuf2, ".far:rxBufs");
81 #pragma DATA_ALIGN (rxBuf2, L2_LINESIZE);
82 static unsigned char rxBuf2[AUDIO_BUF_SIZE];
```

Three buffers are created for the transmit **EDMA** channel and three buffers were created for the receive **EDMA** channel. The `loopBuf` buffer is only used for the first dummy transmit transfer. Starting from the second transfer, the `txBufs` are used.

- Create an array to be able to access the `txBuf0 – txBuf1` and `rxBuf0 – rxBuf1` through indexing.

```
87 /* Array of receive buffer pointers */
88 static unsigned int const rxBufPtr[NUM_BUF] =
89 {
90    (unsigned int) rxBuf0,
91    (unsigned int) rxBuf1,
92    (unsigned int) rxBuf2
93 };
94
95 /* Array of transmit buffer pointers */
96 static unsigned int const txBufPtr[NUM_BUF] =
97 {
98    (unsigned int) txBuf0,
99    (unsigned int) txBuf1,
100   (unsigned int) txBuf2
101 };
```

Now we can access the address of the buffers by indexing though the `txBufPtr` and `rxBufPtr`.

- Declare global variables to keep track of the state of the application.

```
84 // Next buffer to receive data. The data will be received in this buffer.
85 static volatile unsigned int nxtBufToRcv = 0;
86 // The RX buffer which filled latest.
87 static volatile unsigned int lastFullRxBuf = 0;
88 // The offset of the paRAM ID, from the starting of the paRAM set.
89 static volatile unsigned short parOffRcvd = 0;
90 // The offset of the paRAM ID sent, from starting of the paRAM set.
91 static volatile unsigned short parOffSent = 0;
92 // The offset of the paRAM ID to be sent next, from starting of the paRAM set.
93 static volatile unsigned short parOffTxToSend = 0;
94 // The transmit buffer which was sent last.
95 static volatile unsigned int lastSentTxBuf = NUM_BUF - 1;
```

Variable	Description
nxtBufToRcv	The index for the next receive buffer to be used in `rxBufPtr` to be used by the **EDMA**.
lastFullRxBuf	The index of the last receive buffer which was filled by the **EDMA**.
parOffRcvd	The **PaRAM** ID for the last received buffer's **EDMA** setting.
parOffSent	The **PaRAM** ID for the last send buffer's **EDMA** setting.
parOffTxToSend	The **PaRAM** ID for the next transmit buffer to send **EDMA** setting.
lastSentTxBuf	The index for the last transit buffer with was send in the `txBufPtr`.

We will use these variables in the application's state machine to send and receive from the correct buffer and **PaRAM** setting.

- Create "template" **EDMA3 PaRAM Entry** objects to use as the default setting for the EDMA transfers.

```
/*
** Default paRAM for Transmit section. This will be transmitting from
** a loop buffer.
*/
static EDMA3CCPaRAMEntry const txDefaultPar =
{
   (unsigned int)(EDMA3CC_OPT_DAM  | (0x02 << 8u)), /* Opt field */
   (unsigned int)loopBuf, /* source address */
   (unsigned short)(BYTES_PER_SAMPLE), /* aCnt */
   (unsigned short)(NUM_SAMPLES_LOOP_BUF), /* bCnt */
   (unsigned int) CSL_MCASP_0_SLV_REGS, /* dest address */
   (short) (BYTES_PER_SAMPLE), /* source bIdx */
   (short)(0), /* dest bIdx */
   (unsigned short)(PAR_TX_START * SIZE_PARAMSET), /* link address */
   (unsigned short)(0), /* bCnt reload value */
   (short)(0), /* source cIdx */
   (short)(0), /* dest cIdx */
   (unsigned short)1 /* cCnt */
};

/*
** Default paRAM for Receive section.
*/
static EDMA3CCPaRAMEntry const rxDefaultPar =
{
   (unsigned int)(EDMA3CC_OPT_SAM  | (0x02 << 8u)), /* Opt field */
   (unsigned int)CSL_MCASP_0_SLV_REGS, /* source address */
   (unsigned short)(BYTES_PER_SAMPLE), /* aCnt */
   (unsigned short)(1), /* bCnt */
   (unsigned int)rxBuf0, /* dest address */
   (short) (0), /* source bIdx */
   (short)(BYTES_PER_SAMPLE), /* dest bIdx */
   (unsigned short)(PAR_RX_START * SIZE_PARAMSET), /* link address */
   (unsigned short)(0), /* bCnt reload value */
   (short)(0), /* source cIdx */
   (short)(0), /* dest cIdx */
   (unsigned short)1 /* cCnt */
};
```

We created a default **PaRAM Entry** for each transmit and receive channels. In our application code we will always begin with copying these settings over and the modifying only the fields that need to be updated.

The `EDMA3CCPaRAMEntry` structure is defined as follows for the template transmit channel. The first item defines the option fields. You can review the available settings for the OPTION field by reviewing the C6748 TRM.

16.4.1.1 Channel Options Parameter (OPT)

The channel options parameter (OPT) is shown in Figure 16-35 and described in Table 16-15.

NOTE: The TCC field in OPT is a 6-bit field and can be programmed for any value between 0-64. For devices with 32 DMA channels, the TCC field should have a value between 0 to 31 so that it sets the appropriate bits (0 to 31) in the interrupt pending register (IPR) (and can interrupt the CPU(s) on enabling the interrupt enable register (IER) bits (0-31)).

Figure 16-35. Channel Options Parameter (OPT)

31–28	27–24	23	22	21	20	19–18	17–16
Reserved	PRIVID	ITCCHEN	TCCHEN	ITCINTEN	TCINTEN	Reserved	TCC
R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0 R-0	R/W-0

15–12	11–10	9–8	7–4	3	2	1	0
TCC	TCCMOD	FWID	Reserved	STATIC	SYNCDIM	DAM	SAM
R/W-0	R/W-0	R/W-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Table 16-15. Channel Options Parameters (OPT) Field Descriptions

Bit	Field	Value	Description
31-28	Reserved	0	Reserved
27-24	PRIVID	0-Fh	Privilege identification for the external host/CPU/DMA that programmed this PaRAM set. This value is set with the EDMA3 master's privilege identification value when any part of the PaRAM set is written.
23	ITCCHEN		Intermediate transfer completion chaining enable.
		0	Intermediate transfer complete chaining is disabled.
		1	Intermediate transfer complete chaining is enabled.
			When enabled, the chained event register (CER) bit is set on every intermediate chained transfer.

In the options field, we have enabled the **CONSTANT** address mode for the destination address since the destination is a hardware **FIFO**. Finally, the **FIFO** width is set to 32-bit mode by writing a **0x2** to the **FWID** bit range of the options register.

The source address for the template **PaRAM** entry is set to the address for the `loopBuf` buffer. The **ACNT** is set to the number of `BYTES_PER_SAMPLE`. **BCNT** is set to the number of **SAMPLES**. **CCNT** is set to 1 to ensure the transfer occurs. The **BCNT** reload value is set to 0 since no reload is required (**CCNT** = 1).

The destination address for the EDMA transfer is set to the McASP FIFO register address.

The **SOURCE BIDX** is set to the same value as the **ACNT** to ensure that the source address is incremented the same as **ACNT**. The **DESTINATION BIDX** is set to 0 since the McASP FIFO register address does not change. The **SOURCE** and **DESTINATION CIDX** are set to 0.

Finally, the **LINK ADDRESS** is set to point to the next **PaRAM**. In the template, it actually is pointing to itself which is the `PAR_TX_START x SIZE OF EACH PaRAM`.

For the template receive **PaRAM Entry**, the source address is **CONSTANT** address mode with the **FIFO** width set to 32-bit mode.

The source address for the template receive **PaRAM** is set to the **McASP** buffer address. The destination is set to the `rxBuf0` buffer. The **ACNT**, **BIDX**, **CCNT**, **CIDX** are setup, while the **BCNT** is taken care of later when the template is being used. The **LINK ADDRESS** is setup to point to the same **PaRAM** set.

That's it for our global variables. Next, we move on to implement the new functions.

- Define the `ParamTxLoopJobSet` function as shown below.

```
159 /*****************************************************************
160 **                          FUNCTION DEFINITIONS
161 *****************************************************************/
162
163 /*
164 ** Assigns loop job for a parameter set
165 */
166 void ParamTxLoopJobSet(unsigned short parId)
167 {
168     EDMA3CCPaRAMEntry paramSet;
169
170     memcpy(&paramSet, &txDefaultPar, SIZE_PARAMSET - 2);
171
172     /* link the paRAM to itself */
173     paramSet.linkAddr = parId * SIZE_PARAMSET;
174
175     EDMA3SetPaRAM(CSL_EDMA30CC_0_REGS, parId, &paramSet);
176 }
```

This function copies the template transmit **PaRAM Entry**. The number of bytes copied is `SIZE_PARAMSET - 2` because the last 2 bytes of the **PaRAM Entry** is actually RESERVED.

The link address of the **PaRAM Entry** is to ITSELF.

- Define the `I2SDMAParamInit` function as shown below.

```
void I2SDMAParamInit(void)
{
    EDMA3CCPaRAMEntry paramSet;
    int idx;

    /* Initialize the 0th paRAM set for receive */
    memcpy(&paramSet, &rxDefaultPar, SIZE_PARAMSET - 2);

    EDMA3SetPaRAM(CSL_EDMA30CC_0_REGS, EDMA3_CHA_MCASP0_RX, &paramSet);

    /* further paramsets, enable interrupt */
    paramSet.opt |= RX_DMA_INT_ENABLE;

    for(idx = 0 ; idx < NUM_PAR; idx++)
    {
        paramSet.destAddr = rxBufPtr[idx];

        paramSet.linkAddr = (PAR_RX_START + ((idx + 1) % NUM_PAR))
                            * (SIZE_PARAMSET);

        paramSet.bCnt =  NUM_SAMPLES_PER_AUDIO_BUF;

        /*
        ** for the first linked paRAM set, start receiving the second
        ** sample only since the first sample is already received in
        ** rx buffer 0 itself.
        */
        if( 0 == idx)
        {
            paramSet.destAddr += BYTES_PER_SAMPLE;
            paramSet.bCnt -= BYTES_PER_SAMPLE;
        }

        EDMA3SetPaRAM(CSL_EDMA30CC_0_REGS, (PAR_RX_START + idx), &paramSet);
    }

    /* Initialize the required variables for reception */
    nxtBufToRcv = idx % NUM_BUF;
    lastFullRxBuf = NUM_BUF - 1;
    parOffRcvd = 0;

    /* Initialize the 1st paRAM set for transmit */
    memcpy(&paramSet, &txDefaultPar, SIZE_PARAMSET);

    EDMA3SetPaRAM(CSL_EDMA30CC_0_REGS, EDMA3_CHA_MCASP0_TX, &paramSet);

    /* rest of the params, enable loop job */
    for(idx = 0 ; idx < NUM_PAR; idx++)
    {
        ParamTxLoopJobSet(PAR_TX_START + idx);
    }

    /* Initialize the variables for transmit */
    parOffSent = 0;
    lastSentTxBuf = NUM_BUF - 1;
}
```

This function will initialize all the EDMA3 parameters for **I2S** communication. We begin by copying the template receive **EDMA PaRAM Entry**. We copy `SIZE_PARAMSET - 2` because the last of the reserved bytes at the end. For the receive EDMA transfers, we enable the EDMA interrupt. For each of the two **PaRAM** sets (NUM_PAR = 2 for ping-pong buffers), we

set the destAddr and linkAddr of the PaRAM setting. We set the destAddr to the corresponding rxBuf, by indexing the rxBufPtr array. We set the linkAddr to the next PaRAM set for the ping-pong buffer. If you remember, we said that we will set the BCNT for the receive EDMA transfers later. Well now is the time! We set the BCNT to be the number of samples per audio buffer. Before, we initialize the PaRAM settings for the receive ping-pong buffers, it is important to take care of the unique scenario of the first reception. For the first reception, the first sample will already be in rxBuf0 so the EDMA PaRAM setting needs to start filling the buffer at rxBuf0 + BYTES_PER_SAMPLE address. Also the number of bytes for this transfer is 1 sample shorter.

Next, we can initialize some of the variables we use to keep track of the state of our audio application. First, we set the nxtBufToRcv to zero since we want to receive into rxBuf0.

We set the lastFullRxBuff to two indicating rxBuf2. This makes sense since lastFullRxBuff will always be the one before nxtBufToRcv. The parOffRcvd is set to zero. Think of rxBuf0, rxBuf1 and rxBuf2 as a circular buffer.

For the transmit EDMA transfers, we set up the ping-pong transfers using the ParamTxLoopJobSet function we previously wrote. Finally we initialize the state variables for the transmit transfers.

- Define the BufferTxDMAActivate function as shown below.

```
255 /*
256 ** Activates the DMA transfer for a parameterset from the given buffer.
257 */
258 void BufferTxDMAActivate(unsigned int txBuf, unsigned short numSamples,
259                          unsigned short parId, unsigned short linkPar)
260 {
261     EDMA3CCPaRAMEntry paramSet;
262
263     /* Copy the default paramset */
264     memcpy(&paramSet, &txDefaultPar, SIZE_PARAMSET - 2);
265
266     /* Enable completion interrupt */
267     paramSet.opt |= TX_DMA_INT_ENABLE;
268     paramSet.srcAddr =  txBufPtr[txBuf];
269     paramSet.linkAddr = linkPar * SIZE_PARAMSET;
270     paramSet.bCnt = numSamples;
271
272     EDMA3SetPaRAM(CSL_EDMA30CC_0_REGS, parId, &paramSet);
273 }
```

This function will be used to trigger a transfer from the txBuf0, txBuf1 and txBuf2 to the McASP through the EDMA. We start as we always do by copying the default template PaRAM Entry as our starting point. We then enable the transmit complete interrupt for the EDMA. We pick the txBuf from the txBufPtr array as the srcAddr, link the next PaRAM set, and update the BCNT.

- Define the BufferRxDMAActivate function as show below.

```
275 /*
276 ** Activates the DMA transfer for a parameter set from the given buffer.
277 */
278 void BufferRxDMAActivate(unsigned int rxBuf, unsigned short parId,
279                                  unsigned short parLink)
280 {
281     EDMA3CCPaRAMEntry paramSet;
282
283     /* Copy the default paramset */
284     memcpy(&paramSet, &rxDefaultPar, SIZE_PARAMSET - 2);
285
286     /* Enable completion interrupt */
287     paramSet.opt |= RX_DMA_INT_ENABLE;
288     paramSet.destAddr =  rxBufPtr[rxBuf];
289     paramSet.bCnt =  NUM_SAMPLES_PER_AUDIO_BUF;
290     paramSet.linkAddr = parLink * SIZE_PARAMSET ;
291
292     EDMA3SetPaRAM(CSL_EDMA30CC_0_REGS, parId, &paramSet);
293
294 }
```

This function is almost identical to its transmit counterpart. We enable the receive **EDMA** interrupt, set the destination address to the corresponding `rxBuf`, update the **BCNT**, and set the link **PaRAM** setting.

Next, we need to add the interrupt handler for the **EDMA3** interrupts named `EDMA3CCComplIsr`. We will also need to add the ***Hwi*** definition for the interrupt in `main.c`. We also need to be able to signal our audio task when the **EDMA3** interrupt occurs. This ***Semaphore*** must also be defined in `main.c`. We previously declared an extern reference to this semaphore in `audioTask.c`.

- Add the definition for `EDMA3CCComplIsr` to audioTask.c

```
299 void EDMA3CCComplIsr(void)
300 {
301     unsigned short nxtParToUpdate;
302
303     IntEventClear(CSL_INTC_EVENTID_EDMA3_0_CC0_INT1);
304
305     /* Check if receive DMA completed */
306     if(EDMA3GetIntrStatus(CSL_EDMA30CC_0_REGS) & (1 << EDMA3_CHA_MCASP0_RX))
307     {
308         /* Clear the interrupt status for the 0th channel */
309         EDMA3ClrIntr(CSL_EDMA30CC_0_REGS, EDMA3_CHA_MCASP0_RX);
310         /*
311         ** Update lastFullRxBuf to indicate a new buffer reception
312         ** is completed.
313         */
314         lastFullRxBuf = (lastFullRxBuf + 1) % NUM_BUF;
315         nxtParToUpdate =  PAR_RX_START + parOffRcvd;
316         parOffRcvd = (parOffRcvd + 1) % NUM_PAR;
317
318         /*
319         ** Update the DMA parameters for the received buffer to receive
320         ** further data in proper buffer
321         */
322         BufferRxDMAActivate(nxtBufToRcv, nxtParToUpdate,
323                             PAR_RX_START + parOffRcvd);
324
325         /* update the next buffer to receive data */
326         nxtBufToRcv = (nxtBufToRcv + 1) % NUM_BUF;
327
328
329         /* post Semaphore to unblock CopyBufRxToTxTaskFxn to switch EDMA channels, do FIR filter */
330         /* only the receive side posts - assumes Tx and Rx are operating at the same frequency */
331         Semaphore_post(RxDmaReady);
332     }
333
334     /* Check if transmit DMA completed */
335     if(EDMA3GetIntrStatus(CSL_EDMA30CC_0_REGS) & (1 << EDMA3_CHA_MCASP0_TX))
336     {
337         /* Clear the interrupt status for the first channel */
338         EDMA3ClrIntr(CSL_EDMA30CC_0_REGS, EDMA3_CHA_MCASP0_TX);
339         ParamTxLoopJobSet((unsigned short)(PAR_TX_START + parOffSent));
340         parOffSent = (parOffSent + 1) % NUM_PAR;
341     }
342 }
```

The EDMA transmit/receive interrupt handler will first clear the interrupt event. If the interrupt is generated due to a McASP receive event, we clear the EDMA McASP RX channel interrupt flag. We then update the state variables for our audio application. The `lastFullRxBuf` is incremented to point to the correct buffer currently full. The `nxtParToUpdate` is set to the currently received/completed EDMA parameter set. The received EDMA parameter set is also updated to point to the current parameter set. In order to activate the next EDMA McASP receive transfer, we call the `BufferRxDMAActivate` function. Last, we update the `nxtBufToRcv` and post the `RxDmaReady` semaphore to signal any task pending for an EDMA receive interrupt.

If the interrupt is generated due to a McASP transmit event, we clear the EDMA McASP TX channel interrupt flag. We then call the `ParamTxLoopJobSet` function and update the `parOffSent`.

That concludes what the EDMA3 ISR must accomplish.

Now, let's update the `main.c` file to define the ***Hwi*** and ***Semaphore*** required by the EDMA3 module.

Create the EDMA3 BIOS Modules

We need to create the ***Hwi*** and ***Semaphore*** required by the EDMA3 module.

26. Add the BIOS modules for EDMA3 module.

- Open the `main.c` file.
- Add the following #include to the include section.

```
#include "audioTask.h"
```

- Declare the Task handle and Task parameter objects for the `audioTask`.

```
41 /****************************************
42 ** Global Variables
43 ****************************************
44 Task_Handle          ledTask;
45 Task_Params          ledTaskParams;
46
47 Task_Handle          audioTask;
48 Task_Params          audioTaskParams;
49
```

The `audioTask` which we have not defined the function for yet, will do the task of transferring the received audio frames to the audio transmit buffers.

- Declare the ***Hwi*** handle and parameter for the EDMA interrupt.

```
56 Hwi_Handle          hwiEDMA3CC;
57 Hwi_Params          hwiEDMA3CCParams;
58
```

- Rename the McASP ***Hwi*** handle and parameter to show that is it an ERROR interrupt.

```
53 Hwi_Handle          hwiMCASPErr;
54 Hwi_Params          hwiMCASPErrParams;
```

```
76    Hwi_Params_init(&hwiMCASPErrParams);
77    hwiMCASPErrParams.maskSetting = Hwi_MaskingOption_NONE;
78    hwiMCASPErrParams.eventId = CSL_INTC_EVENTID_MCASP0INT;
79    hwiMCASPErr = Hwi_create(
80                            C674X_MASK_INT4,
81                            (Hwi_FuncPtr)McASPErrorIsr,
82                            &hwiMCASPErrParams,
83                            &hwiEb
84    );
85    if (hwiMCASPErr == NULL) {
86        System_abort("MCASP HWI create failed");
87    }
88
```

- Declare the EDMA3 ISR signaling ***Semaphore***.

```
64 Semaphore_Handle    RxDmaReady;
65
```

- Define the EDMA ***Hwi*** in the main() function.

```
91    Error_init(&hwiEb);
92    Hwi_Params_init(&hwiEDMA3CCParams);
93    hwiEDMA3CCParams.eventId = CSL_INTC_EVENTID_EDMA3_0_CC0_INT1;
94    hwiEDMA3CC = Hwi_create(
95                            C674X_MASK_INT11,
96                            (Hwi_FuncPtr)EDMA3CCComplIsr,
97                            &hwiEDMA3CCParams,
98                            &hwiEb
99    );
100   if (hwiEDMA3CC == NULL) {
101       System_abort("EDMA HWI create failed");
102   }
103
```

We set the function for the ***Hwi*** to be `EDMA3CCComplIsr`. The `eventId` is set to the **EDMA3 Channel Controller 0 Interrupt 1**.

- Define the EDMA completion interrupt signaling ***Semaphore*** in the main() function.

```
RxDmaReady = Semaphore_create(0, NULL, Error_IGNORE);
```

- Define the `audioTask` ***Task***.

```
144   Task_Params_init(&audioTaskParams);
145   audioTaskParams.priority = 3;
146   audioTaskParams.stackSize = 2048;
147
148   audioTask = Task_create((Task_FuncPtr)CopyBufRxToTxTaskFxn,
149                           &audioTaskParams, Error_IGNORE);
150   if (audioTask == NULL) {
151       System_abort("AUDIO Task create failed");
152   }
153
154
155   BIOS_start();
```

We set the ***Task*** priority to be 3 and the function which will be used for our audio task will be named `CopyRxToTxTaskFxn`.

All that is left now is implementing the `CopyRxToTxTaskFxn` function.

Create the Audio Task Function

Now we will implement the `CopyRxToTxTaskFxn` function. This task will transfer the received audio frames to the audio transmit buffers.

27. Add the `CopyRxToTxTaskFxn` to `audioTask.c`.

- Open the `audioTask.c` file.
- Implement the function as shown below.

```
277 void CopyBufRxToTxTaskFxn (void)
278 {
279     unsigned short parToSend;
280     unsigned short parToLink;
281     int status;
282
283     /*
284     ** Loop forever. if a new buffer is received, the lastFullRxBuf will be
285     ** updated in the rx completion ISR.
286     */
287     while(1)
288     {
289         status = Semaphore_pend (RxDmaReady, 500);
290
291         if (status==0)
292             Log_info0 ("DMA Receive interrupt never fired");
293
294
295         /*
296         ** Start the transmission from the link paramset. The param set
297         ** 1 will be linked to param set at PAR_TX_START. So do not
298         ** update paRAM set1.
299         */
300         parToSend =  PAR_TX_START + (parOffTxToSend % NUM_PAR);
301         parOffTxToSend = (parOffTxToSend + 1) % NUM_PAR;
302         parToLink  = PAR_TX_START + parOffTxToSend;
303
304         lastSentTxBuf = (lastSentTxBuf + 1) % NUM_BUF;
305
306         /* Copy the buffer */
307         memcpy((void *)txBufPtr[lastSentTxBuf],
308              (void *)rxBufPtr[lastFullRxBuf],
309              AUDIO_BUF_SIZE);
310
311         /*
312         ** Send the buffer by setting the DMA params accordingly.
313         ** Here the buffer to send and number of samples are passed as
314         ** parameters. This is important, if only transmit section
315         ** is to be used.
316         */
317         BufferTxDMAActivate(lastSentTxBuf, NUM_SAMPLES_PER_AUDIO_BUF,
318                             (unsigned short)parToSend,
319                             (unsigned short)parToLink);
320     }
321 }
```

This is our audio ***Task*** function. It will wait for an EDMA3 ISR to post the `RxDmaReady`. If at any point you see the `Log_info0` statement posted in the log results, a timeout has occurred while waiting for the ***Semaphore***.

When the EDMA3 receive operation is completed and the ***Semaphore*** pend returns successfully, we begin updating some state variable for our audio application. We prepare the buffers for sending the audio data by copy the `rxBuf` which was last filled, into the `txBuf` which is going to be sent. We execute the copy action by using the CPU to do a `memcpy`.

Finally, with the `txBuf` setup, we activate the **EDMA3** transfer for the audio data to be sent through the **McASP**.

One last thing we need to take care of is enabling the EDMA3 transfers for McASP RX and TX channels in EVENT mode.

28. Enable EDMA3 transfers for McASP RX and TX in event mode.

We need to enable the EDMA3 transfers for McASP RX and TX channels before activating the McASP TX and RX serializers.

- Open myMcasp.c file.
- Update the I2SDataTxRxActivate as show below.

```
109 void I2SDataTxRxActivate(void)
110 {
111     /* Start the clocks */
112     McASPRxClkStart(CSL_MCASP_0_CFG_REGS, MCASP_RX_CLK_EXTERNAL);
113     McASPTxClkStart(CSL_MCASP_0_CFG_REGS, MCASP_TX_CLK_EXTERNAL);
114
115     /* Enable EDMA for the transfer */
116     EDMA3EnableTransfer(CSL_EDMA30CC_0_REGS, EDMA3_CHA_MCASP0_RX,
117                         EDMA3_TRIG_MODE_EVENT);
118     EDMA3EnableTransfer(CSL_EDMA30CC_0_REGS,
119                         EDMA3_CHA_MCASP0_TX, EDMA3_TRIG_MODE_EVENT);
120
121     /* Activate the  serializers */
122     McASPRxSerActivate(CSL_MCASP_0_CFG_REGS);
123     McASPTxSerActivate(CSL_MCASP_0_CFG_REGS);
124
125     /* make sure that the XDATA bit is cleared to zero */
126     while(McASPTxStatusGet(CSL_MCASP_0_CFG_REGS) & MCASP_TX_STAT_DATAREADY);
127
128     /* Activate the state machines */
129     McASPRxEnable(CSL_MCASP_0_CFG_REGS);
130     McASPTxEnable(CSL_MCASP_0_CFG_REGS);
131 }
```

Last, we need to update our high priority `hardwareInitTaskFxn` to include the EDMA3 initialization code.

29. Add the EDMA3 initialization code to the hardwareInitTaskFxn.

- Open the `hardwareInitiTask.c` file.
- Add the `audioTask.h` as a `#include` to the `hardwareInitTask.c` file.

  `#include "audioTask.h"`

- Update the `hardwareInitTaskFxn` function to include the EDMA3 initialization code.

```
/*****************************************************************************
** hardwareInitTask()
*****************************************************************************/
void hardwareInitTaskFxn (void)
{
    /* Initialize the I2C 0 interface for the codec AIC31 */
    I2CCodecIfInit(CSL_I2C_0_DATA_CFG, INT_CHANNEL_I2C, I2C_SLAVE_CODEC_AIC31);

    EDMAsetRegion(1U);
    EDMA3Init(CSL_EDMA30CC_0_REGS, 0);

    /*
    ** Request EDMA channels. Channel 0 is used for reception and
    ** Channel 1 is used for transmission
    */
    EDMA3RequestChannel(CSL_EDMA30CC_0_REGS, EDMA3_CHANNEL_TYPE_DMA,
                        EDMA3_CHA_MCASP0_TX, EDMA3_CHA_MCASP0_TX, 0);
    EDMA3RequestChannel(CSL_EDMA30CC_0_REGS, EDMA3_CHANNEL_TYPE_DMA,
                        EDMA3_CHA_MCASP0_RX, EDMA3_CHA_MCASP0_RX, 0);

    /* Initialize the DMA parameters */
    I2SDMAParamInit();

    /* Configure the Codec for I2S mode */
    AIC31I2SConfigure();

    /* Configure the McASP for I2S */
    McASPI2SConfigure();

    /* Activate the audio transmission and reception */
    I2SDataTxRxActivate();
}
```

- Save and close all files.

That's it 😊 easy right? You should now have a fully functional EDMA3 based audio loopback application!

Build, Load, Run…OBSERVE!

30. Build, load and run…

- Build your application and load your `.out` file into a new Debug session.
- If there are any build errors, fix them.
- Connect your phone (or any other device with an auxiliary audio output) to the Audio Input Jack connection on the board.
- Connect your headphones (or any device with an auxiliary audio input such as speakers) to the Audio Output Jack connection on the board,
- Play the music on your phone.
- Press "Resume" (Play).

Do you hear the music in your headphones? If not, go resolve the problem. If you did, move on…

- This time, the music will NOT stop due to the McASP buffer being controlled by the EDMA and the system will not face an overflow/underflow issue. You can go ahead

and increase the blinking rate of the LED to see that the audio is not affected by the frequency of the led toggle.

Adding Audio Filters Support (lab_08f_filterAudio)

With the EDMA3 controlling our audio buffer transfers, we can use the CPU bandwidth to apply audio filter to our audio data. Next, we will add the code to filter the audio packets we receive, before sending them back to the McASP.

Audio Filter Code

We will add our audio filter code and audio filter coefficients into two new files named `coeffs.c` and `coeffs.h`.

- Create a new file named `coeffs.h`.
- Update the `coeffs.h` file to match the content below.

myMcasp.c | coeffs.h

```
/*****************************************************************************
** coeffs.h
*****************************************************************************/

#ifndef COEFFS_H_
#define COEFFS_H_

// Used for FIR coefficients located in coeffs.c
extern int16_t COEFFS[ORDER];

// prototype for new cfir algorithm
void cfir(int16_t * x, int16_t * h, int16_t * restrict r, uint16_t nh, int16_t nr);

#endif /* COEFFS_H_ */
```

The variable `COEFFS` is an array of filter coefficients for our audio filter. The `cfir` function is a finite impulse response filter application function.

- Create a new file named `coeffs.c`.
- Update the `coeffs.c` file to match the content below.

```
/*****************************************************************************
** coeffs.c
*****************************************************************************/

#include "myGlobalOptions.h"


int16_t COEFFS[ORDER] = {

        // Allpass Filter Coeff's - simple pass-thru effect...
        32767,  0,      0,0,    0,    0,    0,    0,    0,    0,
        0,      0,      0,0,    0,
                0,      0,0,    0,    0,    0,    0,    0,    0,
        0,      0,      0,0,    0,    0,
                0,      0,0,    0,    0,    0,    0,    0,    0,
        0,      0,      0,0,    0,    0,
                0,      0,0,    0,    0,    0,    0,    0,    0,
        0,      0,      0,0,    0,    0
};


/*****************************************************************************
** coeffs.c
** cfir() - FIR algo
**
** Parms:  x = delayBuffer (input Rcv buffer)
**         h = COEFFS
**         r = output Buffer (Xmt)
**         nh = ORDER of filter
**         nr = BUFFSIZE
**
**  Brief: This is 64-order xyz-pass (depends on COEFFS) FIR filter written in
C.
**
**  Note:  Can use pragma to place this code in a user_defined section. This
**         requires the addition of a user linker.cmd file. Can also incoporate
**         other optimizations as the lab instructions describe.
*****************************************************************************/

void cfir(int16_t * x, int16_t * h, int16_t * r, uint16_t nh, int16_t nr)
{
    int16_t i, j;
    int32_t sum;

    for (j = 0; j < nr; j++)
    {
        sum = 0;

        for (i = 0; i < nh; i++)
        sum += (int32_t)x[i + j] * h[i];

        r[j] = (int16_t)(sum >> 15);
    }
}

// ************************ END OF FILE *********************************
```

The variable `COEFFS` is defined in this file as the filter coefficients for our audio filter. The coefficients are for an ALL PASS filter. Meaning all frequencies are passed through and nothing is actually filtered out.

If you would like to try some other filters, try the Low Pass Filter:

```
int16_t COEFFS[ORDER] =
{

        // Lowpass Filter Coeff's stopband= 1500 Hz, passband= 500 Hz
     -782,   -100,   -100,    -95,    -84,    -68,    -46,    -17,     18,
       59,    107,    162,    222,    288,    358,    433,    511,    592,
      675,    756,    839,    919,    996,   1070,   1138,   1200,   1256,
     1303,   1342,   1372,   1392,   1402,   1402,   1392,   1372,   1342,
     1303,   1256,   1200,   1138,   1070,    996,    919,    839,    756,
      675,    592,    511,    433,    358,    288,    222,    162,    107,
       59,     18,    -17,    -46,    -68,    -84,    -95,   -100,   -100,
     -782
};
```

And for High Pass Filter try:

```
int16_t COEFFS[ORDER] = {

        // Highpass Filter Coeff's stopband= 500 Hz, passband= 1500 Hz
     -296,    377,    271,    235,    233,    245,    265,    286,    305,
      322,    335,    341,    341,    333,    315,    287,    247,    194,
      126,     41,    -63,   -188,   -341,   -525,   -752,  -1036,  -1404,
    -1907,  -2656,  -3938,  -6812, -20814,  20814,   6812,   3938,   2656,
     1907,   1404,   1036,    752,    525,    341,    188,     63,    -41,
     -126,   -194,   -247,   -287,   -315,   -333,   -341,   -341,   -335,
     -322,   -305,   -286,   -265,   -245,   -233,   -235,   -271,   -377,
      296
};
```

Now that we have the filter code in place, let's actually update our audioTask function to use the audio filter.

31. Update the `audioTask.c` file to include the audio filter code.

- Open `audioTask.c` file.
- Add the `#include` for `coeff.h`.

```
#include "coeffs.h"
```

We need to create a new structure to keep track of the previous audio packet to be able to do the audio filtering. We keep a history of the audio samples from the previously received audio packet with the size of `HIST_SIZE`.

- Create a structure to keep track of each audio channel (**Left/Right**) along with the history of the same channel.

```
33 // Structure for internal FIR Rx buffer (L and R)
34 typedef struct rcv_data_buffer
35 {
36     int16_t hist[HIST_SIZE];
37     int16_t data[AUDIO_BUF_SIZE/2];
38 } RCV_DATA_BUFFER;
```

To get the **Left/Right** channel data from the audio packets, we need to DEINTELEAVE the received data. We also need buffers to store the DEINTERLEAVED data so we can apply our audio filter to them.

If the complete INTERLEAVED audio data received is `AUDIO_BUF_SIZE` long, each **Left/Right** channel will be `AUDIO_BUF_SIZE/2` long.

- Create the left and right audio buffers.

```
104
105 #pragma DATA_SECTION (txBufFirL, ".far:txBufs");
106 #pragma DATA_ALIGN (txBufFirL, L2_LINESIZE);
107 int16_t txBufFirL[AUDIO_BUF_SIZE/2];
108
109 #pragma DATA_SECTION (txBufFirR, ".far:txBufs");
110 #pragma DATA_ALIGN (txBufFirR, L2_LINESIZE);
111 int16_t txBufFirR[AUDIO_BUF_SIZE/2];
112
113 /*
114 ** Create a struct with HISTORY and DATA buffer together for FIR filter -
115 ** only receive side is needed. Then use that struct to define the two
116 ** Fir filter (internal) buffers - NOT connected to the EDMA.
117 */
118
119 #pragma DATA_SECTION (rxBufFirL, ".far:rxBufs");
120 #pragma DATA_ALIGN (rxBufFirL, L2_LINESIZE);
121 RCV_DATA_BUFFER rxBufFirL;
122
123 #pragma DATA_SECTION (rxBufFirR, ".far:rxBufs");
124 #pragma DATA_ALIGN (rxBufFirR, L2_LINESIZE);
125 RCV_DATA_BUFFER rxBufFirR;
```

history

Explore Results

We can view our ***Hwi***s in the Runtime Object View tool to see the current status.

Clean up After Yourself

32. That's it, You're Done!

- Save and close all open files.
- Close CCS and power cycle your board…

Congratulations – you have now created and built an application using audio.

Congrats, you are done with this lab.