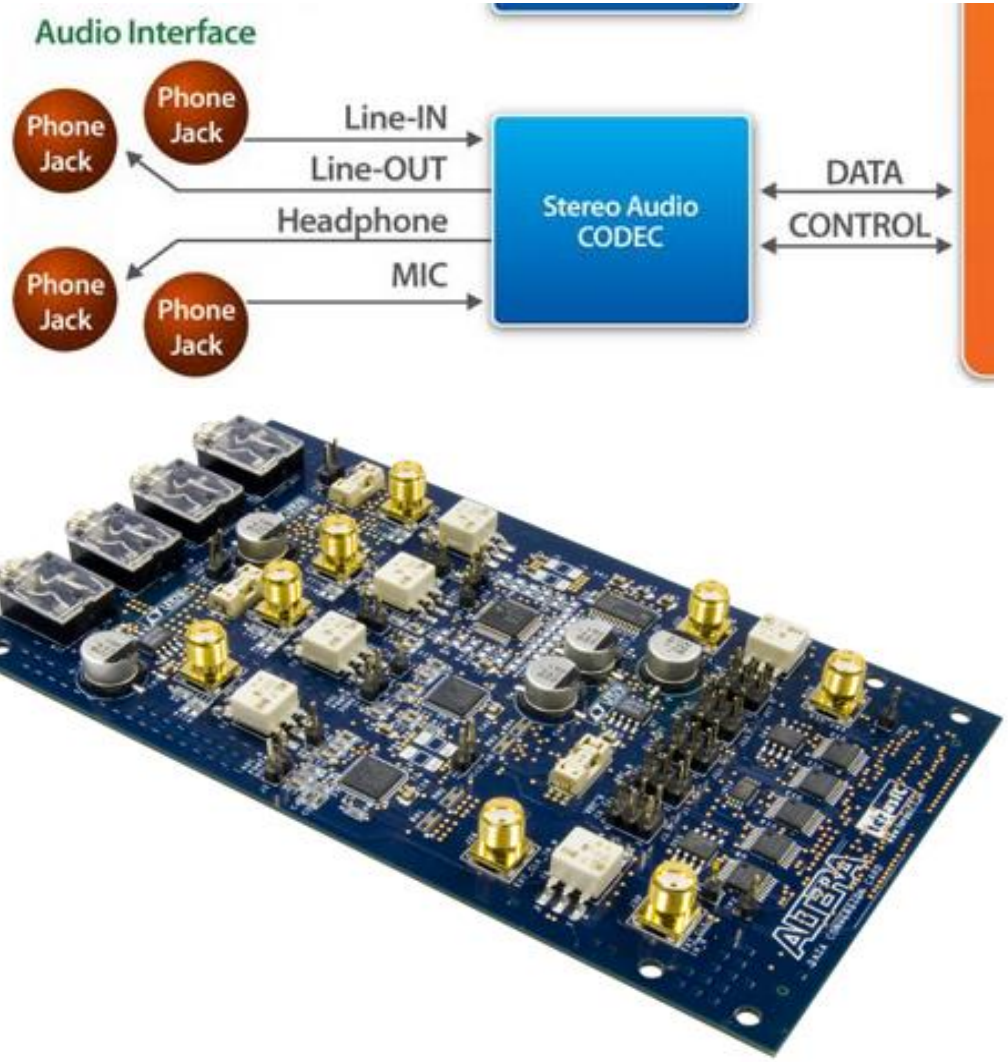
AIC 23

Analog Interface Controller

Stereo Audio Codec



- Texas Instruments TLV320AIC23B on Daughter Card (in AIC Data Sheet of class website)
- The codec communicates using two serial channels, one to control the codec's internal configuration registers and one to send and receive digital audio samples (14 bits/sample) in a synchronous Serial Peripheral Interface (SPI) mode.

Analog Inputs of AIC23

- The TLV320AIC23 has stereo line inputs for the left and the right audio channels (RLINEIN and LLINEIN). Both line inputs have independently programmable volume controls and mutes (Both channels can also be locked to the same value (set RLS and LRS bits).
- The line-input gain is logarithmically adjustable from 12 dB to –34.5 dB in 1.5-dB steps. The full-scale range tracks linearly with analog supply voltage AVDD.
- The other input, MICIN (based on analog audio path control INSEL: 0=line, 1=mic) , is a high-impedance, low-capacitance mono input that is compatible with a wide range of microphones. It has a programmable volume control and a mute function.
- The MICIN (audio) signal path has two gain stages. The first stage (sidetone attenuation) has a 4-step volume control. The second (digital) stage has a software programmable gain (MIC boost) of 0 dB or 20 dB.
- Both line or mic inputs are filtered by an “anti-aliasing” filter before being analog-to-digital converted (ADC).

Analog Outputs of AIC23

- Two low-impedance **line outputs** (LLINEOUT and RLINEOUT), linearly with the analog supply voltage AVDD.
- The DAC outputs, line inputs, and the microphone signal are summed into the **line outputs**. These sources can be switched off independently. e.g.,
  - In bypass mode, the line inputs are routed to the line outputs, bypassing the ADC and the DAC.
  - If sidetone is enabled, the microphone signal is routed to both line outputs via a four-step programmable attenuation circuit.
  - The line outputs can be muted by either muting the DAC (analog) or soft muting (digital) and disabling the bypass and sidetone paths.
- The TLV320AIC23 has stereo headphone outputs (LHPOUT and RHPOUT), volume logarithmically adjustable from 6 dB to –73 dB in 1-dB steps.
- Both channels can be locked to the same value by setting the RLS and LRS bits.

Control of AIC23

- There are 10 control registers (plus reset reg.), each 16 bits wide.
- The address of a control register is 7 bits wide and the address of a register and its 9 bit content form a 16-bit control word
- Control implemented in aic23.c

ADDRESS	REGISTER
0000000	Left line input channel volume control
0000001	Right line input channel volume control
0000010	Left channel headphone volume control
0000011	Right channel headphone volume control
0000100	Analog audio path control
0000101	Digital audio path control
0000110	Power down control
0000111	Digital audio interface format
0001000	Sample rate control
0001001	Digital interface activation
0001111	Reset register

Left Channel Headphone Volume Control (Address: 0000010)

BIT	D8	D7	D6	D5	D4	D3	D2	D1	D0
Function	LRS	LZC	LHV6	LHV5	LHV4	LHV3	LHV2	LHV1	LHV0
Default	0	1	1	1	1	1	0	0	1

LRS — Left/right headphone channel simultaneous volume/mute update
Simultaneous update 0 = Disabled 1 = Enabled

LZC — Left-channel zero-cross detect
Zero-cross detect 0 = Off 1 = On

The volume-control values are updated only when the input signal is close to the analog ground level.

LHV[6:0] — Left Headphone volume control (1111001 = 0 dB default)
1111111 = +6 dB, 79 steps between +6 dB and –73 dB (mute), 0110000 = –73 dB (mute), any thing below 0110000 does nothing – you are still muted

Analog Audio Path Control (Address: 0000100)

BIT	D8	D7	D6	D5	D4	D3	D2	D1	D0
Function	X	STA1	STA0	STE	DAC	BYP	INSEL	MICM	MICB
Default	0	0	0	0	1	1	0	1	0

STA[1:0] Sidetone attenuation 00 = –6 dB 01 = –9 dB 10 = –12 dB 11 = –15 dB
STE Sidetone enable 0 = Disabled 1 = Enabled
DAC DAC select 0 = DAC off 1 = DAC selected
BYP Bypass 0 = Disabled 1 = Enabled
INSEL Input select for ADC 0 = Line 1 = Microphone
MICM Microphone mute 0 = Normal 1 = Muted
MICB Microphone boost 0=OdB 1 = 20dB

4-step gain

X Reserved

Digital Audio Path Control (Address: 0000101)

BIT	D8	D7	D6	D5	D4	D3	D2	D1	D0
Function	X	X	X	X	X	DACM	DEEMP1	DEEMP0	ADCHP
Default	0	0	0	0	0	0	1	0	0

DACM DAC soft mute 0 = Disabled 1 = Enabled
DEEMP[1:0] De-emphasis control 00 = Disabled 01 = 32 kHz 10 = 44.1 kHz 11 = 48 kHz
ADCHP ADC high-pass filter 0 = Disabled 1 = Enabled
X Reserved

Summary of I/O

- Inputs: Lines (L/R) or MIC (by INSEL)
  - Lines (L/R): separate gain/mute control (32 steps) or can be simultaneously gain/mute update
  - MIC: gain control by analog 4-step sidetone attenuation (STA) or digital 2-step MICB
- Outputs: Headphones (L/R) and Lines (L/R, by analog audio path control)
  - Headphones (L/R): separate 79-step volume/mute control, or can be simultaneously gain/mute updated
  - Lines (L/R): Bypass enabled (LineOut+=LineIn), Sidetone enabled (LineOut+=MicIn), DAC only (regular data out, disable sidetone and bypass)

Configuring AIC23 Registers

- Use datasheet to determine appropriate bits and assign to global variable.

```
static int AIC23_Configuration = {
   0x0017,                  /* 0  Left line input channel volume */
   0x0017,                  /* 1  Rig23ht line input channel volume */
   0x00d8,                  /* 2  Left channel headphone volume */
   0x00d8,                  /* 3  Right channel headphone volume */
   0x0012,   (00010010)     /* 4  Analog audio path control */
   0x0000,                  /* 5  Digital audio path control */
   0x0000,                  /* 6  Power down control */
   0x0043,                  /* 7  Digital audio interface format */
   0x0081,                  /* 8  Sample rate control */
   0x0001                   /* 9  Digital interface activation */
};
```

Configuring AIC23 Registers

- You can reconfigure in AIC23 by resetting the appropriate index values of the AIC_configuration array and calling:

AIC23_config(int address, int setting);

address: the register address.

setting: new values (always in Hex form: 0x0042, 0x0031, etc)

Set Sampling Frequency

- Call the AIC23_setFreq(int freq); function to modify sampling frequency
- With frequency being one of the following
- Default is 32kHz (or you can change it by yourself in the global variable section).

```
/* Frequency Definitions */
#define AIC23_FREQ_8KHZ 0x000C
#define AIC23_FREQ_32KHZ 0x0019
#define AIC23_FREQ_44KHZ 0x0023    (44.1 KHz)
 #define AIC23_FREQ_48KHZ 0x0001
```

Sample Rate Control (Address: 0001000)

BIT	D8	D7	D6	D5	D4	D3	D2	D1	D0
Function	X	CLKOUT	CLKIN	SR3	SR2	SR1	SR0	BOSR	USB/Normal
Default	0	0	0	1	0	0	0	0	0

CLKIN Clock input divider 0 = MCLK 1 = MCLK/2

CLKOUT Clock output divider 0 = MCLK 1 = MCLK/2

SR[3:0] Sampling rate control (see Sections 3.3.2.1 AND 3.3.2.2)

BOSR Base oversampling rate

USB mode: 0 = 250 f_s 1 = 272 f_s

Normal mode: 0 = 256 f_s 1 = 384 f_s

USB/Normal Clock mode select: 0 = Normal 1 = USB

X Reserved

In the USB mode, the following ADC and DAC sampling rates are available:

SAMPLING RATE†		FILTER TYPE	SAMPLING-RATE CONTROL SETTINGS				
ADC (kHz)	DAC (kHz)		SR3	SR2	SR1	SR0	BOSR
96	96	3	0	1	1	1	0
88.2	88.2	2	1	1	1	1	1
48	48	0	0	0	0	0	0
44.1	44.1	1	1	0	0	0	1
32	32	0	0	1	1	0	0
8.021	8.021	1	1	0	1	1	1
8	8	0	0	0	1	1	0
48	8	0	0	0	0	1	0
44.1	8.021	1	1	0	0	1	1
8	48	0	0	0	1	0	0
8.021	44.1	1	1	0	1	0	1

† The sampling rates are derived from the 12-MHz master clock. The available oversampling rates do not produce exactly 8-kHz, 44.1-kHz, and 88.2-kHz sampling rates, but 8.021 kHz, 44.117 kHz, and 88.235 kHz, respectively. See Figures 3−17 through 3−34 for filter responses

MCLK = 12.288 MHz

SAMPLING RATE		FILTER TYPE	SAMPLING-RATE CONTROL SETTINGS				
ADC (kHz)	DAC (kHz)		SR3	SR2	SR1	SR0	BOSR
96	96	2	0	1	1	1	0
48	48	1	0	0	0	0	0
32	32	1	0	1	1	0	0
8	8	1	0	0	1	1	0
48	8	1	0	0	0	1	0
8	48	1	0	0	1	0	0

MCLK = 11.2896 MHz

SAMPLING RATE		FILTER TYPE	SAMPLING-RATE CONTROL SETTINGS				
ADC (kHz)	DAC (kHz)		SR3	SR2	SR1	SR0	BOSR
88.2	88.2	2	1	1	1	1	0
44.1	44.1	1	1	0	0	0	0
8.021	8.021	1	1	0	1	1	0
44.1	8.021	1	1	0	0	1	0
8.021	44.1	1	1	0	1	0	0

AIC23 Variables and Library Functions

- unsigned int aic23_demo[10] =

{0x0017, 0x0017, 0x01f9, 0x01f9, 0x0012, 0x0000, 0x0000, 0x0042, 0x0019, 0x0001};

Array with ten elements which stores the configurations info for each control register.

AIC23 Variables and Library Functions

- void spi_send(unsigned int address, unsigned int data);

Function performs SPI communication protocol to place one value into one AIC23's control register.

- void AIC23_demo();

Function runs spi_send() function ten times to refresh all control register values with the newest content stored in aic23_demo[10] array.

- void AIC_setFreq(int sampleRate);

Function specifically uses to configure sampling frequency rate.

Our Defined AIC23 I/O Functions for Our Board

void IOWR_ALTERA_AVALON_PIO_DATA(BASE_ADDRESS, value);
Library function to write value to a parallel port. Please note that some parallel port has 8 bits input but some only has 1 bit.

int IORD_ALTERA_AVALON_PIO_DATA(BASE_ADDRESS);
Library function to read and return the value from a parallel port.

void IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BASE_ADDRESS, value);
Library function to reset edge capture bit (usually used in ISR).

int IORD_ALTERA_AVALON_PIO_EDGE_CAP(BASE_ADDRESS);
Library function to read edge capture bit (usually used in ISR).

Example: Modify AIC sampling frequency in software

This example shows how to change the sampling frequency by configuring AIC control register.

```
static void handle_key1_interrupt(void* context, alt_u32 id) {
   volatile int* key1ptr = (volatile int *)context;
   *key1ptr = IORD_ALTERA_AVALON_PIO_EDGE_CAP(KEY1_BASE);
   IOWR_ALTERA_AVALON_PIO_EDGE_CAP(KEY1_BASE, 0);
   setFreqFlag = 1;
}

int main(void) {
  system_initialization();
  AIC23_demo();
  /*Your main infinity while loop*/
  while(1){
    if(setFreqFlag  == 1){
       AIC_setFreq(AIC23_FREQ_8KHZ);
       setFreqFlag = 0;
    }
  }
  return 0;
}
```

Initialize and configure the system

Set value to all AIC control registers

Key1 ISR to set 'setFreqFlag'

User triggers the modification by pushing key1

This is a "#define" value

Reset 'setFreqFlag' flag.

Note: You can change the if statement condition by yourself to trigger the sampling frequency modification.

Please also make sure you have all global variables declared.

Example: Modify AIC sampling frequency in real time.

This example shows how to modify sampling frequency with different value in real time.
Here we use two switches to represent four different cases.

```
void updateFreq(){
   if(IORD_ALTERA_AVALON_UART_RXDATA(SWITCH0_BASE) == 0){
      if(IORD_ALTERA_AVALON_UART_RXDATA(SWITCH1_BASE) == 0){
            sampleFrequency = 0x000C; //8k
      } else {
            sampleFrequency = 0x0019;//32k
      }
    } else {
      if(IORD_ALTERA_AVALON_UART_RXDATA(SWITCH1_BASE) == 0){
           sampleFrequency = 0x0023;//44.1k
      } else {
           sampleFrequency = 0x0001; //48k
      }
   }
}
```

Function use to update sampling rate global variable base on the switch0 and switch1 type
switch0, switch1

0 0-> 8KHz
0 1-> 32KHz
1 0-> 44.1KHz
1 1-> 48KHz

```
int main(void) {
      system_initialization();
      AIC23_demo();
      /*Your main infinity while loop*/
      while(1){
      updateFreq();
      /*If a user request to update the sampling frequency*/
      if(setFreqFlag == 1){
          aic23_demo[8] = sampleFrequency;
          AIC23_demo();
          setFreqFlag = 0;
}
```

Set control value to the array.

Update configuration to AIC

In the while(1) loop the system will continouesly update the value of “sampleFrequency” global variable based on two switches types. If you need to modify the sampling frequency, you need to firstly change switches type, after, press key1 to trigger the setting.

//Sine8_LED.c Sine generation with DIP switch control

Switch on the DIP switch #0, which should light LED #0 on and generate an 1kHz tone.

```
#include "system_init.h"
short loop = 0;                                          //table index
short gain = 10;                                         //gain factor
short sine_table[8]={0,707,1000,707,0,-707,-1000,-707};  //sine values

void main()
{
    sampleFrequency = 0x000C;                            //8k
    system_initialization();                             //init LED and switches
    while(1)                                             //infinite loop
    {
            if(IORD_ALTERA_AVALON_PIO_DATA(SWITCH0_BASE) == 1)                    // if DIP switch #0 on
            {
                    IOWR_ALTERA_AVALON_PIO_DATA(LED_BASE, 0x01);                  //turn LED #0 ON
                    IOWR_ALTERA_AVALON_PIO_DATA(LEFTSENDDATA_BASE, gain*sine_table[loop]);   //output to AIC23 D/A
                    if (loop < 7) ++loop;                                         //check for end of table
                    else loop = 0;                                                //reinit loop index
            }
            else IOWR_ALTERA_AVALON_PIO_DATA(LED_BASE, 0x00);                     //turn LED off if not pressed
    }                                                                             //end of while (1) infinite loop
}                                                                                 //end of main
```

$$f = \frac{F_s}{number\ of\ points}$$

Change sampling freq., output freq., gain, DIP switch, LED #

Example: Use interrupt to collect , store and play back audio data

When properly configure the interrupt, this ISR will be running when a new right channel data is ready. Please make sure you have all global variables declared.

Your ISR:

```
static void handle_rightready_interrupt_test(void* context, alt_u32 id) {
   volatile int* rightreadyptr = (volatile int *)context;
   *rightreadyptr = IORD_ALTERA_AVALON_PIO_EDGE_CAP(RIGHTREADY_BASE);
   IOWR_ALTERA_AVALON_PIO_EDGE_CAP(RIGHTREADY_BASE, 0);
   /*******Read, playback, store data*******/
   rightChannel = IORD_ALTERA_AVALON_PIO_DATA(RIGHTDATA_BASE);
   IOWR_ALTERA_AVALON_PIO_DATA(RIGHTSENDDATA_BASE, rightChannel);
   rightChannelData[rightCount] = rightChannel;
   rightCount = (rightCount+1) % BUFFERSIZE;
}
```

Your main:

```
int main(void) {
  system_initialization();
  AIC23_demo();
  /*Your main infinity while loop*/
  while(1){
  }
  return 0;
}
```

Initialize and configure the system

Set value to all AIC control registers

IORD_ALTERA_AVALON_PIO_DATA(RIGHTDATA_BASE);
Function that return the audio data from the AIC chip.

IOWR_ALTERA_AVALON_PIO_DATA(RIGHTSENDDATA_BASE, rightChannel);
Function that writes a digit data to the AIC chip.

rightChannelData[rightCount] = rightChannel;
Store data into an integer ring buffer.

rightCount = (rightCount+1) % BUFFERSIZE;
Update rightCount value.

Limitations

- Raw data from the AIC23 and processed on the NIOS II Processor has no speed limitations
- Due to data transmission limits of UART at 115200bps, the data being sent back to the PC (Matlab) must be limited to ~1k integer/second for continuous transmissions (OK for one-time transfer only).
- Possible workarounds include using MicroC OS and TCIP/IP to implement data transmission to LabVIEW with NO data loss
- This requires more work to learn but significantly improved results

External Events Discovery

- Polling
  - Check for new data on a regular basis
  - Main program has to integrate polls into the main loop
  - Simpler but less efficient than interrupt.
- Interrupt
  - Codec notifies NIOS II when ready for input or output
  - After the interrupt occurs, current values such as registers are pushed to stack and it jumps to ISR (interrupt service routine), usually done with a vector.asm file
  - When the interrupt is fulfilled, values in stack are popped back and the program continues.
  - Main program doesn’t have to be aware of what is going on with interrupts

NIOS II Interrupts

- Interrupt: an external event that stop the current CPU work and go to a specific interrupt service routine to serve the interrupt event.
- ISR: Interrupt service routine is a function to serve the interrupt event.
- Vector table: CPU has a specific hard-wired memory address called the interrupt vector table. The interrupt vector table is an array of function pointers which point to different ISR.

AIC23 as External Event

- Determine if you will use interrupts or polling
- Interrupts are much faster and are much more accurate than polling
- When new data is ready to be received or sent to the AIC23, the system will enter the interrupt handler
- You can
  - Configure and enable the appropriate interrupts
  - Receive data from left, right, or both channels of Line In or MIC
  - Output data to the left, right, or both channels of Line Out or Headphones
- See EE443 AIC23 API Manual & NIOS Documentation for specific functionality

NIOS II Interrupts Setup

Please refer to “interrupt setup tutorial” and other interrupt example we provide.