Collect stereos audio data from AIC23 chip

How data being sent:

In the hardware level, AIC23 chip samples the data with its designed sampling rate and the digital signal is transferred back to the chip through I2S communication protocol.

3.3.1.3 I²S Mode

In I²S mode, the MSB is available on the second rising edge of BCLK, after the falling edge on LRCIN or LRCOUT (see Figure 3-7).

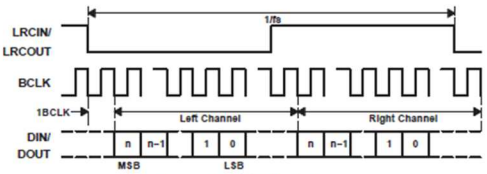

Figure 3–7. I²S Mode Timing

Brief explain in I2S: When LRC line (Left-right channel) changes from high to low. Left channel data will be ready and clocked into the CPU memory through BCLK from MSB to LSB. Once the LRC line changed from low to high. Right channel data will be ready and clocked in by the similar process. The total length of a single sample is 16 bit long (-32767 <= value range <= 32767).

Store Data:

In the hardware level, the program itself handles the I2S data transmission and data from two different channels will be stored into two 16-bit registers. The register values will be refresh based on the ADC sampling frequency.

Data ready:

Once all 16 bit data are collected completely, a pulse signal will be generated and we will treat this pulse signal as an interrupt source to tell the CPU to collect data. Please be sure that every pulse signal represents one sample data. If the sampling frequency is 48 KHz, there will be 48k pulse signal being generated for each channel, total at 96k pulses will be generated in one second.

Pulse signal name:

Leftready

Rightready

Data collection in software:

The software, you C code, will be using two pulse signals (leftready, rightready) as two interrupt sources to trigger interrupt event. Once an interrupt event happens, CPU will stop their normal task and go to the specific interrupt

service routine (ISR) to do some specific jobs. You will write your code to poll out the data from 16-bit registers for each individual channel.

Here is simple ISR example:

```
static void handle_switch0_interrupt(void* context, alt_u32 id) {
        volatile int* switch0ptr = (volatile int *)context;
        *switch0ptr = IORD_ALTERA_AVALON_PIO_EDGE_CAP(SWITCH0_BASE);

        /* Write to the edge capture register to reset it. */
        IOWR_ALTERA_AVALON_PIO_EDGE_CAP(SWITCH0_BASE, 0);

        /*Perform Jobs*/
       IOWR_ALTERA_AVALON_PIO_EDGE_CAP(LED_BASE, 0x01);

}
```

The ISR is named specify that this function will be called when the switch0 interrupt happens. If we properly register and enable the interrupt for switch0, this ISR will be called when you toggle switch0. The code you put under the comment /*Perform Jobs*/ is the task you want the ISR to fulfill. In this example, when you toggle swtich0, the first green LED light will be turned on.

Data storage: Ring buffer

In our package, we use a ring buffer to store value from each channel. The ring buffer structure looks like:

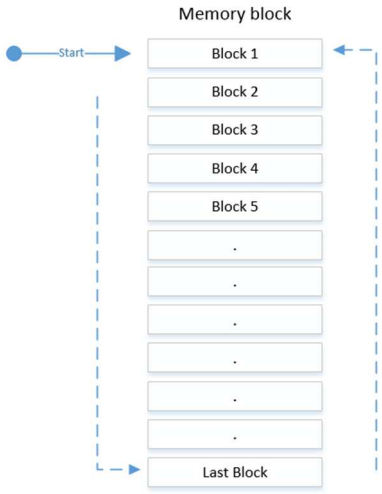

Figure. Ring buffer structure

In the software level, we create an integer array with a fix size equals to the constant value “BUFFERSIZE”. When the CPU detects and serves the ready signal, the ISR keep reading data from the 16-bit register and store it to the integer array. After each storing the array index number will be incremented by 1 so the next data will be stored in

the next index. Once the buffer is full we reset the index number back to 0 and start overwrite the old data when a new data is being collected.

Software implementation:

Define a constant value:

```
#define BUFFERSIZE 256
```

Declare an integer variable in C:

```
alt_16 leftChannel;
```

Declare an integer array in C:

```
alt_16 leftChannelData[BUFFERSIZE];
alt_16 rightChannelData[BUFFERSIZE];
```

Read 16-bit data from 16-bit registers (inside ISR):

```
leftChannel = IORD_ALTERA_AVALON_PIO_DATA(LEFTDATA_BASE);
```

Store data into the array (inside ISR):

```
leftChannelData[leftCount] = leftChannel;
```

Increment the array index value and reset index value when the buffer is full(inside ISR):

```
leftCount = (leftCount + 1) % BUFFERSIZE
```

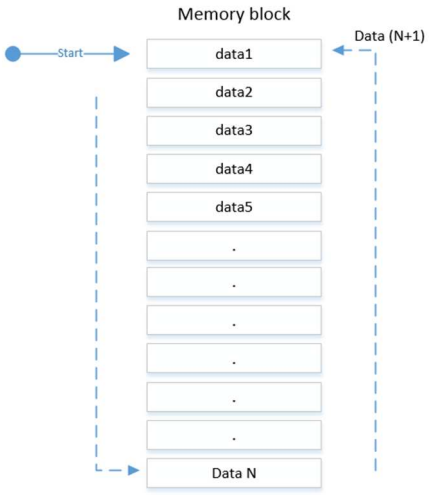

Figure. Data storage process

As the figure demonstrates above. Both the integer array (leftChannelData[], rightChannelData[]) and index number (leftCount, rightCount) are declare as global variables (on the very top of your file, please refer to the example code). Global variables can be accessed and modified by any other functions within that c file.

Anytime you want to have the previous signal flame you can always refer to the index number. For example, you want to have a copy of previous 256 left channel ADC sample data, and now the index number leftCount is 100. Then the data from 101->255->0->100 is the data you want.

Play back to Lineout:

When we want to play the signal back, we need to send the data from our CPU to the hardware level and send it back to the AIC23 chip through I2S communication protocol. The hardware level stuff has been ready so students only need to upload the data to a hardware 16 bits registers:

```
IOWR_ALTERA_AVALON_PIO_DATA(LEFTSENDDATA_BASE, leftChannel);
```

LEFTSENDDATA_BASE is the interface that connects to the hardware, leftChannel is the integer value that is going to be uploaded. In the example we provided, we collect the data from the AIC23 chip and immediately send back that data.

Overall data flow chart:

Data is sent from AIC23 and being collected by the hardware in DE2i-150 board. Data from each channel is temporarily stored in a 16 bits register (LEFTDATA_BASE, RIGHTDATA_BASE). Once a full 16 bits data has been store into the register, a pulse will then be generated and sent to the CPU (In our case, the FPGA). Students will write their own code to register and enable the interrupt function to detect the left,right ready signal, and write the interrupt service routine (ISR) to serve the interrupt. The main purpose of these two ISRs are to collect the data, store it to the ring buffer and play it back to Lineout.

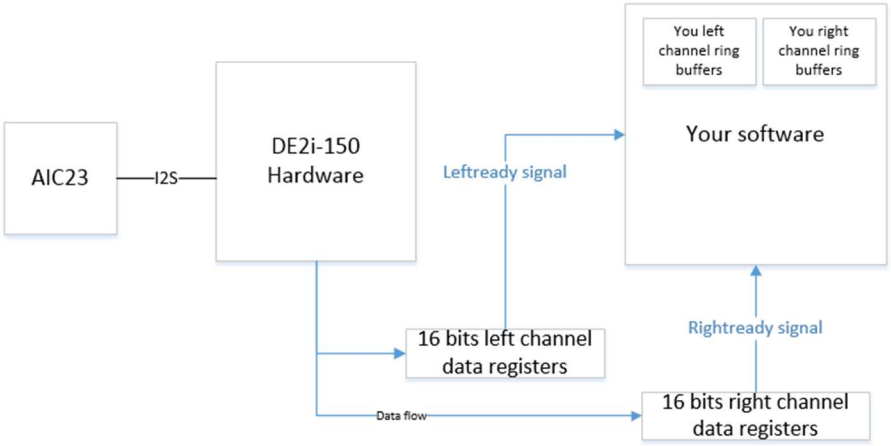

Figure. Data flow process