Digital Signal Processing LAB
LAB 2 (Report)

Mohammad Parsa Dini - std id: 400101204
Mohammad Hossein Momeni - std id: 99102359

Introduction

In this lab, we implemented an interrupt-based system for real-time audio signal processing. The DSP processor was configured to read an audio signal from the microphone, amplify it using a predefined gain factor, and then send the processed signal to the headphones. This process was handled using software interrupts, ensuring efficient and timely processing of incoming audio data.

Necessity of Interrupts in Signal Processing

In real-time digital signal processing, using interrupts is essential because processors cannot simply remain idle while waiting for new data. Instead of constantly checking for new input, interrupts allow the system to efficiently handle data as soon as it arrives. This prevents unnecessary CPU usage and ensures timely execution of critical tasks.

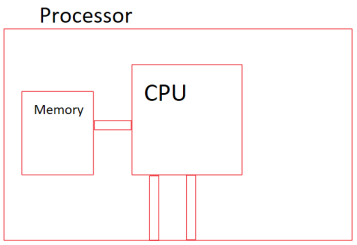

Figure 1

Interrupt Configuration Function

we also used a function named `hook_int` which:

1. **disables global interrupts** to prevent interference during configuration,
2. **Enables Non-Maskable Interrupts** `NMI` which are high-priority interrupts,
3. **Maps the serial port receive interrupt** to interrupt line 15.
4. **Enables the** `RINT2` **interrupt**, which is triggered when new audio data arrives.
5. **Re-enables global interrupts** to allow normal operation.

```
    void hook_int(){
        IRQ_globalDisable();    // Disable global interrupts
        IRQ_nmiEnable();        // Enable Non-Maskable Interrupts (NMI)
        IRQ_map(IRQ_EVT_RINT2, 15); // Map RINT2 (Receive Interrupt 2) to
            interrupt line 15
        IRQ_enable(IRQ_EVT_RINT2);  // Enable the receive interrupt (RINT2)
        IRQ_globalEnable();    // Re-enable global interrupts
    }
```

Interrupt Service Routine

The `serialPortRcvISR()` function is an interrupt service routine that processes incoming audio data. It reads a sample from the codec, amplifies it by multiplying with `volumeGain`, and writes the modified sample back to the codec. This ensures real-time signal processing with minimal CPU usage.

```
interrupt void serialPortRcvISR() {
    Uint32 temp;
    DSK6416_AIC23_read(hCodec, &temp);  // Read an audio sample from the
        codec
    temp = (temp * volumeGain);         // Apply volume gain (amplify the
        sample)
    //temp = temp & 0xFF00;  // (Commented out) Optional bit-masking
    DSK6416_AIC23_write(hCodec, temp);  // Write the processed sample back
        to the codec
}
```

Implementation

We configured the DSP board and set the input source to the microphone. Using the `hook_int()` function, we disabled global interrupts, enabled non-maskable interrupts for critical signals, mapped the serial port receive interrupt to line 15, and activated it for new audio data arrivals. The `serialPortRcvISR()` function managed the interrupt service routine by reading incoming audio

samples from the codec, amplifying them with a gain factor, and sending the amplified signal to the output. The continuous `while(1)` loop kept the processor active and efficiently handled incoming data using interrupts.

```
int main(){
    MBZIO_init(DSK6416_AIC23_FREQ_8KHZ); // Initialize DSP system at 8 kHz
        sample rate
    MBZIO_set_inputsource_microphone();  // Set input source to microphone

    //volumeGain = 3; // (Commented out) Set the volume gain factor

    hook_int();  // Configure and enable interrupts

    while(1){ }  // Infinite loop (processor waits for interrupts)
}
```

We also used this line of code since our microphone is not stereo but the handsfrees are stereo, so we masked half of the 32 bits in order to determine which of the MSB and LSB corresponds to which handsfree. We found that the MSB corresponds to the left handsfree. We also set a Gel file gain in order to adjust the volume.

```
    temp = ( temp * volumeGain ) & 0xFF00;
```