

# Lab 2

## Interrupts and Stereo Outputs

ECE 406: Real-Time Digital Signal Processing

February 23, 2015

Christopher Daffron

[cdaffron@utk.edu](mailto:cdaffron@utk.edu)

There were three objectives for this lab. The first one was to learn how to use interrupts. The second was to explore some additional features of the DSK such as generating and outputting a stereo output. Finally, the third objective was to prepare for lab 3 by implementing a simple sum-of-product operation.

## Task 1

The primary purpose of this task was to become more familiar with outputting stereo signals. To achieve this, the code for `sine_stereo.c` from the Chassaing book was used. This code creates two sinusoids and outputs one to each of the channels. It outputs a 1kHz wave to one of the channels and a 3 kHz wave to the other one. After creating the project and building it, I ran it on the DSK board while connected to an oscilloscope. The two sinusoids were clearly visible and had the correct frequencies, as shown in the picture below.



Figure 1: 2 channel oscilloscope output

## Task 2

The first step of the second task was to build and run the project and view the results on the oscilloscope. This code adds together two sinusoids and outputs them through one of the channels. The resulting signal is shown below.



Figure 2: Single channel DTMF output

Next, I modified the source code to output each of the individual frequencies in the DTMF tone separately to the channels. While implementing this, I did run into one issue. Using the API calls `output_left_sample` and `output_right_sample` separately did not give the correct results. The likely reason for this is that the codec would output the first sample received and then not be ready for the next sample for the other channel to be sent. By sending the two calls individually, I was effectively trying to get the codec to output at 32 kHz even though it was only running at 16 kHz. To fix this, I used the `output_sample` function that outputs both samples simultaneously, which is the method used in task 1. The results of this are shown below and the individual frequencies of 1447 Hz and 697 Hz are clearly visible.



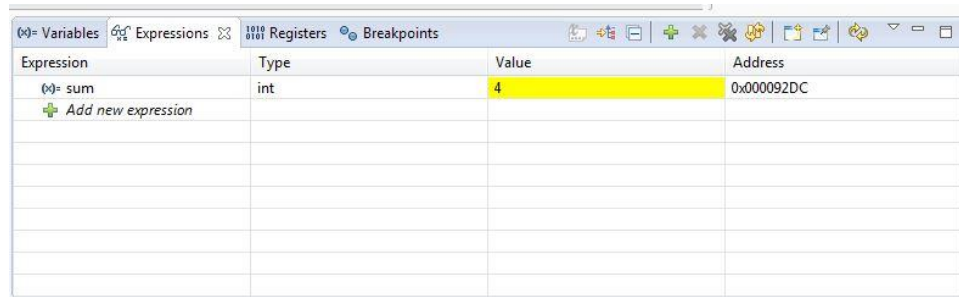
Figure 3: Separated DTMF waveforms

Next, in step 3 of task 2, I implemented code to output one of the 12 DTMF tones for 2 seconds based on the state of the DIP switches. I realized that there was one issue with using the DIP switches to output the number and then keeping the output constant for 2 seconds. The code has no way of knowing if the current state of the DIP switches is an intermediate state while entering a number or a final state of a number that needs to be outputted. To handle this, I included code that only outputs a number for 2 seconds once the DIP switches have been in a constant state for 2 seconds. This is similar to debounce logic when working with buttons connected to digital logic. To make it easier for the user, I also reversed the order of the bits on the DIP switches so that DIP switch 4 is used to set bit 0 of the number. This makes it so that the rightmost DIP switch is used to set the least significant bit of the input number and the bits are displayed by the DIP switch states in the correct order. After implementing this, I experimented with all of the combinations of DIP switches and listened to the generated tone and compared it to my phone to verify that the correct tone is created.

## Task 3

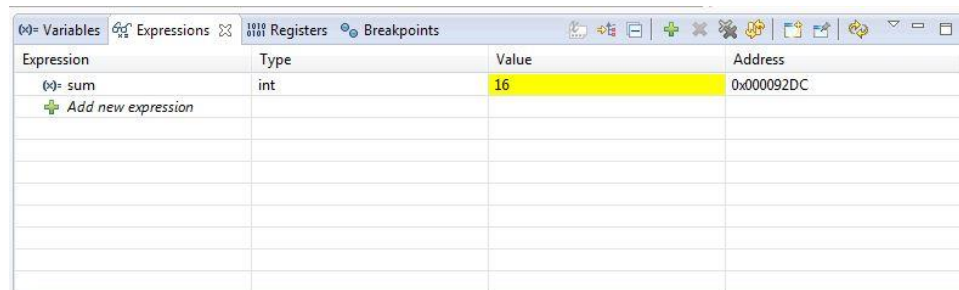
The primary goal of the third task is to implement a sum-of-products (SOP) and experiment with some of the more advanced features of Code Composer Studio (CCS). To begin, I created a project based around the dotp4.c code from the Chassaing book and built and ran it. I created a variable watch on the sum variable, created a breakpoint when it is changed, and then watched as the variable changed on successive iterations of the loop. The various iterations are

shown below. Note that the first iteration is not shown because it adds 0 to the variable which results in no change.



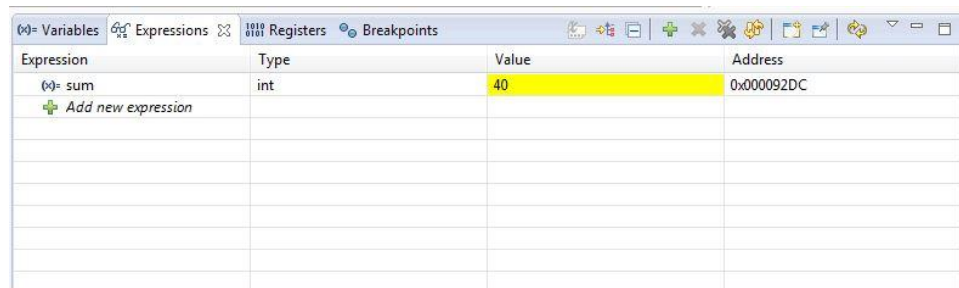
Expression	Type	Value	Address
(*) sum	int	4	0x000092DC
+ Add new expression			

Figure 4: Sum variable after second iteration



Expression	Type	Value	Address
(*) sum	int	16	0x000092DC
+ Add new expression			

Figure 5: Sum variable after third iteration



Expression	Type	Value	Address
(*) sum	int	40	0x000092DC
+ Add new expression			

Figure 6: Sum variable after fourth iteration, final state

After viewing the changes in the sum variable, I used the profile clock function to measure the number of cycles taken for computing this value. As shown in the figure below, the execution of the summing function took 183 cycles when no optimization is enabled. With each cycle taking 4.44 ns, this equates to 812.52 ns. The clock cycle view is shown below.



Figure 7: Clock cycles with no optimization

After measuring the number of clock cycles with no optimization enabled, I recompiled with the optimization level set to 2. This resulted in 116 clock cycles, much less than the number with optimization disabled. At 4.44 ns per cycle, this equates to 515.04 ns. The clock cycle view is shown below.



Figure 8: Clock cycles with level 2 optimization

## Conclusion

In this lab, I learned about how to output stereo signals, how DTMF signaling works, how to implement SOP, and how to use some of the more advanced features of CCS. After completing this lab, I feel like I better understand how stereo signals are outputted and how the codec works with them. I also better understand how interrupts work and why they are much more efficient from a processing standpoint when compared to a polling technique.

## Appendix

### Task 1

#### sine\_stereo.c

```
//sine_stereo Sine generation to both LEFT and RIGHT channels

#include "dsk6713_aic23.h"           //codec support
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //set sampling rate
#define DSK6713_AIC23_INPUT_MIC 0x0015
#define DSK6713_AIC23_INPUT_LINE 0x0011
Uint16 inputsource=DSK6713_AIC23_INPUT_MIC; // select input

#define LEFT 0
#define RIGHT 1
union {Uint32 uint; short channel[2];} AIC23_data;

#define LOOPLength 8                // size of look up table
short sine_table_left[LOOPLength]={0,7071,10000,7071,0,-7071,-10000,-7071};
short sine_table_right[LOOPLength]={0,-7071,10000,-7071,0,7071,-10000,7071};
short loopindex = 0;                // look up table index

interrupt void c_int11()             //interrupt service routine
{
    AIC23_data.channel[RIGHT]=sine_table_right[loopindex]; //for right channel;
    AIC23_data.channel[LEFT]=sine_table_left[loopindex]; //for leftchannel;

    output_sample(AIC23_data.uint); //output to both channels
    if (++loopindex >= LOOPLength)
```

```

        loopindex = 0; // check for end of look up table
    return;
}

void main()
{
    comm_intr();           //init DSK,codec,McBSP
    while(1) ;             //infinite loop
}

```

## Task 2

### sinedtmf\_intr.c – Single channel DTMF output

```

//sinedtmf_intr.c DTMF tone generation using lookup table
#include "DSK6713_AIC23.h"           // codec support
#include "stdio.h"
Uint32 fs=DSK6713_AIC23_FREQ_16KHZ; //set sampling rate
#define DSK6713_AIC23_INPUT_MIC 0x0015
#define DSK6713_AIC23_INPUT_LINE 0x0011
Uint16 inputsource=DSK6713_AIC23_INPUT_MIC; // select input
#include <math.h>
#define PI 3.14159265358979

#define TABLESIZE 512              // size of look up table
#define SAMPLING_FREQ 16000
#define STEP_770 (float)(770 * TABLESIZE)/SAMPLING_FREQ
#define STEP_1336 (float)(1336 * TABLESIZE)/SAMPLING_FREQ
#define STEP_697 (float)(697 * TABLESIZE)/SAMPLING_FREQ
#define STEP_852 (float)(852 * TABLESIZE)/SAMPLING_FREQ
#define STEP_941 (float)(941 * TABLESIZE)/SAMPLING_FREQ
#define STEP_1209 (float)(1209 * TABLESIZE)/SAMPLING_FREQ
#define STEP_1477 (float)(1477 * TABLESIZE)/SAMPLING_FREQ
#define STEP_1633 (float)(1633 * TABLESIZE)/SAMPLING_FREQ

short sine_table[TABLESIZE];
float loopindexlow = 0.0;           // look up table index
float loopindexhigh = 0.0;
short i;

interrupt void c_int11()           //interrupt service routine
{
    output_left_sample((sine_table[(short)loopindexlow] + sine_table[(short)loopindexhigh])/10);
    loopindexlow += STEP_697;
    if (loopindexlow > (float)TABLESIZE)
        loopindexlow -= (float)TABLESIZE;
    loopindexhigh += STEP_1477;
    if (loopindexhigh > (float)TABLESIZE)
        loopindexhigh -= (float)TABLESIZE;
    return;                         //return from interrupt
}

void main()
{
    comm_intr();                   // initialise DSK
    printf("Hello\n");
    for (i=0 ; i< TABLESIZE ; i++)
        sine_table[i] = (short)(10000.0*sin(2*PI*i/TABLESIZE));
    while(1);
}

```



```
}
```

## sinedtmf\_intr.c – Separate channel DTMF output

```
//sinedtmf_intr.c DTMF tone generation using lookup table
#include "DSK6713_AIC23.h"           // codec support
#include "stdio.h"

Uint32 fs = DSK6713_AIC23_FREQ_16KHZ; //set sampling rate
#define DSK6713_AIC23_INPUT_MIC 0x0015
#define DSK6713_AIC23_INPUT_LINE 0x0011
Uint16 inputsource = DSK6713_AIC23_INPUT_MIC; // select input
#include <math.h>
#define PI 3.14159265358979

#define TABLESIZE 512           // size of look up table
#define SAMPLING_FREQ 16000
#define STEP_770 (float)(770 * TABLESIZE)/SAMPLING_FREQ
#define STEP_1336 (float)(1336 * TABLESIZE)/SAMPLING_FREQ
#define STEP_697 (float)(697 * TABLESIZE)/SAMPLING_FREQ
#define STEP_852 (float)(852 * TABLESIZE)/SAMPLING_FREQ
#define STEP_941 (float)(941 * TABLESIZE)/SAMPLING_FREQ
#define STEP_1209 (float)(1209 * TABLESIZE)/SAMPLING_FREQ
#define STEP_1477 (float)(1477 * TABLESIZE)/SAMPLING_FREQ
#define STEP_1633 (float)(1633 * TABLESIZE)/SAMPLING_FREQ

short sine_table[TABLESIZE];
float loopindexlow = 0.0;        // look up table index
float loopindexhigh = 0.0;
short i;

#define LEFT 0
#define RIGHT 1
union { Uint32 uint; short channel[2]; } AIC23_data;

interrupt void c_int11()         //interrupt service routine
{
    // set left channel data
    AIC23_data.channel[LEFT] = sine_table[(short)loopindexlow];
    // set right channel data
    AIC23_data.channel[RIGHT] = sine_table[(short)loopindexhigh];
    output_sample(AIC23_data.uint); // output data to separate channels
    loopindexlow += STEP_697;
    if (loopindexlow > (float)TABLESIZE)
        loopindexlow -= (float)TABLESIZE;
    loopindexhigh += STEP_1477;
    if (loopindexhigh > (float)TABLESIZE)
        loopindexhigh -= (float)TABLESIZE;
    return;                       //return from interrupt
}

void main()
{
    comm_intr();                 // initialise DSK
    printf("Hello\n");
    for (i = 0; i < TABLESIZE; i++)
        sine_table[i] = (short)(10000.0*sin(2 * PI*i / TABLESIZE));
    while (1);
}
```



## sinedtmf\_intr.c – DIP switch digit selection

```
//sinedtmf_intr.c DTMF tone generation using lookup table
#include "DSK6713_AIC23.h"           // codec support
#include "stdio.h"
Uint32 fs = DSK6713_AIC23_FREQ_16KHZ; //set sampling rate
#define DSK6713_AIC23_INPUT_MIC 0x0015
#define DSK6713_AIC23_INPUT_LINE 0x0011
Uint16 inputsource = DSK6713_AIC23_INPUT_MIC; // select input
#include <math.h>
#define PI 3.14159265358979

#define TABLESIZE 512           // size of look up table
#define SAMPLING_FREQ 16000
#define STEP_770 (float)(770 * TABLESIZE)/SAMPLING_FREQ
#define STEP_1336 (float)(1336 * TABLESIZE)/SAMPLING_FREQ
#define STEP_697 (float)(697 * TABLESIZE)/SAMPLING_FREQ
#define STEP_852 (float)(852 * TABLESIZE)/SAMPLING_FREQ
#define STEP_941 (float)(941 * TABLESIZE)/SAMPLING_FREQ
#define STEP_1209 (float)(1209 * TABLESIZE)/SAMPLING_FREQ
#define STEP_1477 (float)(1477 * TABLESIZE)/SAMPLING_FREQ
#define STEP_1633 (float)(1633 * TABLESIZE)/SAMPLING_FREQ

short sine_table[TABLESIZE];
float loopindexlow = 0.0;        // look up table index
float loopindexhigh = 0.0;
short i;

int inLoop = 0;
int toneCounter = 0;
int startCounter = 0;
int dip0state = 1;
int dip1state = 1;
int dip2state = 1;
int dip3state = 1;

#define LEFT 0
#define RIGHT 1
union { Uint32 uint; short channel[2]; } AIC23_data;

interrupt void c_int11() //interrupt service routine
{
    // output current staged sample
    output_left_sample((sine_table[(short)loopindexlow] + sine_table[(short)loopindexhigh]) /
10);

    // if currently outputting a digit for 2 seconds
    if (inLoop == 1)
    {
        if ((dip0state == 1) && (dip1state == 1) && (dip2state == 1) && (dip3state == 1)) // DIP
input = 0
        {
            // output a 0
            loopindexlow += STEP_941;
            if (loopindexlow > (float)TABLESIZE)
                loopindexlow -= (float)TABLESIZE;
            loopindexhigh += STEP_1336;
            if (loopindexhigh > (float)TABLESIZE)
                loopindexhigh -= (float)TABLESIZE;
        }
    }
}
```

```

else if ((dip0state == 1) && (dip1state == 1) && (dip2state == 1) && (dip3state == 0)) //
DIP input = 1
{
    // output a 1
    loopindexlow += STEP_697;
    if (loopindexlow > (float)TABLESIZE)
        loopindexlow -= (float)TABLESIZE;
    loopindexhigh += STEP_1209;
    if (loopindexhigh > (float)TABLESIZE)
        loopindexhigh -= (float)TABLESIZE;
}
else if ((dip0state == 1) && (dip1state == 1) && (dip2state == 0) && (dip3state == 1)) //
DIP input = 2
{
    // output a 2
    loopindexlow += STEP_697;
    if (loopindexlow > (float)TABLESIZE)
        loopindexlow -= (float)TABLESIZE;
    loopindexhigh += STEP_1336;
    if (loopindexhigh > (float)TABLESIZE)
        loopindexhigh -= (float)TABLESIZE;
}
else if ((dip0state == 1) && (dip1state == 1) && (dip2state == 0) && (dip3state == 0)) //
DIP input = 3
{
    // output a 3
    loopindexlow += STEP_697;
    if (loopindexlow > (float)TABLESIZE)
        loopindexlow -= (float)TABLESIZE;
    loopindexhigh += STEP_1477;
    if (loopindexhigh > (float)TABLESIZE)
        loopindexhigh -= (float)TABLESIZE;
}
else if ((dip0state == 1) && (dip1state == 0) && (dip2state == 1) && (dip3state == 1)) //
DIP input = 4
{
    // output a 4
    loopindexlow += STEP_770;
    if (loopindexlow > (float)TABLESIZE)
        loopindexlow -= (float)TABLESIZE;
    loopindexhigh += STEP_1209;
    if (loopindexhigh > (float)TABLESIZE)
        loopindexhigh -= (float)TABLESIZE;
}
else if ((dip0state == 1) && (dip1state == 0) && (dip2state == 1) && (dip3state == 0)) //
DIP input = 5
{
    // output a 5
    loopindexlow += STEP_770;
    if (loopindexlow > (float)TABLESIZE)
        loopindexlow -= (float)TABLESIZE;
    loopindexhigh += STEP_1336;
    if (loopindexhigh > (float)TABLESIZE)
        loopindexhigh -= (float)TABLESIZE;
}
else if ((dip0state == 1) && (dip1state == 0) && (dip2state == 0) && (dip3state == 1)) //
DIP input = 6
{
    // output a 6
    loopindexlow += STEP_770;
    if (loopindexlow > (float)TABLESIZE)

```

```

        loopindexlow -= (float)TABLESIZE;
        loopindexhigh += STEP_1477;
        if (loopindexhigh > (float)TABLESIZE)
            loopindexhigh -= (float)TABLESIZE;
    }
    else if ((dip0state == 1) && (dip1state == 0) && (dip2state == 0) && (dip3state == 0)) //
DIP input = 7
    {
        // output a 7
        loopindexlow += STEP_852;
        if (loopindexlow > (float)TABLESIZE)
            loopindexlow -= (float)TABLESIZE;
        loopindexhigh += STEP_1209;
        if (loopindexhigh > (float)TABLESIZE)
            loopindexhigh -= (float)TABLESIZE;
    }
    else if ((dip0state == 0) && (dip1state == 1) && (dip2state == 1) && (dip3state == 1)) //
DIP input = 8
    {
        // output a 8
        loopindexlow += STEP_852;
        if (loopindexlow > (float)TABLESIZE)
            loopindexlow -= (float)TABLESIZE;
        loopindexhigh += STEP_1336;
        if (loopindexhigh > (float)TABLESIZE)
            loopindexhigh -= (float)TABLESIZE;
    }
    else if ((dip0state == 0) && (dip1state == 1) && (dip2state == 1) && (dip3state == 0)) //
DIP input = 9
    {
        // output a 9
        loopindexlow += STEP_852;
        if (loopindexlow > (float)TABLESIZE)
            loopindexlow -= (float)TABLESIZE;
        loopindexhigh += STEP_1477;
        if (loopindexhigh > (float)TABLESIZE)
            loopindexhigh -= (float)TABLESIZE;
    }
    else if ((dip0state == 0) && (dip1state == 1) && (dip2state == 0) && (dip3state == 1)) //
DIP input = 10
    {
        // output a *
        loopindexlow += STEP_941;
        if (loopindexlow > (float)TABLESIZE)
            loopindexlow -= (float)TABLESIZE;
        loopindexhigh += STEP_1209;
        if (loopindexhigh > (float)TABLESIZE)
            loopindexhigh -= (float)TABLESIZE;
    }
    else if ((dip0state == 0) && (dip1state == 1) && (dip2state == 0) && (dip3state == 0)) //
DIP input = 11
    {
        // output a #
        loopindexlow += STEP_941;
        if (loopindexlow > (float)TABLESIZE)
            loopindexlow -= (float)TABLESIZE;
        loopindexhigh += STEP_1477;
        if (loopindexhigh > (float)TABLESIZE)
            loopindexhigh -= (float)TABLESIZE;
    }
    else // any other DIP input

```

```

    {
        // invalid input, output nothing
    }
    toneCounter++;
    // if outputted for 2 seconds, stop and change state
    if (toneCounter > 32000)
    {
        toneCounter = 0;
        inLoop = 0;
    }
}
else // if not currently outputting digit
{
    if ((DSK6713_DIP_get(0) == dip0state) && (DSK6713_DIP_get(1) == dip1state) &&
        (DSK6713_DIP_get(2) == dip2state) && (DSK6713_DIP_get(3) == dip3state))
    {
        // increment counter if state is the same as last cycle
        startCounter++;
    }
    else
    {
        // if state changed, reset counter, store state
        startCounter = 0;
        dip0state = DSK6713_DIP_get(0);
        dip1state = DSK6713_DIP_get(1);
        dip2state = DSK6713_DIP_get(2);
        dip3state = DSK6713_DIP_get(3);
    }

    // if state has been constant for 2 seconds, start outputting tone
    if (startCounter > 32000)
    {
        startCounter = 0;
        inLoop = 1;
        loopindexlow = 0.0;
        loopindexhigh = 0.0;
    }
}
return; //return from interrupt
}

void main()
{
    comm_intr(); // initialise DSK
    printf("Hello\n");
    for (i = 0; i < TABLESIZE; i++)
        sine_table[i] = (short)(10000.0*sin(2 * PI*i / TABLESIZE));
    while (1);
}

```

## Task 3

### dotp4.c

```

//dotp4.c dot product of two vectors

int dotp(short *a, short *b, int ncount); //function prototype
#include <stdio.h> //for printf
#define count 4 //# of data in each array

```

```

short x[count] = {1,2,3,4};    //declaration of 1st array
short y[count] = {0,2,4,6};    //declaration of 2nd array
int sum = 0;

main()
{
    int result = 0;              //result sum of products

    result = dotp(x, y, count); //call dotp function
    printf("result = %d (decimal) \n", result); //print result
}

int dotp(short *a, short *b, int ncount) //dot product function
{
    int i;
    sum = 0;
    for (i = 0; i < ncount; i++)
        sum += a[i] * b[i];      //sum of products
    return(sum);                 //return sum as result
}

```